Distributed simulation of AADL specifications in a polychronous model of computation

Yue Ma Jean-Pierre Talpin Sandeep K. Shukla Thierry Gautier
INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France
FERMAT LAB, Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, USA
Yue.Ma@irisa.fr Jean-Pierre.Talpin@irisa.fr Shukla@vt.edu Thierry.Gautier@irisa.fr

Abstract

Architecture Analysis and Design Language (AADL) is used to describe the hardware and software architectures of embedded applications at the system level. The implementation of such systems is often distributed across asynchronous communication infrastructures. Such a distributed system is usually composed of locally synchronous processes communicating in a globally asynchronous manner, a GALS system. Yet, in a step-wise refinement based approach, one would prefer to model, simulate and validate such a system in a synchronous programming framework, and then automatically generate its GALS implementation. In this paper, we present a methodology to implement such an approach using the polychronous (multi-clocked synchronous) model of computation of the data-flow synchronous language SIGNAL. We show how to model partially asynchronous application and to generate distributed simulation code starting from system-level AADL specifications.

1 Introduction

Embedded systems are an integral part of safety critical systems in various domains, such as avionics, automotive and telecommunications. For various reasons (performance increase, location of sensors, independency, flexibility, fault tolerance), many real-time applications require code distribution. In some cases the distribution is at the discretion of the designer attempting to achieve improved computational speed. In other situations it is naturally enforced by geographical separation between subsystems. Another practical consideration is current processor capabilities which may not be adequate to complete massive centralized computations in a given time. This is particularly important as more and more computations are becoming necessary for controlling an ever increasing number of features and options. However, distributed systems are hard to design, debug, test and formally verify, as these systems have to conform to many stringent functional and non-functional requirements from multiple contexts. Ensuring all these requirements and features becomes very difficult if the whole system is hand-coded. Thus, a large part of the application code should preferably be generated automatically from a verifiable and analyzable model to make the engineering work faster and easier. This can be achieved through the use of the maximum degree of automaton, especially with respect to verification and distributed code generation. This paper is hence trying to propose a model to automatically generate formal models and distribution code from AADL specification.

Architecture Analysis and Design Language (AADL [1]) is an SAE (Society of Automotive Engineers) standard aimed at the high level design and evaluation of the architecture of embedded systems. AADL can capture the design of a complete application and its key components. Since they allow for a more abstract view of the application than programming languages, they help in identifying the structural components, and eventually in expressing properties of the whole architecture. At the AADL specification level, the system is distributed into a set of execution platforms without necessarily having a physical implementation of the system at hands [2]. Large projects, for example the ASSERT project [3] and the COTRE project [4], rely on AADL to design embedded systems.

Synchronous languages can significantly ease the modeling, programming and validation of embedded systems [5]. However, when the target architecture is a distributed system, implementing a synchronous specification as a synchronous design may be inefficient in terms of both size and performance. A more elaborate implementation style where the basic synchronous paradigm is adapted to distributed architectures by introducing elements of asynchrony is highly desirable. That is why numerous works are devoted to combing synchrony with asynchrony. For
instance, the paradigm of “Globally asynchronous locally synchronous system” (GALS) has been proposed to describe asynchronously communicating synchronous modules [6].

The solution we proposed is based on SIGNAL, a domain-specific, synchronous data-flow, language dedicated to embedded and real-time system design [7]. While being declarative like Scade or Lustre [8], and not imperative like Esterel [9], its multi-clocked model of computation (MoC) stands out by providing the capability to design systems where components own partially related activation clocks. This *polychronous* MoC is called *polychrony*. Polychrony also provides the mathematical foundation to define a notion of behavioral refinement. Behavioral refinement is the ability to model a system from the early stages of its requirement specifications (properties) to the late stages of its synthesis and deployment (functions) by its iterative upgrade with correctness-preserving, automated or manual, program transformations.

**The TopCased Project** TopCased [10] is a large open-source project devoted to the design of critical embedded systems. In the TopCased process, several meta-models are proposed, including those for describing architectures in AADL and those for modeling synchronous components. The POLYCHRONY platform [11], that implements the polychrony model, is part of the TopCased software. It provides models and methods for a rapid, refinement-based, integration and a formal conformance-checking of GALS hardware/software architectures.

We use APEX-ARINC library of the SIGNAL environment (which proves to be suitable and adequate to model embedded architectures in the specific case of Integrated Modular Avionics (IMA [12]) considered in the TopCased project), to transform the AADL architectural components into SIGNAL process. Then, we distribute the component processes to different processor modules for simulation.

![Figure 1. From AADL to SIGNAL model](image)

The paper is organized as follows. Section 2 gives a brief summary of the syntax and semantics of SIGNAL. Section 3 presents an overview of IMA architectures. In Section 4, we present how to translate the AADL components to SIGNAL programs, and how the generated SIGNAL programs are automatically distributed to create a distributed simulation of the system. Section 5 discusses the related work and Section 6 draws our conclusions.

**2 An Overview of the SIGNAL Language**

The data-flow synchronous formalism SIGNAL language dedicated to the design of embedded systems for critical application domains. A SIGNAL program, also called *process*, is a system of equations over *signals* that specifies relations between values and clocks of the *signals*.

SIGNAL handles unbounded series of typed values \((x_t)_{t \in \mathbb{N}}\), called *signals*, denoted as \(x\) and implicitly indexed by the discrete pace of its clock, noted \(\dot{x}\). At a given instant, a *signal* may be present or absent (denoted by \(\bot\)). The clocks \([x]\) (resp. \([\neg x]\)) denotes the instants at which the signals \(x\) is present and holds the value true (resp. false).

\[ e := \dot{x}[x][\neg x] \]  

Two signals are synchronous if they are always present (or absent) at the same instants. They have the same clock (operator \(=\)). In the example below, the signal \(x\) and \(z\) are synchronous: \(x = z\), by contrast, signal \(x\) and \(y\) are not synchronous.

\[
\begin{align*}
  x &: 1 3 6 2 5 7 8 \ldots \\
  y &: 1 2 5 6 9 \ldots \\
  z &: 8 4 2 1 3 6 5 \ldots 
\end{align*}
\]

In Signal, a process (written \(P\) or \(Q\)) consists of the synchronous composition (noted \(P || Q\)) of equations on signals (written \(x = y f z\)). An equation \(x = y f z\) defines the output signal \(x\) by the relation of its input signals \(y\) and \(z\) through the operator \(f\). The process \(P/x\) restricts the lexical scope of the signal \(x\) to the process \(P\).

The abstract syntax of a process \(P\) in SIGNAL is defined in the following:

\[ P, Q ::= x = y f z \mid P \mid Q \mid P/x \]

Signal defines the following primitive equations:
- A delay \(x = y \text{ pre } v\) initially defines \(x\) by the value \(v\) and then by that \(y\) had last time it was evaluated.
- A sampling \(x = y \text{ when } z\) defines \(x\) by \(y\) when \(z\) is true.
- A merge \(x = y \text{ default } z\) defines \(x\) by \(y\) when \(y\) is present and by \(z\) otherwise.

**3 Integrated Modular Avionics**

IMA is the recent architecture proposed for avionics functions in order to reduce the potential risk of massive use of resources[13]. In an IMA architecture, several avionics applications with possibly different criticality can be hosted on a single, shared computer system. It ensures a safe allocation of shared resources so that no fault propagation occurs from one component to another component. This
is achieved by partitioning of the applications with respect to available time and memory resources [12]. The allocation unit that results from this decomposition is the PARTITION.

Mechanisms are provided to prevent a partition from accessing resources of other partition. A processor is allocated to each PARTITION for a fixed time window within a major time frame maintained by the module-level OS. When a PARTITION is activated, its composed PROCESSESes, which represent the executive units, run concurrently to perform the functions associated with the PARTITION. Partition-level OS is responsible for the correct execution of PROCESSES within a PARTITION. Suitable mechanisms and devices are provided for communication and synchronization between PROCESSES (e.g. buffer, event, semaphore) and PARTITIONs (e.g. ports and channels).

APEX-ARINC (Avionics Application Software Standard Interface [14]) is a standard proposed operating system interface for IMA applications. The APEX interface allows IMA applications to access the underlying OS functionalities. The interface includes both services to achieve communications and synchronizations, and services for the management of PROCESSES and PARTITIONs.

The POLYCHRONY design environment includes a library in SIGNAL, modeling both services to achieve communications and synchronizations, and services for the management of PROCESSES and PARTITIONs defined in the ARINC-653 standard [14].

4 Translating AADL specifications into SIGNAL distributed models

In this section, we present a framework to automatically transform AADL descriptions into a polychronous model. We addressed the transformation in two stages. First, we use IMA architecture and library to translate AADL components into SIGNAL processes. Second, we distribute the generated code to different processor modules based on the AADL architecture specifications for simulation. Here we present how the generated codes are distributed for detail.

The AADL components to SIGNAL transformation is presented in short, the reader can reference [16] for more information.

4.1 From AADL components to SIGNAL processes

The purpose of a model in AADL is to describe the execution characteristics of the system. Because such characteristics depend on the hardware executing the software, an AADL model includes the description of both software and hardware [15]. An AADL model is made of a hierarchical assembly of software and hardware component types and implementations. The top-level component is a system.

A system is made of several devices and processors; each processor can run several processes; each process can execute several threads; and each thread can call several subprograms. The leaves in the parse tree of an AADL model are either subprograms or component types.

We present some general rules to translate AADL models into the SIGNAL programming language. We put our translation to work by studying the similarity between the AADL and the APEX-ARINC services. For space limitation, we only present the transition rules for a system component, for more component detailed translations, the readers may refer to [16].

**System** The system is the top-level component of the AADL model, mixing hardware and software components.

```
system ::= (S_TYPE, S_IMPL) and (S_IMPL.compType=S_TYPE)
S_TYPE ::= system ID [Feature] [Flow] [Property] end ID;
S_IMPL ::= system implementation ID [Subcomp] [Connect] [Flow] [Mode] [Property] end ID;
Subcomponent ::= system|data|process|processor|memory|bus|device
```

A system \( S = (C, M, W, F, P, Z) \) specifies the runtime architecture of an operational physical system, where:

- \( C \) is the set of port connections, which are explicit relations between ports that enable the directional exchange of data or event among components, and data/bus access connections, which enable multiple components access to a common data/bus component.
- \( M \) is the mode abstract which is an explicitly defined configuration of contained components, connections, and property value associations.
- \( W \) specifies the flows that enable the detailed description and analysis of an abstract information path through a system.
- \( F \) is the feature that declares the ports.
- \( P \) is the property which provides descriptive information.
- \( Z = (Ss, Da, Ps, Pr, Me, Bs, Dv) \) is the subcomponent set of the system, including system \( Ss \), data \( Da \), process \( Ps \), processor \( Pr \), memory \( Me \), bus \( Bs \) and device \( Dv \).
A system may have one or several processors, and each processor can execute one or several processes, while an ARINC PARTITION cannot be distributed over multiple processors. The processor is allocated to each PARTITION for a fixed time window within a time frame maintained by the core module level OS. So here we translate the system to a top-level SIGNAL process, and separate it into several PARTITIONS according to its owned processors. Here we introduce a notation $T$ to represent the translation of the AADL component $C$ into SIGNAL.

**Rule 1.** $T(S) = \text{module } M$, if $S$ is the top level system (the main system, which may contain sub-systems).

**Rule 2.** $T(S) = \text{process } P$, if $S$ is a subsystem.

**Rule 3.** $T(pr, ps) = \text{PARTITION } R$, where $pr \in Pr$, $ps \subseteq Ps$, and the processes in $ps$ is bound to $pr$, which is restricted in the property $P$.

**Rule 4.** $T(i, p) = i$, $T(o, p) = o$, where $i, p, o, p$ are the input/output port of a process $Ps$, $i, p \in Ps.F$, $o, p \in Ps.F$, and the port is connected to some port of the system $S$, $\exists p1, C(i, p, p) \in C$, $\exists p2, C(o, p, p) \in C$, $i/o$ are the input/output of the PARTITION $R$.

**Rule 5.** $T(da) = \text{type } A$, $T(dv) = \text{process } B$, where $da \in Da$, $dv \in Dv$. Data is translated to a type, and device is translated to a process outside the PARTITIONS.

**Rule 6.** $M = (P, R, A, B, I)$. The translated system process contains all the processes from the subcomponents and the data types (which will be used by the processes). The processes will be model instances in $M$, the instances are specified in $I$.

**Rule 7.** $P = (P, R, B, I)$, the translated system process contains all the processes from the subcomponents, and they will be model instances in it, which are specified in $I$.

For instance, the client system in Figure 2 has two processors, Processor1 is bound to processor1, and the other two processes are bound to processor2. The corresponding SIGNAL model contains two PARTITIONS (Figure 3 left), each is composed of a processor and the bound processes, for instance, Partition1 contains PARTITION_LEVEL_OS1 (translated from Processor1) and CLIENT_PROCESS_i (translated from Process1) as shown in the zoom on Figure 3 right, and the device subcomponent is translated as a process in the model.

**Process** The AADL process component represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside the process. Here we translate the AADL processes executed on the same processor to a PARTITION.

**Thread** A thread is a concurrent schedulable unit of sequential execution through source code. In APEX ARINC, PROCESSES represent the executive unit for an application. They share common resources and execute concurrently within a PARTITION. The thread component can be translated as an ARINC PROCESS, the properties, such as Dispatch_Protocol, Period, can be translated as the PROCESS attributes.

**Subprogram** The subprogram is a callable component that operates on data or provides functions to components that call it. Subprogram components represent elementary pieces of code that processes inputs to produce outputs. The
ARINC block represents elementary pieces of code to be executed without interruption and within a bounded amount of time. The subprogram component can be mapped into a block, the code should be executed without interruption.

**Device** Device components are used to interface the AADL model with its environment. Devices are not translated as the other components, they are modeled outside the PARTITION, the implementation can be provided in some host language, such as C/JAVA.

**Processor** Processor component is an abstraction of hardware and software responsible for executing and scheduling threads. In ARINC services, PROCESSES run concurrently and execute functions associated with the PARTITION in which they are contained. The processor can be translated as the scheduler of the AADL threads which are bounded to the processor, corresponding to the PARTITION_LEVEL_OS in SIGNAL, which is the control for the concurrent execution of PROCESSES within the PARTITION.

### 4.2 Distributed simulation model generation

A distributed system can be much larger and more powerful given the combined capabilities of the distributed components than combinations of stand-alone systems. The purpose of our distribution is, given a centralised program and some distribution specifications, to build the program on different processors. These programs must be able to communicate harmoniously, such that their combined behavior will be functionally equivalent to the behavior of the initial centralised source program.

After the translation step, now we have two elements. The first one is a SIGNAL program, which includes all the computation components and depicts flows of data. The other one is the architecture: in the AADL architecture specification, it is clearly defined how the system should be distributed, for example, which process should be executed and scheduled by which processor, and how the processor will periodically or sporadically schedule the process.

Our goal is to obtain automatic distributed code generation from:

1. the software architecture of the application,
2. a representation of the distributed target architecture,
3. a manual mapping of the software modules onto the hardware components.

To automatically distribute a SIGNAL program, we first map its specified components on the target architecture, add a scheduler for each location, synthesize the clock synchronizations between the partitioned components, then add communications in place of these synchronizations, and finally generate code. We illustrate this methodology by an example showing how the SIGNAL program is obtained from the AADL specification and then distributed using POLYCHRONY.

#### 4.2.1 Map to the target architecture

By analyzing the AADL architecture, the system is composed of, say, two processors (see Figure 5), so we have two PARTITIONS translated from the AADL model. Then we need to distribute the PARTITIONS to different machines/processors. Because each PARTITION is translated from the processes bound to the same processor, no two PARTITIONS share the same processor, so we can assign each PARTITION to a different processor.

![Figure 4. Distributed code generation](image)

![Figure 5. Distribution](image)

Each PARTITION is paced by its own clock. The SIGNAL pragma “RunOn” in the POLYCHRONY environment is used for partitioning information: when a parti-
tioning based on the use of the pragma “RunOn” is applied to an application, the global application is partitioned according to the different values of the pragma so as to obtain sub-models. The tree of clocks (the root of the tree represents the most frequent clock) and the interface of these sub-models may be completed in such a way that they represent endochronous processes [17] (an endochronous process is mostly insensitive to internal and external propagation delays). The program $P$ will be rewritten as $(|P_1|...|P_n|)$ where $n$ is the number of processors. This step is only a syntactic restructuring of the original program. The SIGNAL process $P_i$ is annotated with a pragma “RunOn $i$”. Here is a simple example for distributing two PARTITIONS run on different processors:

```plaintext
process DIST = (? boolean request, H; ! event response;)
pragmas
  target "MPI" RunOn {e1} "1" RunOn {e2} "2"
end pragmas
(|e1::(|(x,response):= CLIENT(request,y,init1,active1)
  |(init1,active1):= SCHEDULE1(H))|
|e2::(|y:= SERVER(x,init2,active2)
  |(init2,active2):= SCHEDULE2(H)))|
where label e1,e2; Message_type x,y;
  event init1,init2;
  integer active1,active2; end;
```

We put the SCHEDULE implementation details off until later refinement stages and focus on its distribution in this step: the two PARTITIONS are assigned to two different labels, each label will run on a different processor.

4.2.2 Scheduler

The purpose of scheduling is to be able to structure the code into pieces of sequential code and a scheduler, aiming at guaranteeing separate compilation and reuse. The scheduler selects enabled tasks for execution according to the scheduling policy. In APEX-ARINC model, the MODULELEVEL_OS is the scheduler for scheduling the PARTITIONS within the same module (See Figure 6). Scheduling partitions is strictly deterministic over time in the ARINC653 standard. Based upon the configuration of partitions within the core module, overall resource requirements/availability and specific partition requirements, a time-based activation schedule is generated that identifies the partition windows allocated to the individual partitions. Each partition is then scheduled according to its respective partition window. The schedule is fixed for the particular configuration of partitions on the processor. Within the PARTITION, there is a PARTITIONLEVEL_OS for scheduling the PROCESSES which is already implemented during the translation phase.

In our transformation rules, each processor runs only one PARTITION, so the scheduler for the PARTITIONS running on one processor is very simple. We can create a simple scheduler reference to the MODULELEVEL_OS in APEX which schedules only one PARTITION. Then the SIGNAL process can be refined like the following example (the processes INIT() and GET_ACTIVE_PARTITION() are left for initializing and activating the partitions, they would be non-trival for several partitions):

```plaintext
process SCHEDULE = (? event H;
  ! event initialize; integer active_partition;
  (!|initialize := INIT(H)|)
  !|active_partition := GET_ACTIVE_PARTITION()
  |b_init := (not initialize)$1 init true
  |active_partition ˆ= when (not b_init)|)|
where boolean b_init; end;
```

After adding the scheduler to the program, now we can make a global compiling to separate the parts for further development. This phase makes explicit all the synchronizations of the application and detects synchronization constraints (the compiler uses clock calculus to statically analyze every program statement, identify the structure of the clocks of variables, and schedule the overall computation. If the collection of the statements as a whole contains clock constraints violation, then synchronization constraints exist). If it is not, the program is made endochronous. The compiling stops if constraints are detected. The distributed program is automatically generated in the POLYCHRONY environment, as below:

**PART 1**

```plaintext
process DIST_1 = (? boolean request, H; Mess_type y;
  ! event response; Mess_type x;)
pragmas RunOn “1” end pragmas (|...|);
```

**PART 2**

```plaintext
process DIST_2 = (? boolean H; Mess_type x; ! Mess_type y;)
pragmas RunOn “2” end pragmas (|...|);
```

The program is separated into two parts, each one is labeled as $XX_j$ (XX represents the name of the original SIGNAL program, “DIST” in the example), and the internal inputs/outputs from the other part are automatically added. In our example, the two parts $DIST_1$ and $DIST_2$ are generated after the global compiling, message $x$ and $y$ are added by the compiler automatically, message $x$ is transferred from
DIST_1 to DIST_2, and DIST_1 will receive message \(y\) from DIST_2.

Compared with the original program, the separated parts perform a rewritten. All the computation properties are preserved, and a scheduler is added for each of the processor.

**4.2.3 Adding communications**

A distributed program consists of a set of processes interacting with the environment and communicating among themselves. Reasons for the communication are to send data or signal to another process, to synchronize with other processes, or to request an action from another process.

A common distributed processing environment is constituted by several parts that are interconnected forming a network and they communicate and coordinate their actions by passing messages. Usually a real-time distributed executive provides services such as time and resource management, allocation and scheduling of processes and inter-processes communication and synchronization. Among these services, one of the most important is the communication support. We emphasize this issue because of the peculiar role of inter-processor communications in distributed memory multi-processors.

In distributed computing, a common assumption is that when a task sends a message to some other task it should not need to know where this task is situated, making the message communication transparent. Some commercial operating systems (e.g. VAX/ELN) provide a distributed kernel, which directly supports this transparency in the message communication. If this property is not available then a Distributed Task Manager (DTM) is usually developed to provide this transparency. The DTM is a layer of software that stands above each operating system on each node. Therefore some form of communication and synchronization mechanism which guarantees that the semantics of synchronous communication will be preserved needs to be added.

There are two basic solutions for communications: shared variables and message passing. Shared variables do not allow synchronization between parallel processes, unless some complex mechanism is built on top of them. Moreover, they make formal verification harder. Message passing in distributed systems can be synchronous or asynchronous. In our implementation, we use MPI (Message Passing Interface) [18]. MPI is a language-independent communication protocol used to program parallel computers. It is a message passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI technology tends to provide an efficient and portable standard for message passing communication programs used in distributed memory and parallel computing. There are currently several MPI implementations such as MPI/Pro, IBM MPI, and LAM. These implementations provide different communication modes such as asynchronous communication, virtual topologies and efficient message buffer management.

The problem of timed synchronous communication between two processes, \(P\) and \(Q\), can be stated as follows. To send a message to process \(Q\), process \(P\) executes:

\[
\text{MPI\_send (msg, count, mess\_type, dest, tag, MPI\_COMM)}
\]

Meanwhile, process \(Q\) executes a receive command:

\[
\text{MPI\_recv (msg, count, mess\_type, source, tag, MPI\_COMM, MPI\_STATUS)}
\]

MPI preserves the ordering and the integrity of the messages (for example, by ACK schemes and sequence numbers). This will ensure that values are not mixed up or out of sequence, provided that the \textit{send} actions are executed on one location in the same order as the corresponding \textit{recv} actions in the other location. Based upon the distribution specification, a unique tag is assigned to each common variable of the program. Figure 7 shows the communications in our case.

![Figure 7. MPI Communication](image-url)

To execute the distributed parts, we need to create configuration files to identify machines to be used for execution. Then the parts can be compiled locally and executed separately.

Our approach is a practical application of formal results that have been proved in the polychronous MoC. It has been shown that if \(|\{P_1|...|P_n\}|\) is such that \(P_1, \ldots, P_n\) are endo-isochronous (all of them are endochronous and the pairwise compositions of their restrictions on their common variables are endochronous) [6], assume the deployment is simply performed by using an asynchronous mode of communication between the different processes, then the original semantics of each individual process of the deployed GALS architecture is preserved (the sequence of values are preserved for all flows). For our example, comparing the execution results with the ones obtained from sequential simulation, we get the same outputs for the same inputs.

As a conclusion, from a complete representation of an application, including its virtual distribution on a target architecture, it is possible to make a global compilation, partitioning, insertion of communication features, and to simulate the application on the considered architecture.

**5 Related work**

We can mention a few related approaches addressing AADL model transformation. Dissaux [19] presents an
6 Conclusion

We have presented in this paper a polychronous model to work for the AADL architecture from the early stages of its functional specification to the late stages of its distributed simulation. Our approach has two main characteristics: 1) it is based on model transformation, from AADL dependency models to SIGNAL that can be processed by existing technologies and services, and 2) it reduces the dependency of the model partitions, since all the partitions are distributed.

Our model has been designed to generate synchronous models written in SIGNAL language, target simulations being distributed. The main advantage of this approach is that it provides a quite systematic way of modeling a distributed system, since directly designing a distributed system is always more difficult and error-prone. The other advantage of this approach is the ability to debug and formally verify the centralised program before its distribution, which is always easier and faster than debugging a distributed program.

For future works, we intend to design a simulation model for a loosely time-triggered architecture (LTTA) [26]. A big improvement of our current approach would be the support for automatic distribution of the PARTITIONS. Secondy, distributed real-time executives are expected to provide important fault-tolerance facilities. And we also want to translate the behavior annex to introduce the behavior transitions and actions into SIGNAL.

References

[16] Y. Ma, J-P. Talpin, T. Gautier, Virtual prototyping AADL architectures to a synchronous model, MEMOCODE’08, June 2008