Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2008) 3–17
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Executable Specifications for Real-Time Distributed Systems

Arnab Ray [1]

Fraunhofer USA Center for Experimental Software Engineering
University of Maryland at College Park
4321 Hartwick Road Suite 500
College Park MD 20742-3290

Rance Cleaveland [2]

Department of Computer Science
University of Maryland at College Park
AV Williams Building
College Park MD 20742

Abstract

One of the challenges in designing distributed, embedded systems is the paucity of formal, executable specification notations that provide support for both real-time and asynchronous communication. This paper describes a timed architecture design language (Timed Architecture Interaction Diagrams or TAID) that, by virtue of its formal, executable semantics, combines the benefits of synchronous specification notations with the advantages of traditional architecture description languages. In addition, TAID provides support for a variety of temporal inter-process communication (IPC) primitives as a native feature of the language, so that the encapsulated communication behavior (captured by real-time "buses" in TAID) may be re-used across designs and serve as specifications for more detailed model implementations.

Keywords: Software Architecture, Real-time, Simulations, Formal Methods, Distributed Systems.

1 Introduction

Software architectures have emerged as important artifacts of the software design process, as they support better system comprehension [12], pre-implementation analysis [16], identification of units of reuse [17] and product-line engineering [2], among other development activities. The research community has developed several formalisms for expressing architectural designs: WRIGHT [13], TRACTA [4],

[1] Email: `arnabray@fc-md.umd.edu`

[2] Email: `rance@cs.umd.edu`

doi:10.1016/j.entcs.2008.05.007

AADL [6], Rapide [11], and AID [15] to name a few. These notations seek to provide a (semi-)formal modeling framework wrapped inside an intuitive and expressive design language. Of these AID (Architectural Interaction Diagrams) distinguishes itself by virtue of its ability to define different inter-process communications (IPC)s as native features of the language, thus facilitating concise and re-usable system specifications.

Most software architecture notations describe system behavior as a causal sequence of events (e.g *A causes B*). This makes them insufficiently expressive when it comes to describing real-time embedded systems, where precise temporal constraints between actions (e.g. *A causes B within 5 seconds of A*) need to be encoded. Notations like AADL do include real time, but do not have a formal semantics. In contrast, foundational synchronous notations such as timed process algebras [5] provide a rich formal framework for describing real-time systems, but these theories are often considered to be too abstract/high-level to be used for realistic system specifications.

This paper demonstrates how, by using ideas from timed process algebras, we may add a notion of discrete time to a non-timed architecture specification language (AID), resulting in Timed Architectural Interaction Diagrams (TAID). TAID is an executable notation that combines the theoretical rigor of timed process algebra with the user-friendliness of an architecture description language. In TAID, communication is defined through semantic devices called *buses*, which may be re-used across designs and serve as an abstract specification for a more detailed model implementation in notations like Simulink®/Stateflow®.

The utility of TAID specifications is that they provide a unified formalism for representing the entire system (both components as well as connectors), thus allowing embedded software engineers the flexibility of performing system simulations on a desktop computer. This leads to large savings of time and money that traditional embedded-system design processes incur, with their emphasis on prototyping, networking testbeds and hardware-in-the-loop testing as virtually the only strategy for design verification and validation.

The paper is arranged as follows. Section 2 outlines the basic concepts of the non-timed architecture design language, Architectural Interaction Diagrams (AID) which this paper extends to create a timed design language: Timed Architectural Interaction Diagrams (TAID) (Section 3). We then illustrate out formalism with two examples of timed inter-process communication. Section 5 details related work, and Section 6 concludes the paper.

2 Background: Architectural Interaction Diagrams

This section introduces some of the different elements that constitute a AID architecture. From Figure 1, one may identify the following concepts: 1) AIDs describing subsystems/components; 2) interfaces, containing read and write ports (each port is a conduit point for data for the subsystem surrounding the interface); 3) connections between ports in a subsystem and ports in an interface (cf. the dotted lines

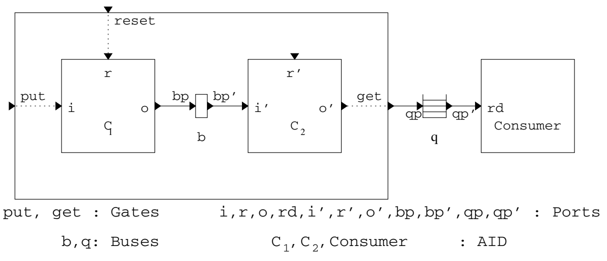

Fig. 1. A sample architecture

in Figure 1) called gates, which enable one to selectively expose ports in an inner AID; 4) buses, the "connectors" through which subsystems communicate with each other; 5) links from interface ports to buses.

The execution semantics of AID are formalized in terms of a transition relation describing system-level execution steps. At the lowest level, an AID component is a state machine that comprises states and transitions. It can perform write transitions (i.e. output a value to a "write" port), read transitions (ie read in a value from a "read" port), or an internal transition.

In the remainder of the section, we provide formal definitions for some of the concepts on which TAID is based on. Let us first define formally what ports are.

- $\mathbb{W}$ 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. 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.

2.1 I/O Labeled Transition Systems

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

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 $T = T_{write} \cup T_{read} \cup T_{internal}$ is the transition relation such that:

(i) $T_{write} \subseteq Q \times \mathbb{O} \times Q$,

(ii) $T_{read} \subseteq Q \times \mathbb{I} \times (\mathbb{V} \longrightarrow Q)$,

(iii) $T_{internal} \subseteq Q \times Q$.

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

An IOLTS P also has an interface $I(P) \subseteq \mathbb{W} \cup \mathbb{R}$ containing the write and read ports used by transitions in P. We also write $Q(P)$ for the set of states for IOLTS P and $q_0(P)$ fot the start state of P.

2.2 Buses

Buses are the most critical elements of the AID paradigm. Mathematically, they can be seen as transducers that convert transitions of incident components (i.e. components that are connected to them) to system-level transitions. As such, they have two responsibilities: the transfer of data between senders and receivers, and the synchronization of sender/receiver transitions.

Formally, a bus in AID has form $\langle I, B, b_{init}, T \rangle$, where I is the interface, B is the set of of bus states, b_{init} is the initial bus state, and the transition relation T contains a single kind of transition – communication — which represents an instantaneous transfer of data among incident components. A communication transition of a bus M is written as:

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

The way to read this transition is as follows: "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.

The sets WV, R, W and RV deserve further comment. WV contains pairs of the form $\langle w, v \rangle$, where w is a write port on the bus and v is a value. Intuitively $\langle w, v \rangle \in WV$ if there is a writer connected to the bus on its port w that wishes to write value v to it. Similarly, $r \in R$ means that there is a reader connected to the bus on its port r that is interested in reading. The bus then, out of these enabled transitions, chooses writers whose port values are stored in W and readers as indicated by RV where $\langle r, v\rangle \in RV$ if reader connected to port r gets value v.

Like we had in the definition of IOLTS, $B(N)$ returns the set of bus states for a bus N and $b_0(N)$ the start state of N.

2.3 Architecture Interaction Diagrams

An *architecture interaction diagram (AID)* is either:

(i) an IOLTS P, with an interface $I(P)$; or

(ii) a network $N = \langle \bar{C}, \bar{M}, L\rangle$, where:
   (a) $\bar{C} = \langle C_1, \ldots, C_n\rangle$ 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 (the formal definition of gate may be found in [15]);
   (b) $\bar{M} = \langle M_1, \ldots M_k\rangle$ is a tuple of buses; and
   (c) L is the set of links. Each link connects a component port with a port on a bus. Interested readers may refer to the formal definition of links in [15]. It should be noted that for the purpose of the paper, the basic intuition behind links and gates is all that is necessary.

Intuitively, a network contains a list of components, each containing a subsystem description, an interface, and a gate definition 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 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.

Mathematically, we define the semantics in the Structural Operational Semantic (SOS) style. 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 the semantics associates with N an IOLTS $\langle Q_N, T_N, q_N\rangle$ describing the operational behavior of N. 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.

Let $N = \langle\langle C_1, \ldots, C_n\rangle, \langle M_1, \ldots, M_k\rangle, L\rangle$ be a network AID. Then:

(i) $Q_N = C_N \times M_N$, where:

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

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

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.

T_N is defined by providing SOS rules that allow us to deduce the set of transitions of N from the transitions of its constituent components and buses as obtained from the structure of N. The SOS rules for AID is given in [15]

3 Timed Architectural Interaction Diagrams

In this section, we define the syntax and the semantics of TAID by augmenting the syntactic and semantic framework used for defining AID. Using concepts from timed process algebra [5], TAID can be "layered" on top of the original language without modifying the semantics of the original theory. In other words, the additional semantics that is needed to encode time can be seamlessly superposed on the original theoretical framework. In this paper, we provide the semantics for only that *incremental* part that provides support to time. For a formal definition of the terms used and the semantics for the untimed part of TAID, the interested reader is invited to consult [15].

3.1 Timed Input-Output Labelled Systems

In TAID, we extend the original definition of IOLTSs to include time transitions (we call these T-IOLTS). A time transition may be thought of as a "clock tick" representing the passage of time.

In timed process algebra time transitions are typically required to satisfy two conditions.

Maximal progress. Time transitions are disabled when internal transitions are possible.

Time determinacy. At most one time transition is possible in any state.

The reason for these assumptions is to separate the modeling of the passage of time from the modeling of system interaction. Maximal progress guarantees that an action must occur as soon as all participants are ready to do it. In other words, enabled actions may not be delayed for even a single clock tick. Time determinacy ensures that the only ambiguity about the state a system can be in is due to the actions it performs, not just the passage of time. A fuller discussion of these issues may be found in [1].

Mathematically, the transition relation of a T-IOLTS P has a component $T_{tick} \subseteq Q \times Q$, where Q is the set of states of P. T_{tick} is also required to satisfy two conditions that are given below. In order to define these, we introduce the following notations. We write $T_{tick}(P)$ for the clock-transition relation of P and $q \xrightarrow{1}_P q'$ when $\langle q, q' \rangle \in T_{tick}(P)$. When P is evident from context, we write simply $q \xrightarrow{1} q'$. We also use $q \xrightarrow{1}$ if there is a q' such that $q \xrightarrow{1} q'$ and $q \not\xrightarrow{1}$ when this is not the case. The notation $q \not\xrightarrow{\tau}$ is used similarly, and when this holds of state q we refer to q as stable.

We can now formulate the properties that $T_{tick}(P)$ must satisfy for T-IOLTS P. Recall that $Q(P)$ is the set of states in P.

Maximal progress. $\forall q \in Q(P).\ q \xrightarrow{1}$ implies $q \not\xrightarrow{\tau}$

Time enabledness. Time is always enabled in stable states.
$\forall q \in Q(P).\ q \not\xrightarrow{\tau}$ implies $q \xrightarrow{1}$

Time determinacy. $\forall q \in Q(P).\ q \xrightarrow{1} q'$ and $q \xrightarrow{1} q''$ implies $q' = q''$

3.2 Timed Buses

Recall from the last section that buses were defined by a set of bus states (B), an interface (I), an initial bus state (b_{init}) and only one kind of transition—untimed communication. In TAID, besides communication transitions, buses contain timed transitions of the form $T_{tick} \subseteq B \times B$, where B is the set of bus states. We write timed transitions as

$$b \xrightarrow{1}_M b'$$

and $b \xrightarrow{1}_M$ if there exists a b' such that $b \xrightarrow{1}_M b'$.

States in AID buses do not have τ-transitions, and thus maximal progress is not an issue for T_{tick} relations in buses. We do require time determinacy and time-enabledness, however; each bus state is required to have exactly one time transition. Assuming M is a timed bus and $B(M)$ its set of states, we write this condition as follows.

$$\forall b \in B(M).\ \exists! b' \in B(M).\ b \xrightarrow{1}_M b'$$

3.3 TAID

Given the notions of T-IOLTS and timed bus, we may now formally introduce Timed Architectural Interaction Diagrams (TAIDs). The definition closely follows that of AID given in Section 2.3. Specifically, a TAID is either:

(i) a T-IOLTS P with interface $I(P)$; or

(ii) a network $\langle \boldsymbol{C}, \boldsymbol{M}, L \rangle$, where $\boldsymbol{C}$ is a vector of components, $\boldsymbol{M}$ is a vector of timed buses, and L is a link relation.

The definitions of component, interface, link, etc. do not change, except that components now include TAIDs instead of AIDs, in addition to inteface and gate specifications.

3.4 TAID Semantics

As with AID, the semantics of TAID with associate with each TAID N a T-IOLTS $\langle Q_N, T_N, q_N \rangle$ describing the operational behavior of N. The structure of a TAID state (i.e, Q_N) is defined recursively on the structure of N as done for AID in

Section 2.3. For T_N, we use all the SOS rules for AID [15] and add to it another SOS rule (given below) that defines the timing behavior of N in terms of the timing behavior of its constituent components and buses. More precisely the rule will allow a network to do a clock tick only if all the components and all the buses that constitute the network can do a clock transition, and if there is no enabled communication between a component and a bus inside the network.

SOS rules have 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.

Now let us define some auxiliary notations that will enable us to define the new SOS rule.

- Recall that a bus M includes a timed transition relation $T_{tick} \subseteq B \times B$, where B is the set of states of M, with the property that if $b \in B$ then there is a unique b' such that $\langle b, b' \rangle \in T_{tick}$. We may therefore define function $CT(b, M)$ which, given bus state b, returns the b' such that $\langle b, b' \rangle \in T_{tick}$.
- Let us now define a function $NSAT$ ("NextStateAfterTick") that, given a TAID state and a TAID, outputs the *next* state of the TAID after a clock tick has taken place.

  Mathematically, $NSAT$ is a function $(Q_N \times N) \longrightarrow Q_N$ that is defined as follows.

  If N is a T-IOLTS of form $\langle Q, T, q \rangle$ then

$$NSAT(q, N) = \begin{cases} q' & \text{if } q \xrightarrow{1}_N q' \in T \\ \textit{undefined} & \text{otherwise} \end{cases}$$

  If N is a network of form $\langle \bar{C}, \bar{M}, L \rangle$, where $\boldsymbol{C} = \langle \langle S_1, I_1, G_1 \rangle, \ldots, \langle S_n, I_n, G_N \rangle \rangle$ and $\boldsymbol{M} = \langle M_1, \ldots, M_k \rangle$, and $\boldsymbol{s} = \langle s_1, \ldots, s_n \rangle$ and $\boldsymbol{b} = \langle b_1, \ldots, b_k \rangle$, then

$$NSAT(\langle \bar{s}, \bar{b} \rangle, N) =$$
$$\begin{cases} \langle \langle NSAT(s_1, S_1), \ldots, NSAT(s_n, S_n) \rangle, & \\ \quad \langle CT(b_1, M_1), \ldots, CT(b_k, M_k) \rangle \rangle & \text{if for all } i,\ NSAT(s_i, S_i) \text{ is defined} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

- A state s in TAID N is said to be stable if it cannot perform any τ transitions, i.e. $\mathit{Stable}(s, N)$ iff $s \overset{\tau}{\not\longrightarrow}_N$

The SOS rule may now be given as follows.

$$\frac{Stable(q, N)}{q \xrightarrow{1}_N NSAT(q, N)}$$

This SOS rule states that if a network cannot perform any internal transitions, only then can it take a clock tick. We may now prove the following.

Lemma 3.1 *Let N be a TAID and q be a stable state of N. Then $NSAT(q, N)$ is defined.*

The lemma is useful in proving the following theorems. It also guarantees that the T-IOLTS associated with a TAID N is time-enabled.

Theorem 3.2 *Let N be a TAID. Then the T-IOLTS associated with N satisfies time determinacy.*

Theorem 3.3 *Let N be a TAID. Then the T-IOLTS associated with N satisfies maximal progress.*

The truth of these statements may be inferred from the definition of $NSAT$, the restrictions imposed on T-IOLTSs and timed buses, and the fact that the SOS rule is the only one that can be used to infer clock-tick transitions for TAIDs.

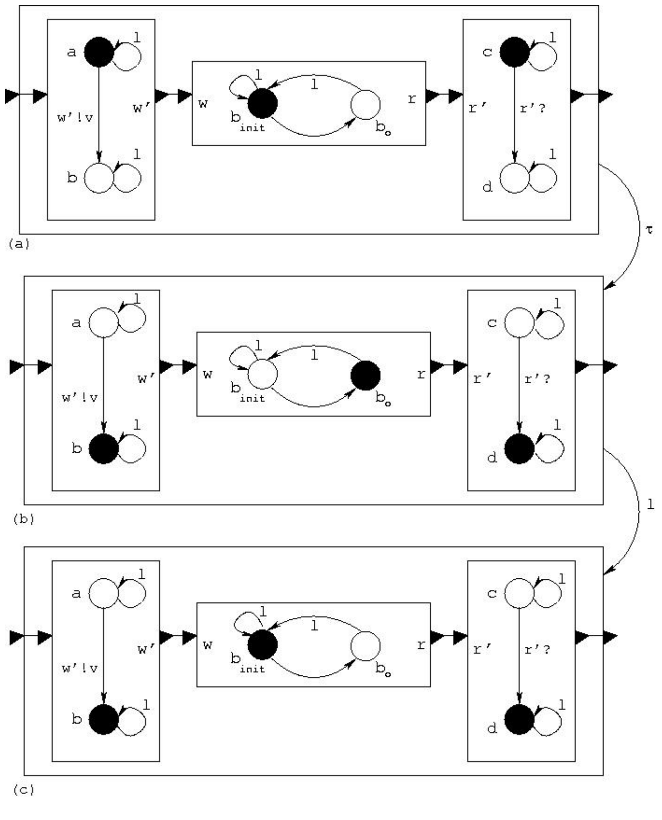

Fig. 2. A TAID execution example

Let us illustrate the concepts introduced in this section with a small example (Figure 2) of a very simple unit-delay handshake where the writer and reader block till communication is finished. Here we have a TAID network consisting of 2 T-IOLTSs and a bus that connects them. One of the T-IOLTS that has the port w' is executing a write transition labeled by $w'!v$ (to be read as: output value v to port w'); this w' port is connected to the w port of the bus via a link. The other T-IOLTS has a port r' (connected via a link to bus port r) where the T-IOLTS in question collects the value from the bus by virtue of the transition labeled by $r'?$. The states colored black represent the states where control presently resides. The bus initially has a transition from its initial state (b_{init}) to state b_0; in this transition (whose labels are omitted), the bus blocks a reader or writer until its partner is ready and then permits a synchronization of the read and write transitions. The bus then delays for one clock tick before cycling back to its initial state.

Initially in configuration (a) in Figure 2, control resides in states a and c of the writing and reading T-IOLTS and in state b_{init} for the bus. A bus communication transition takes place (denoted by the τ transition) wherein the control inside the bus passes to b_0; the reader and writer also change state, and the value v is transmitted to the reader. In (b), the system as a whole takes a clock tick (denoted by 1); the reader and writer remain in the given states, while control in the bus passes from b_0 back to b_{init}. Symbolically, this entire sequence of actions can be represented thus:

$$\langle a, b_{init}, c\rangle \xrightarrow{\tau} \langle b, b_0, d\rangle \xrightarrow{1} \langle b, b_{init}, d\rangle$$

4 Modeling Communication Primitives Using TAID

4.1 Timed Bi-party Handshake

The first example we consider is a bi-party handshake communication that takes δ time units to execute. In this kind of communication, there is no limit on how many subsystems are allowed to use the bus. The bus requires all senders and receivers to block until an exchange of data between one selected writer and selected reader occurs, after which the selected writer and reader are free to continue executing. This transfer of data consumes δ time.

In this semantics, the sequence of events are:

(i) The bus chooses a writer and a reader among the writers and readers who want to use the bus.

(ii) At each instant of time, the bus checks to see if either of the reader or the writer *timed out*, i.e. is no longer interested in the communication it initiated. If that is the case, then the bus transitions to an error state.

(iii) Once the delay is finished and the original writer and reader both are still enabled for the communication, the bus finishes the handshake and releases the writer and reader. Anything else causes the bus to transition to the error state.

Table 1
Synchronous handshake bus $\langle I, B_\delta, b_{init}, T\rangle$ with δ delay

$$I = \langle \mathbb{W}, \mathbb{R} \rangle$$

$$B_\delta = \{b_{init}, b_{err}\} \bigcup \{\langle i, w, r, v\rangle \mid i \in \{0..\delta\}, v \in \mathbb{V}, w \in \mathbb{W}, r \in \mathbb{R}\} \bigcup \{b_{i,w,r,v} \mid i \in \{0..\delta\}\}$$

T is defined as follows.

(i) $b_{init} \xrightarrow[WV\ R]{W\ RV} \langle 0, w, r, v\rangle$ iff $\langle w, v\rangle \in WV \wedge r \in R \wedge W = \emptyset \wedge RV = \emptyset$

(ii) $\langle i, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{i,w,r,v}$ iff $i < \delta \wedge \langle w, v\rangle \in WV \wedge r \in R$

(iii) $b_{i,w,r,v} \xrightarrow{1} \langle i+1, w, r, v\rangle$

(iv) $\langle i, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{err}$ iff $\langle w, v\rangle \notin WV \vee r \notin R$

(v) $\langle \delta, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{init}$ iff $\langle w, v\rangle \in WV, r \in R \wedge W = \{w\} \wedge RV = \{\langle r, v\rangle\}$

(vi) $\langle \delta, w, r, v\rangle \xrightarrow[WV\ R]{W\ RV} b_{err}$ iff $\langle w, v\rangle \notin WV \vee r \notin R$

Note that the semantics of timed handshake is different from the semantics of a FIFO queue with capacity 1 and propagation delay 1 because in a FIFO, a writer writes to the FIFO and is immediately unblocked while here, the writer remains blocked till a reader has read the value.

The responsibility of any bus that implements a timed handshake is twofold: 1) transfer data between a writer and a reader 2) enforce the blocking of the chosen writer and reader for the duration of the handshake. Since it is not possible for any communicating medium to physically block a writer and reader (since they may always execute a time-out transition), the bus should not complete any interaction where at some time during the δ delay, either the writer or the reader timed out. As a result, when a successful handshake is completed, we can then automatically deduce that the writer and the reader were indeed blocked for δ time units.

The bus definition is provided in in Table 1. The first line in the table indicates that the bus's interface has a set $\mathbb{W}$ of write and a set $\mathbb{R}$ of read ports. The second line in the table defines the bus states. There are two distinguished bus states: b_{init} (the initial state) b_{err} (the error state) in addition to 1) a set of general bus states, each of which is a tuple that contains a variable i i.e. the number of clock ticks already elapsed at that state, a variable w that stores the value of the selected write port, a variable v that stores the data value obtained from w and a variable r that contains the value of the read port selected 2) a set of "trap" states of the form $b_{i,w,r,v}$ which are the states at which the bus can perform a clock tick.

Rule 1 states that if there exists at least one writer and a reader, then a writer is chosen along with its value as also a reader and stored in the bus state. The counter representing time elapsed since initiation of communication is set to 0. Rule 2 says that if the chosen writer and reader are present in the set of enabled writers and readers respectively, and the upper limit of the counter (i.e. δ) has not been reached, then the bus transitions to a state where it is allowed to perform a tick (vide Rule

3). If however, the writer or reader has timed out, then the bus (vide Rule 4) transitions to an error state. If when the counter expires, the original writer and reader are still present in the interaction, the handshake is completed and the writer and reader are released (Rule 5). Else a transition to an error state is made (Rule 6).

4.2 Timed Buffered Communication

In a timed buffered communication channel, writers and readers communicate over a FIFO buffer whose capacity is assumed to be N data elements. There exists a pre-determined propagation delay (δ time units) between the tail and the head of the FIFO buffer. In other words, a data element written to the head of the FIFO buffer is read δ time units later at the tail end.

Let us now define the set of bus states and auxiliary get and put functions that enable us to add/remove data elements from the internal data structure of the bus.

- $B_{N,\delta} = 2^{\{0,\ldots,\delta\}\times\mathbb{V}}$ ie $B_{N,\delta}$ is the set of subsets of tuples, the first element being a *time in flight* (TIF) counter (with the initial TIF of any element in the FIFO being equal to δ) and the second element being the actual data value that is stored as a cell in a FIFO where the cardinality of the set is N.
- $$put(\langle t,v\rangle,b) = \begin{cases} b\bigcup\{\langle t,v\rangle\} & \text{if } \neg\exists v'.\langle t,v'\rangle \in b \\ b-\{\langle t,v'\rangle\} & \text{if } \langle t,v'\rangle \in b \\ undefined & \text{if } \mid b \mid = N \end{cases}$$

  The intuition behind this is that when an element is written to the FIFO, the *put* logic checks to see if there is another data value with the same TIF already present in the FIFO. If there is not, then the data is added to the FIFO. If there exists a data value with the same TIF (i.e. the same time-stamp), then a data collision has occurred or in other words, simultaneous writes to the same location at the same time has taken place. In this case, we allow none of the colliding values to enter the FIFO.
- We define a *tick* function that decrements the TIF associated with each element in the FIFO.
  $tick(b,n) = \{\langle t-n,l\rangle \mid \langle t,l\rangle \in b\}$

  Note that the definition of *tick* allows the TIF to become negative. A data value with an associated negative TIF can be interpreted as a value that has propagated from the head to the tail of the FIFO but has still not been read (and thus removed) from the FIFO. If more than one data value with a negative TIF exists, the one with the minimum TIF is the one that is read (since it has been in the FIFO for the longest time).
- Before we define, the *get* function for a FIFO, we need to establish some auxiliary definitions.

Table 2
FIFO bus $\langle I, B_{N,\delta}, \emptyset, T\rangle$, with capacity N and δ delay

$$I = \langle \mathbb{W}, \mathbb{R}\rangle$$
$$B_{N,\delta} = 2^{\{0,\delta\}\times\mathbb{V}}$$

T is defined as follows.

(i) $b \xrightarrow[WV\ R]{W\ RV} b'$ iff $\exists\langle w, v\rangle \in WV.\, W = \{w\} \wedge b' = put(\langle\delta, v\rangle, b)$
or $\exists r \in R, v \in \mathbb{V}.\, get(b) = \langle v, b'\rangle \wedge RV = \{\langle r, v\rangle\}$

(ii) $b' \xrightarrow{1} b''$ iff $b'' = tick(b', 1)$

lnp is a function that takes a set of tuples of the form $\langle t, v\rangle$ and returns a tuple with the lowest non-positive TIF.

$$lnp(T) = \begin{cases} \langle t, v\rangle & \text{if } \langle t, v\rangle \in T,\ t \le 0 \text{ and } \forall\langle t', v'\rangle \in T.t' \ge t \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Now we define the *get* function.

$$get(b) = \begin{cases} \langle v, b - \{lnp(b)\}\rangle & \text{if } lnp(b) = \langle t, v\rangle \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Rule 1 in Table 2 state that as long as the buffer is not full, writers can keep on writing to the bus. All such writes are instantaneous, i.e. time progresses only after all enabled writes are completed. When a data value enters the bus, there is a time-stamp (or more precisely a time in flight) that is attached to it—in this example since propagation delay is assumed to be δ, we attach the value δ to every data value once it is written to the bus. Every time a clock ticks, the "time in flight" is decremented by 1 (Rule 2). Readers who want to read are allowed to do so only if the data element at the read end of the buffer has its "time in flight" less than or equal to 0 which means that it has been in the buffer for at least δ time units.

5 Related Work

The SAE Architecture Analysis & Design Language (AADL) [6] (that grew out of MetaH [8]) is a textual and graphical language supports model-based engineering of embedded real-time systems. However AADL, not having an explicit *execution semantics* for inter-component communication lacks the power to package communication behavior into architectural elements which can then be used as atomic blocks for system construction. All AADL communication takes place implicitly through queued or nonqueued data at ports whereas in TAID, components perform communication via an explicit entity called a bus that encapsulates the specific communication semantics which is not limited to queued/non-queued communication only.

WRIGHT [14] and TRACTA [4] are ADLs that, like TAID, have a formal executable semantics "under the hood". However, they do not provide notions for modeling time and also do not provide for the ability to parameterize communication ie these languages have a single communication primitive that is built into it and that cannot be extended by any means. Ptolemy [10] is a modeling environment that provides support for heterogenous specifications by allowing for encodings of different models of computation. TAID distinguishes itself from Ptolemy by its use of buses as a means for encapsulating different models of computation (synchronous, asynchronous) in a concise manner.

There are several tool implementations eg Artisan Studio [7] that realize UML-RT [3] and SysML [9] constructs but this approach suffers from UML/SysML not having a standardized execution semantics.

For a more full-fledged discussion of different ADLs with respect to AID,interested readers are requested to refer to [15]

6 Conclusions and Future Work

This paper describes a executable, timed specification language, TAID, for describing real-time, distributed systems. TAID is intended for use in the design process for high-integrity embedded systems. Our future work consists of using Simulink/Stateflow as modeling infrastructure for representing and simulating distributed real-time system designs comprising TAIDs whose components are Simulink / Stateflow models. The effort also involves the development of Simulink blocksets as specifications for standard communication platforms. A project on this topic is underway with a major international automotive company.

References

[1] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: real time and discrete time. *Handbook of Process Algebra (J. A. Bergstra and A. Ponse and S. A. Smolka, editors)*, pages 627–684, 2001.

[2] P. Clements and L. Northrop. Software product lines: Practices and patterns. *Boston, MA: Addison-Wesley*, 2002.

[3] Bruce Powel Douglass and David Harel. Real-time UML: Developing Efficient Objects for Embedded Systems. 1997.

[4] D. Giannakopoulou. The TRACTA Approach for Behaviour Analysis of Concurrent Systems. *Department of Computing, Imperial College of Science, Technology and Medicine DoC 95/16*, 1995.

[5] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput*, 117(2):221–239, 1995.

[6] `http://www.aadl.info`.

[7] `http://www.artisansw.com/pdflibrary/Studio6.0.pdf`.

[8] `http://www.htc.honeywell.com/metah/`. Honeywell corp.

[9] http://www.sysml.org/.

[10] Edward Lee. Overview of ptolemy project. *Technical Memorandum*, 6(3):213–249, 2001.

[11] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September.

[12] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[13] R.Allen and D.Garlan. Formalizing architectural connection. *16th International Conference on Software Engineering*, 1994.

[14] R.Allen and D.Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[15] Arnab Ray and Rance Cleaveland. Architectural interaction diagrams: Aids for system modeling. *Proceedings of the International Conference on Software Engineering,(ICSE)*, pages 396–406, 2003.

[16] Mary Shaw. Software Architectures for Shared Information Systems. *D.M. Steier and T.M. Mitchell (Eds.), Mind Matters: A Tribute to Allen Newell. Mahwah, NJ*, pages 219–251, 1996.

[17] Kevin J. Sullivan and John C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. *Proceedings of 18th International Conference on Software Engineering*, pages 220–229, 1996.