Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

Formal Modeling Of Middleware-based Distributed Systems

Arnab Ray [1,2]

Computer Science, SUNY at Stony Brook, Stony Brook NY 11794-4400, USA

Rance Cleaveland [3]

Computer Science, SUNY at Stony Brook, Stony Brook NY 11794-4400, USA

Abstract

Effective design of middleware-based systems requires modeling notations that allow the use of process-interaction schemes provided by different middleware packages directly in designs. Traditional design notations typically only support a fixed class of interprocess interaction schemes, and designers wishing to use them for modeling middleware-based systems must devote significant effort to encoding the middleware primitives in the notation. In this paper, we demonstrate how a new graphical design notation, Architectural Interaction Diagrams (AIDs), which provides parameterized support for different interaction schemes, may be used to model a real-life middleware-based system like the Event Heap coordination infrastructure of the i-Room ubiquitous computing environment.

Key words: Software Architecture, Middleware, Formal Methods, Distributed Systems

1 Introduction

Pre-implementation modeling of distributed systems [1] enables the designer to study systems in a systematic fashion, without worrying about implementation details, in order to isolate design bugs early in the development cycle. Informal design notations (eg ball-and-stick type diagrams) despite their easy understandability have limited utility when one needs to mathematically analyze designs. Design formalisms are needed in order to specify properties the designed system should satisfy as well as verify these properties on the designs. In addition, formal models

[1] Research supported by Army Research Office grants DAAD190110003 and DAAD190110019, and National Science Foundation grants CCR-9988489 and CCR-0098037

[2] Email: `arnabray@cs.sunysb.edu`

[3] Email: `rance@cs.sunysb.edu`

of distributed systems may be simulated and implementation code generated from these simulate-able models. Middleware-based systems are ideal candidates for pre-implementation modeling because such highly concurrent distributed systems may hide subtle bugs that may not be captured by informal analysis.

One of the main features of middleware-based systems is that they usually are highly communication-intensive distributed systems supporting a wide variety of application synchronization mechanisms. Classical architecture description languages (ADLs) usually support one basic form of interaction and thus cannot naturally express the rich communication disciplines of middleware. Architectural Interaction Diagrams [16] (AIDs), however, provide a parameterized mechanism that naturally supports a wide variety of interprocess-communication and synchronization primitives as language-supported constructs. This frees the designer from having to simulate different IPCs by-hand: something he would have to do if she with traditional ADLs [2,5,15]. This leads to cleaner models, as there is no clutter due to communication code. Another benefit is that AIDs models have smaller state spaces, as each communication is now a single transition, as opposed to the multiple transitions needed when communication schemes are encoded using others.

In this paper, we illustrate the utility of our AIDs formalism with a study of the Event Heap Coordination Framework used by applications to coordinate themselves inside the i-Room [9], an experimental ubiquitous computing environment at Stanford University. We demonstrate how a model of the Event Heap may be rendered using our approach and why it is more efficient than conventional modeling techniques. In order to do this, we extend our work in [16] by introducing a remote procedure call transition into the syntax of the language. As middleware systems typically include event-handling functionality similar to that of the Event Heap, in studying the Event Heap, we make a case for the application of our formalism to the problem of modeling different kinds of middleware systems.

Related Work. Formal analysis has been used to model and verify systems such as the i-Protocol [4], network protocols like Rether [2] and e-commerce protocols. While little has been done on formalizing middleware-based systems, there has been work on semi-formal UML-based analysis using the UML profile for Enterprise Distributed Object Computing [13] and a notion of behavioral semantics called choreography. AIDs extends existing work by providing a full operational semantics that gives us the ability to execute as well as formally analyze designs.

There are many features inherent in enterprise middleware-based systems, such as QoS, which our formalism is able to handle at present. But by encoding communication naturally into the semantics of the language, we contend that AIDs offers a vital component for any toolbox for formally analyzing middleware-based systems.

2 Finite State System Representations

A distributed system may be looked upon as a collection of multiple agents or processes interacting continuously with each other. For a formal model of distributed systems, one needs to define a) a base formalism for representing agents, b) a notion of concurrency, d) a semantics for inter-agent communication and d) a concept of hierarchy. In this paper, we are concerned only with finite-state systems.

The base formalism for representing agents has traditionally been finite-state machines (FSMs). The role of a FSM is to simulate the execution of the agent it represents. When a system executes, it moves from one "state" to another and these "moves" are denoted by directed arrows called transitions. Each transition represents either an unit of interaction with the environment or an unit of the agent's own internal computation. A transition which represents an environment interaction is provided a label which is the name of the *port* or access point through which the agent interacts with the environment. A transition which represents internal computation is denoted by the "silent" action called τ. A transition may be an input or an output one (input and output with respect to the agent) with an output transition being denoted by $out(p, v)$[emit data value v on port p] and an input transition by $in(p, v)$[consume data value v on port p]

The notion of concurrency most popular with distributed systems is that of *interleaving semantics*. What this means is that the set of actions of the whole system consist of all the possible interleavings of the actions of the constituent agents. In

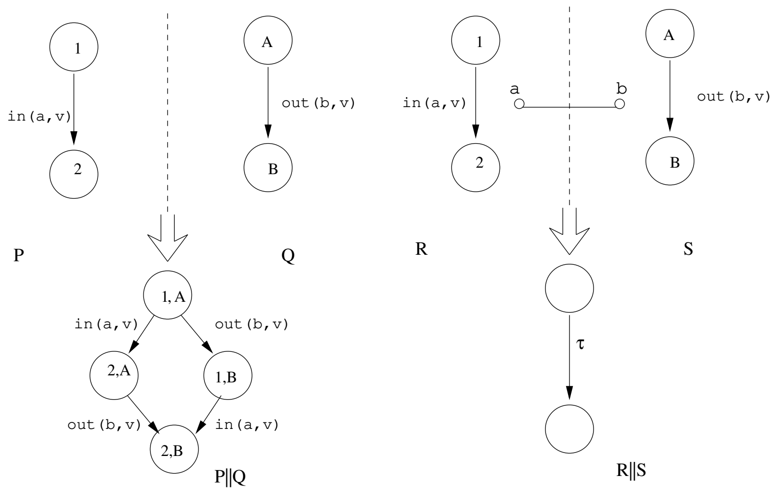

Fig. 1. Examples of action interleaving and of handshaking communication

Figure 1 the left hand side of the figure represents two FSM, P and Q. P has two states: 1 (its start state) and 2 while Q has two states: A (its start state) and B. P can do a transition labeled by $in(a, v)$ (ie interact with the environment through input port a) at state 1 and make a transition to state 2. Similarly, Q can make an output transition $out(b, v)$ at port b and go from state A to B. (The dotted line denotes the concurrent(parallel) composition of the two processes denoted by $P||Q$.)

The third obligation for defining a formal framework for distributed systems is to provide a semantics for communication. Existing distributed system representations (eg CCS [17], CSP [6]) typically support a fixed collection of communication primitives. A closely related question is when can two agents in parallel composition communicate with each other? They can communicate when one agent executes an output transition and the other agent executes an input transition and the ports that are being used by this input-output transition pair are connected to each other. The result of the communication is a silent or τ transition at the global level ie at the level of the parallel composition of the systems. The right hand example of 1 shows two processes R and S synchronizing on the data value v by passing it through the connection between input port a and output port b (denoted by a line)

Statecharts [3] use a different notion of communication/synchronization where rather than the binary handshake illustrated above, events are globally transmitted to all agents in the system (Broadcast). But here also the problem is that the language supports only one form of communication natively.

The final requirement — hierarchy —stipulates that systems also be embeddable inside other systems. This may be achieved by equipping the distributed system representation with a compositional operational semantics, which in effect allows system descriptions to be ”translated” into FSMs by first converting system components into FSMs and then combining these according teo the semantics. This enables us to build systems in a bottom-up structured fashion.

3 Architectural Interaction Diagrams

We use Architectural Interaction Diagrams(AIDs) for specifying systems in this paper. The base formalism for AIDs are IOLTSs (Input-Output Labeled Transition Systems), which are FSMs, consisting of states, transitions, a transition relation, a start state and a set of ports (the set being called an *interface*). A AID agent may have *output* transitions (writing data to a port),*input* transitions (reading data from a port) and composite transitions called *remote procedure calls* which consist of a single transition that containing an output and input action in sequence. This is analogous to a traditional remote procedure call in a programming language, with the output part signifying the supplying of actual parameters by the AID agent and the input part denoting the return value supplied back to the AID agent. A remote procedure call transition differs from an output and input transition in sequence because the former must occur atomically.

An AID agent can take one of two forms: either it can be an IOLTS or it maybe a *network* containing other AID agents embedded in interfaces and connected together in a communication topology as shown in Figure 1. The entities that actually perform the mechanism of communication and synchroinzation are called buses which like AIDs are also provided ports. The ports of a AID component and a bus are connected by *links*. It is also possible to export ports on an interface to an embedee interface through *gates*

The AID 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 basic requirement is that the modeling formalism can be converted to IOLTSs ie for each input language there has to be translation to an IOLTS.

We do not intend to provide a full semantic description of AIDs but the interested reader is requested to refer to [16]. What we do provide is the intuition behind

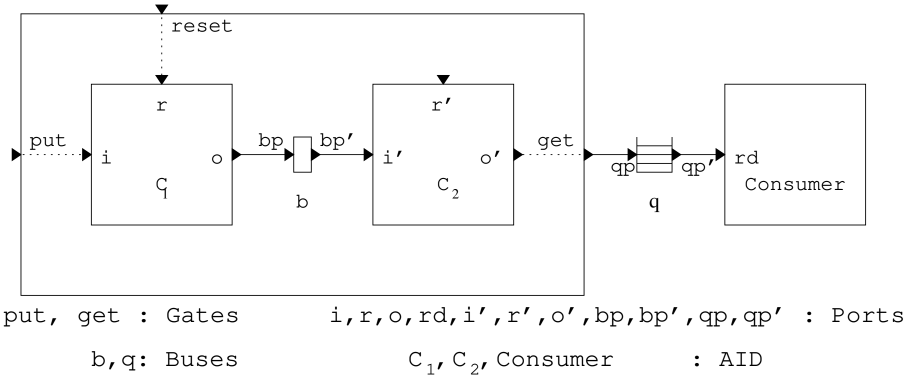

Fig. 2. A nested AID.

buses ie the communication abstraction mechanism of AIDs.

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.

In particular, we wish to view buses as "devices" that combine transitions of subsystems connected to the bus into system-level transitions, according to the synchronization discipline the bus is intended to capture. This is where AIDs differ from conventional approaches . Normally the combining of subsystem transitions to form system-level transitions is done through the || operator and the native handshaking discipline that is "hard-coded" into the semantics of the language. What we want however is to have a more general mechanism by which it would be possible for the user to define her own systems of communication and this newly created communication discipline can then be plugged seamlessly into the native semantics of the language. Buses are the means by which this goal is achieved.

Definition 3.1 A *bus* is a tuple of form $\langle I, B, T, b_0 \rangle$, where I is an interface ie a pair of set of ports (the first set representing write and the second set the read ports), B is a set of bus states, T is a transition relation and and $b_0 \in B$ is the start state

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

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

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.

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.

In order to provide bus transitions, we have two obligations. The first is to define a transition predicate T_P involving free variables WV, R, RV and W with the property that $b \xrightarrow[WV\ R]{W\ RV} b'$ holds exactly when $T_P(WV, R, W, RV)$ is true. The second is to show how the target of the transition ie b' is related to b.

An Example. The Graphical Calculus of Communicating Systems (GCCS) [2] supports synchronous message passing. This form of communication is common in other process algebras like CCS and CSP as well, and we show how it may be encoded as a bus. Buses in GCCS 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 = a tuple consisting of two finite set of ports (read and write).
- $B = \{b\}$ consists of a single state, with $b_0 = b$.
- T contains all transitions for which the following is true: $\exists \langle w, v \rangle \in WV.\ r \in R.W = \{w\} \wedge RV = \{\langle r, v \rangle\}$. Since the bus does not need to store the data and merely needs to pass it on, there is only a single state in the bus. So the target of a transition is always b.

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.

4 The Event Heap

The rest of this paper is devoted to showing how AIDs may be used to capture a much more sophisticated interaction mechanism: an Event Heap. The remainder of this section discusses the Event Heap in more detail, while subsequent ones show how Event Heaps may be captured as buses in AIDs.

An interactive workspace is a localized ubiquitous computing environment where people come together for collaborative activities. Any ubiquitous computing environment needs to coordinate the interactions of applications running on a diversity of mobile and embedded devices. A tuple-space [12] discipline is the most common coordination and communication mechanism in such environments [7,14]. In this section we consider a particular implementation of the general tuple-spaced model of communication, the Event Heap coordination infrastructure [9], a component of IROS, the operating system that runs the i-Room or "Intelligent Room" [10] at Stanford University. The Event Heap extends the standard tuple-space model by providing support to registrations, snooping and support to both blocking as well as nonblocking communication.

The IROS, or i-Room OS, provides a rich set of facilities to applications built on top of it to share data and control. The sharing and coordination of data between applications is done through the Event Heap component of the IROS. The principal aims of the Event Heap is to make applications, that were not apriori designed to work together, interoperate in a dynamic, heterogeneous, ubiquitous environment.

In the tuplespace model, all participants coordinate through a commonly accessible tuplespace. Tuples, which are a collection of ordered type-value pairs, may be posted on the space, or read from the space in either a destructive or non-destructive manner. The tuple to be retrieved is chosen by a template tuple specified by the retrieving application. A client application which is interested in a particular type of event sends templates to the tuple-space and then if the template "matches" any event on the tuplespace, then that event is retrieved. While the application waits for matching event(s) it may choose either to block (ie the thread that initiated the interaction remains suspended till a matching event is found) or to not block (ie if no matching event is found it will return with a suitable message).

The Event Heap extends this basic functionality of tuplespaces by providing facilities to automatically retrieve events that have not been posted but are still of interest to the client. This it does by providing the facility of registration. A client can register for tuples of a particular type by inserting its tuple in a registration tuple and whenever a matching tuple is posted by a server, the tuple will be automatically sent to the client as well as other clients that are registered for it.

Thus the Event Heap provides for anonymous communication as there is no need to explicitly rendezvous applications. As long as two applications understand the same event types they will automatically coordinate with each other. The indirect interaction mechanism of Event Heap discourages strong dependency coupling among applications leading to better interoperability and fault tolerance.

5 Modeling The Event Heap With AIDs

In this section, we show how AIDs may be used to encode the client-view of the Event Heap (ie the view any application using the Event Heap gets of it). For reasons of space, in this paper we show encodings of only some of the functionalities of the Event Heap; a AIDs description of the full Event Heap can be easily provided by considering minor modifications to this model.

Our main aim is to define the Event Heap as a bus. As mentioned before, a bus is the abstraction of communication in AIDs. If we are able to define a bus for Event Heap we can then use this Event Heap bus as a primitive, then plug in different applications into the Event Heap framework and use the entire system for simulation and modeling. The Event Heap coordination and communication framewor will then become an atomic, native mode of communication like synchronous handshake is for standard state-machine based approaches.

In conventional approaches, the Event Heap introduced four components modules whose state-spaces and transitions got tagged onto the application or Event Heap client models. In AIDs, the Event Heap's contribution to the state-space of the client applications is simply the snapshot of its constituent data structures (the event table and the registration table). This is much more efficient as the communication "behaviors" are not getting explicitly encoded as states and transitions as they would have done in conventional approaches . In addition, each communication using Event Heap is guaranteed to be one transition in the global state-space, rather than a sequence of transitions and states. Also AIDs specifications releases the designer of having to explicitly interleave transitions, the responsibility is discharged by the language itself. Consequently, different client modules using the Event Heap can be plugged into the model without having to recode the Event Heap model everytime. While AIDs overcomes the limitations of standard approaches, it maintains compositionality and all the other benefits that conventional state-machine based approaches provide us.

In AIDs the ports on a component (eg an application using the Event Heap) are connected to ports on the bus (eg the Event Heap) by *links*. Links are one-to-one correspondences between component and bus ports. So given a port on the bus, we can find the port and the component it is connected to by following the link in a backward direction. This is what is known as a *pre-image*. In other words, *pre-image* is a function that takes in a bus port and returns the corresponding port on the component obtained by following the link backwards. We define binary relation $samecomp$ on ports so that two ports related by $samecomp$ belong to the same component. That is, for any two ports p_1 and p_2, $samecomp(p_1, p_2)$ holds iff p_1 and p_2 both belong to the interface of the same component in an AIDs network. As examples, in Figure 2, pre-image of port bp is port o of AID component C_1 while r and i both belonging to C_1 are related by the $samecomp$ relation.

In order to provide a definition of the Event Heap bus, we need to discharge three obligations. We have to provide a set of ports for the Event Heap and this set of ports will constitute the interface of the Event Heap. There may be any number of ports in this set signifying that the Event Heap semantics does not impose any restrictions on the components that may use it. The other two obligations are defining the data structure for the bus and providing bus rules.

Data Structures. Our obligation here is to define the data representation inside the bus as well as to define certain accessor functions that will be used by bus rules to manipulate the data structures. We start by providing some elementary types: *PortType*, *EventType* and *TemplateType*. Each of these types may be thought of as structures with fields that may be used for comparison with each other. We do not go into the details of the type structure because that is not the focus of our study. The interested reader may refer to the original (textual) specification of the Event Heap [8]. Ports are of *PortType*, events are of *EventType*, templates are of *TemplateType*. The structure of *Porttype* can disambiguate between multiple ports of the same name but belonging to different components. We also define an atomic boolean operation on types called $match$ which takes an *EventType* and a *TemplateType* and determines if they match by a comparison of certain fields.[Again we do not consider the mechanism of how this is done] We derive the following types from the above elementary types.

- Each data packet sent to the EventHeap contains,among other things, a payload which can either be a Template or an Event. Thus we define *PacketType* which encapsulates a *EventType* or an *TemplateType*.
- *RegType* is a tuple of the form $(TemplateType, PortType)$ where the first element and second element can be extracted by the functions $frst(t)$ and $sec(t)$ where t is a an element of *RegType*.It should be noted that there is a distinguished value of type *TemplateType* called $*$ which denotes the "most general template" ie all events would match this template.
- *EventTableType* and *RegTableType* are lists of types *EventType* and *RegType* respectively. A list l is a sequence of values concatenated by the sequence concatenation operator . and symbolically denoted by V^* where V is the type of the value and its length is denoted by $\mid l \mid$. That is *EventTableType* is a list of type *EventType**, and *RegTableType* is a list of type *RegType**

We now define the state set B of the bus Event Heap as follows.

$$B_{EH} = \{(l, l') \mid l \in \textit{EventTableType}, l' \in \textit{RegTableType}\}$$

Each state of the bus is a snapshot of two data structures: the *Event Table* or *ET* and the *Registration Table* or *RT* respectively. Now we define the following operations on these data structures.

- $Event(v)$, $Template(v)$, $Regport(v)$ are functions that operate on a data value of type $PacketType$ and extract that part of the packet .that contains respectively the event, template and the port that the client registers to receive a particular type of event. Recall that when a client registers, it sends a template to the Event Heap and tells it to send all events that match the template to a particular port on its interface. $Regport(v)$ returns that port at which the client wants to listen to.

- $PUT_{ET}(v, l) = l.Event(v)$ if $| \ l \ | \leq$ MAXLENGTH and is undefined otherwise. The PUT_{ET} function takes in a value of type *PacketType*, extracts the event portion of it and inserts it into the *Event Table*.
- $PUT_{RT}(v, l') = l'.e$ where $e = (Template(v), RegPort(v))$ if $| \ l \ | \leq$ MAXLENGTH and is undefined otherwise. Here the PUT_{RT} function takes in a value of type *PacketType*, extracts the template and the registered port, makes it into a tuple of type *RegType* and inserts it into the *Registration Table*. Sometimes the packet may not contain any particular template but a distinguished character $*$ which tells the Event Heap that the client who sent the packet wants to register for all events to be posted on the Event Heap. In that case, $Template(v)$ = $*$
- $MATCH_{ET}(v) = e$ if $\exists e \in ET.\ match(Template(v), e)$ and is $NULL$ otherwise. Here the $MATCH_{ET}$ extracts the template from a packet, and checks if there is a corresponding match in the Event Table. In general, there may be multiple matches in which case e may be a set of events rather than a single event. But here, to simplify the discussion and notation, we assume that e is a single element. This can however be easily extended to the more general setting.
- $INREG(v) = \{r \mid r \in PortType.\ \exists t \in RT.\ match(frst(t), event(v))! = NULL \wedge r = sec(t)\} \bigcup \{r \mid r \in PortType.\ \exists t \in RT.\ frst(t) = * \wedge r = sec(t)\}$. $INREG$ extracts the event from the packet, checks to see if there is any template in the registration table that would match the event. If there is, then it returns the set of all ports that are interested in receiving the event. $INREG$ also returns the ports present in those registrations that have registered for all events ie whose template is the most general template $*$.

Bus Rules. We now provide the bus rules. Before providing the bus rules, we provide another function called $PacketName$ which operates on an packet and extracts its name.(eg GetEventNonBlocking, GetEventBlocking , PutEvent, RegisterForEvents, RegisterForAllEvents) This function enabled the bus to know which of the rules it is going to apply for a particular interaction. The general structure of a bus transition is: $b \xrightarrow[WV\ R]{W\ RV} b'$. In order to use the Event Heap bus, a client has to execute a "function-call" transition. This can be thought of as a model of an Event Heap API call whereas the write port contains the parameters passed to the API and the read port contains the return value. Now, each of the communication primitive supported by Event Heap can be encoded in this general infrastructure.

As mentioned before, defining bus transitions obliges the designer to specify a transition predicate T_P and the relation between the "before-transition" and "after-transition" state of the Event Heap.

NonBlocking Get. The predicate T_P is as follows:

$\exists \langle w, v\rangle \in WV.\ \exists r \in R.\ samecomp(preimage(w), preimage(r)) \wedge$
$PacketName(v) = GetEventNonBlocking \wedge$
$W = \{w\} \wedge RV = \{\langle r, e\rangle \mid e = MATCH_{ET}(v)\}$

Intuitively, this means that if there exists a port that wishes to write to the bus, and a port that wishes to read from the bus and both these belong to the same component(ie a function call executed by the component) and the packet being considered is a nonblocking get packet, then the transition that wishes to write using the write port is enabled, and the read transition, through the read port, gets the requested event; the requested event being the event in the Event Table that matched the template provided. If no match is made then a null is returned.

It should be recalled that a state of an Event Heap is a tuple consisting of its *Event Table* and *Registration Table* ie the snapshot of an EventHeap bus is the snapshot of these two tables. Since retrieving a tuple modifies neither of these two structures, the state of the Event Heap bus before and after the transition remains same. So formally $b' = b$.

Blocking Get. The predicate T_P is as follows:

$$\exists\langle w, v\rangle \in WV.\ \exists r \in R.\ samecomp(preimage(w), preimage(r)) \wedge$$
$$PacketName(v) = GetEventBlocking \wedge ((e = MATCH_{ET}(v)! = NULL) \wedge$$
$$W = \{w\} \wedge RV = \{\langle r, e\rangle\}$$

This represents the blocking version of the previous "get" event. Note that while in the last rule, the write transition is allowed to fire immediately, here it is delayed till there is actually a match between the posted template and an event in the Event Table. This is according to the semantics of blocking ie the client wishing to get an event is blocked till a matching event is obtained. Like in the case of nonblocking get, the state of the Event Heap bus does not change.

Register. The predicate T_P is as follows:

$$\exists\langle w, v\rangle \in WV.\ \exists r \in R.\ samecomp(preimage(w), preimage(r)) \wedge$$
$$PacketName(v) = RegisterForEvents \wedge$$
$$W = \{w\} \wedge RV = \{\langle r, ack\rangle\}$$

Intuitively, this means that if there exists a port that wishes to write to the bus, and a port that wishes to read from the bus and both these belong to the same component and the event being considered is a register event, then the transition that wishes to write using the write port is enabled, and the read transition, through the read port, gets an "ack" ie an acknowledgment that there has been a successful registration. At the same time, the state of the Event Heap changes as the registered template is inserted into the registration table. So for this rule, if $b = (ET, RT)$ then $b' = (ET, PUT_{RT}(v, RT))$ ie the *Event Table* remains unchanged before and after the transition but the *Registration Table* gets changed by the insertion of v.

PutEvent. The predicate T_P is as follows:

$$\exists\langle w, v\rangle \in WV.\ \exists r \in R.\ samecomp(preimage(w), preimage(r)) \wedge$$
$$PacketName(v) = PutEvent \wedge$$
$$W = \{w\} \wedge RV = \{\langle r_1, event(v)\rangle \mid r_1 \in INREG(v)\} \cup \{\langle r, ack\rangle\}$$

The meaning of the above expression is that if there exists a port that wishes to write to the bus, and a port r that wishes to read from the bus and both these belong to the same component and the event being considered is a put event, then the transition that wishes to write using the write port is enabled, and the read transition, through the read port, gets an "ack" on the r port ie an acknowledgment that there has been a successful put. In addition all those read ports that have expressed interest in the event (either specifically by template or generally for all events) by having registrations in the registration table are sent a copy of the event. At the same time, the state of the Event Heap changes as the event is inserted into the *Event Table*.

So for this rule, if $b = (ET, RT)$ then $b = (PUT_{ET}(v, ET), RT)$ ie the *Registration Table* remains unchanged before and after the transition but the *Event Table* gets changed by the insertion of v.

Event Heap Bus. The Event Heap Bus is defined as $M_{EH} = \langle I, B_{EH}, T, b_0 \rangle$:

- I = tuple consisting of a finite set of write ports and a finite set of read ports.
- B_{EH} is the set of bus states as defined above.
- T consists of all the transition rules defined above.
- $b_0 = (nil, nil)$ where nil denotes the empty list.

Components representing Event Heap client applications may be plugged into Event Heap buses and use the primitives provided natively by it. AIDs imposes minimal structure on the components themselves and only expects them to have an underlying IOLTS semantics. As a result, applications can be modeled in the designer's favorite modeling language and plugged into the AIDs framework.

An Example. We give a simple example of Event Heap clients using the Event Heap Bus. These clients can be models of any distributed application like a synchronized Powerpoint presenter or a Multibrowser client-server [11]. Let us assume that an Event Heap client is interested in a particular type of event which can be posted by any other Event Heap client of the i-Room. In order to do that, the interested client executes a remote procedure call transition by which it retrieves these specific events. A remote procedure call transition may be notationally represented as $M \xrightarrow{w!v_1;r?v_2} M'$, where w is the writeport where the parameter v_1 is provided, the r is the readport where the return value v_2 is obtained and the ; denotes the fact that this is a single atomic transition which binds together the ports w and r to be part of the same transition. These same type of remote procedure call transitions are used by application servers to post their events.

Figure 3 above shows such a scenario. There are two Event Heap clients, one of which serves as the application client and the other as application server. Both the client as well as the server use the Event Heap to communicate and coordinate. Ports are shown as triangles; the diamond is actually a write and a read port bound together by a remote procedure call transition.

The server posts all its events to the Event Heap. It creates a packet v_1 of

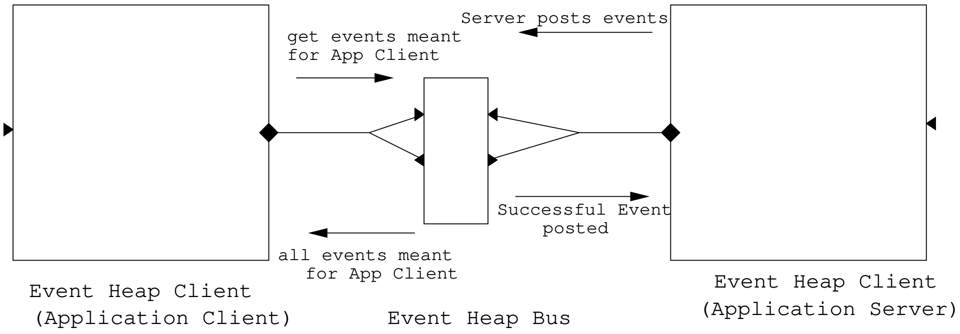

Fig. 3. A simple example of the way the Event Heap Bus is used.

type *PacketType* for which $PacketName(v_1) = PutEvent$ and whose payload contains the event. This packet is then emitted on the write port participating in the remote procedure call transition. Following the link, this packet reaches the bus. The bus then checks to see if there are any registrations for this type of event and sends the event to the read ports which are registered for that particular type of event. In addition to that, the Event Heap sends an acknowledgement back to the server, where it is received by the read port participating in the remote procedure call. The Event Heap also updates the event table with the posted event so that the event is available for any clients who may query for it subsequently.

The client wishes to do a nonblocking check on events it is interested in. For the purpose of our example, we assume that such an event has already been posted to the Event Heap by the server. To retrieve these events, the client creates a packet v_1 of type *PacketType* for which $PacketName(v_1) = GetEventNonBlocking$ and $Template(v_1)$ which says "get the event I am interested in". This packet is then emitted on w and following a link it reaches the Event Heap bus. The Event Heap sees a packet with its name $GetEventNonBlocking$ and on seeing it the corresponding bus transition gets enabled which then, as shown in the first bus rule, extracts the template part of the sent packet, checks for a match in its event table and packs off the result in v_2 which in turn is received at port r by the client.

It should be noted that we do not fix a formalism for the application models using the Event Heap. The user is free to use whatever formalism she chooses, so long as it has an IOLTS-based semantics.

6 Future Work and Conclusions

The aim of this paper has been to take a representative middleware system, the Event Heap, and show how it can be defined in the AIDs framework. Other middleware systems would reasonably be expected to support the same set of features as the Event Heap and differ in implementation only. So our study makes a case for using AIDs for modeling middleware-based systems.

Future work entails incorporating AIDs into the Concurrency Workbench framework and providing designers with the power to apply sophisticated analysis routines on AIDs models. We also seek to explore ways to extend AIDs by providing

v $_1$ of