Distributed simulation of AADL specifications in a polychronous model of computation [*]

Yue Ma Jean-Pierre Talpin Sandeep Kumar 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

In the avionics domain, the Architecture Analysis and Design Language (AADL) is often 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

It is well admitted, 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.

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 general asynchronous systems, while keeping as much as possible the advantages of synchronous programming [6].

SIGNAL is a domain-specific, synchronous data-flow,

[*]This work is partially supported by the ANR project TopCased.

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.

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.

In order to support the virtual prototyping, simulation and formal validation of early, component-based, distributed embedded architectures, we define a model of the AADL into the polychronous MoC of the SIGNAL programming language [12], and generate distribution code for simulation. The simulation model resulting from a SIGNAL program and a target multiprocessor has a formal specification allowing model verification.

Our main difficulty is to model asynchronous AADL descriptions into a polychronous model. We address it in two stages. First, we use an existing library of the SIGNAL environment, modeling real-time operating system services, to translate the AADL architectural concepts into SIGNAL programs. It proves to be a suitable and adequate library to model embedded architectures in the specific case of Integrated Modular Avionics (IMA [13]) considered in the TopCased project. Second, we distribute the generated code based on the AADL architecture specifications to different processors.

The paper is organized as follows. Section 2 presents a model of IMA architectures in SIGNAL. Section 3 explains the general principles of interpreting AADL concepts using the SIGNAL library of APEX services. In Section 4, we present how the generated SIGNAL programs are then automatically distributed to create a distributed simulation of the system. Section 5 discusses the related work and Section 6 draws our conclusions. The appendix gives a brief summary of the syntax and semantics of SIGNAL.

2 Modeling ARINC concepts in SIGNAL

The POLYCHRONY design environment includes a library in SIGNAL modeling the real-time executive services defined in the ARINC-653 standard [14].

APEX services The APEX services modeled in SIGNAL include communication and synchronization services used by PROCESSes (e.g. *SEND_BUFFER*, *WAIT_EVENT*, *READ_BLACKBOARD*), PROCESS management services (e.g. *START*, *RESUME*), PARTITION management services (e.g. *SET_PARTITION_MODE*), and time management services (e.g. *PERIODIC_WAIT*).

PARTITION-level OS The role of the PARTITION-level OS is to ensure the correct concurrent execution of PROCESSes within the PARTITION. A sample interface of the PARTITION-level OS is depicted in Figure 1.

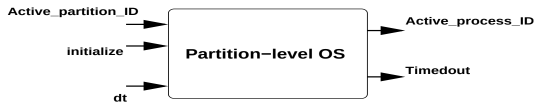


Figure 1. Interface of PARTITION-level OS

In Figure 1, the input `Active_partition_ID` represents the identifier of the running PARTITION selected by the module-level OS. When current, the PARTITION is executed. Then, the `PARTITION_LEVEL_OS` selects an active PROCESS within the PARTITION. The PROCESS is identified by the value carried by the output signal `Active_process_ID`, which is sent to each PROCESS.

ARINC PROCESSES The definition of an ARINC PROCESS model takes into account its computation and control parts. Two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE* (Figure 2). Any PROCESS is seen as a reactive component, which reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timedout` notifies PROCESSes of time-out expiration. The CONTROL and COMPUTE sub-components cooperate to achieve the correct execution of the PROCESS.

The CONTROL sub-component specifies the control part of the PROCESS. Basically, it is a transition system that indicates which statements should be executed when the PROCESS model reacts. Whenever the input `Active_process_ID` identifies the ARINC PROCESS, this PROCESS "executes". Depending on the current state of the transition system representing the execution flow of the PROCESS, a *block* [15] of actions in the COMPUTE sub-component is selected to be executed. The COMPUTE sub-component describes the actions computed by the PROCESS. It is composed of *blocks* of actions. The statements associated with a *block* are assumed to *complete within a bounded amount of time*.

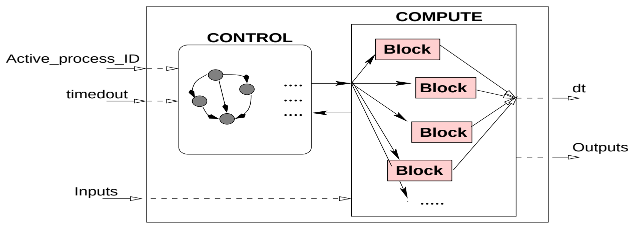


Figure 2. ARINC PROCESS model.

PARTITIONS After initialization, a PARTITION is activated (i.e. when receiving *Active_partition_ID*). The PARTITION-level OS selects an active PROCESS within the PARTITION. Then, the CONTROL subpart of each PROCESS checks whether or not the concerned PROCESS can execute. In the case a PROCESS is designated by the OS, the following actions will be performed: the PROCESS executes a *block* from its COMPUTE subpart, and the duration corresponding to the executed *block* is returned to the PARTITION-level OS in order to update its timing counters.

3 From AADL models 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 [16]. 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 general rules to translate AADL systems into the SIGNAL programming language. We put our translation to work by studying the similarity between the AADL and the APEX-ARINC services. In the following, we present the translation rules from three categories: system, software components and hardware components. For space limitation, we only present some classical components, for the detailed translation, the readers may refer to [17].

3.1 System

The system is the top-level component of the AADL model, mixing hardware and software components. 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 seperate it into several PARTITIONS according to its owned processors.

Rules:

1. The top level system (may contain several sub-systems) is mapped to the top level SIGNAL *process* (which contains all the PARTITIONS translated from the sub-systems).
2. Each sub-system is seperated into one or several PARTITIONS according to the processors it owns, each processor and its bound processes compose an ARINC PARTITION.
3. For each PARTITION, the input (output) port of the process which is connected to the system is mapped into an input (output) of the PARTITION.
4. For system implementations, each sub-component is mapped into a SIGNAL *process*, for example, an AADL thread can be mapped as an ARINC PROCESS, a subprogram can be a *block*.
5. The top level SIGNAL *process* will call each of the PARTITIONS.
6. The connections and communications between systems will be added later (in the distribution step, which will be explained in the next section).

For instance, the client system in the example (Figure 3 left) has only one processor and the corresponding SIGNAL model contains only one PARTITION (Figure 3 right), which is composed of the processor and the bound processes.

3.2 Software components

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 consider that the AADL processes executed on the same processor constitute a PARTITION.

Thread A thread is a concurrent schedulable unit of sequential execution through source code. Multiple threads represent concurrent paths of execution. A variety of execution properties can be assigned to threads, including timing, dispatch protocols, memory size and processor binding. In APEX ARINC, PROCESSES represent the executive unit for an application. They share common resources and execute concurrently within a PARTITION. A set of unique attributes are defined for each PROCESS. These attributes differentiate between unique characteristics of each PROCESS as well as defined resource allocation requirements. Fixed attributes, for example, Entry point, Base priority, Period and so on, are statically defined and cannot be changed

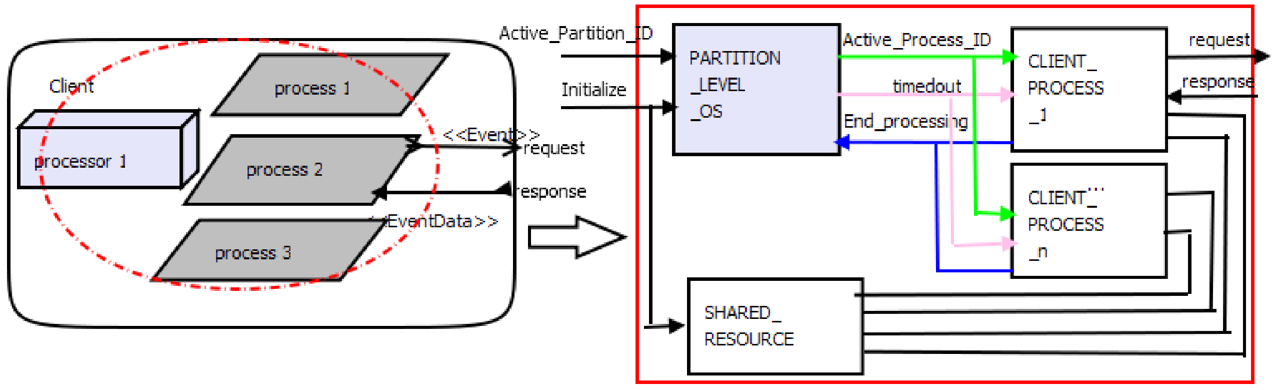


Figure 3. Mapping an AADL system

once the PARTITION has been loaded. 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 with or without parameters 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. The detailed implementation of the function can be programmed in C/JAVA language.

3.3 Hardware components

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 PARTITION_LEVEL_OS selects an active PROCESS within the PARTITION whenever the PARTITION executes, that is to say, at any time, there is only one PROCESS that is activated. 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 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.

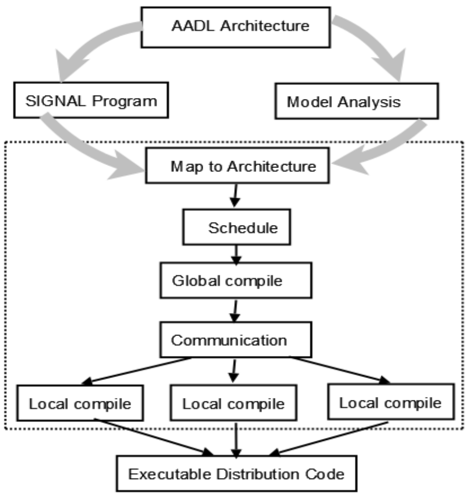


Figure 4. Distributed code generation

4.1 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.

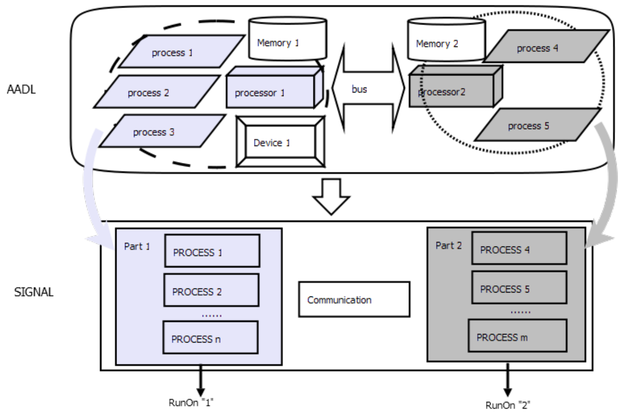


Figure 5. Distribution

Each PARTITION is paced by its own clock. The SIGNAL pragma "RunOn" in the POLYCHRONY environment is used for partitioning information: when a partitioning 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 [18] (an endochronous process is mostly insensitive to internal and external propagation delays). The program P will be rewritten as `(|P1|...|Pn|)` where n is the number of processors. This step is only a syntactic restructuration of the original program. The SIGNAL process `Pi` is annotated with a pragma "RunOn i". Here is a simple example for distributing two PARTITIONS run on different processors:

```
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 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 MODULE_LEVEL_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 PARTITION_LEVEL_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 MODULE_LEVEL_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):

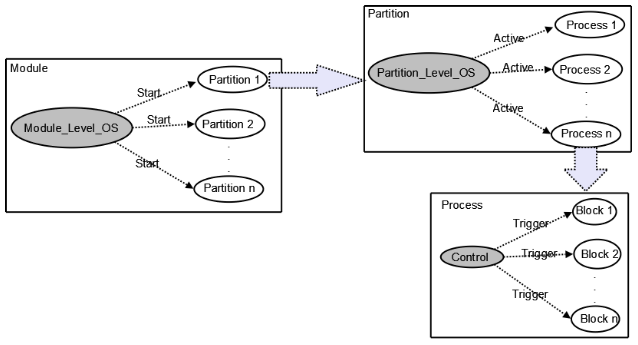


Figure 6. Scheduling

```
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 ^= when (not b_init)|)|)
where boolean b_init;  end;
```

After adding the scheduler to the program, now we can make a global compiling to seperate 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

process DIST_1=(?boolean request,H; Mess_type y;!event response; Mess_type x;)

pragmas RunOn "1" end pragmas (|...|);

PART 2

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_i (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 transfered from DIST_1 to DIST_2, and DIST_1 will receive message y from DIST_2.

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

4.3 Adding communications

A distributed program consists of a set of processes interacting with the enviroment 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.

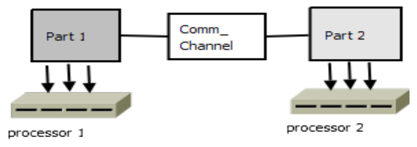


Figure 7. Communication

Therefore some form of communication and synchronization mechanism which guarantees that the semantics of synchronous communication will be preserved (Figure 7) 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) [19]. 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:

```
MPI_send (msg, count, mess_type, dest, tag, MPI_COMM)
```

Meanwhile, process Q executes a receive command:

```
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 *send* actions are executed on one location in the same order as the corresponding *recv* actions in the other location. Based upon the distribution specification, a unique tag is assigned to each common variable of the program. Figure 8 shows the communications in our case.

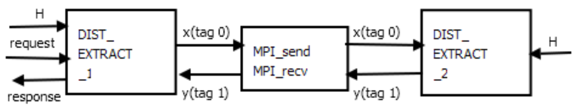


Figure 8. MPI Communication

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. For our example, comparing the execution results with the ones getting 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

A number of related approaches have been proposed. Dissaux [20] presents an approach for AADL model transformations. This approach concentrates on the analysis of components from legacy code aimed specifically towards use with the HOOD Stood tool [20]. Bertolino and Mirandola [21] propose an approach for the specification and analysis of performance related properties of AADL components using the RT-UML profile.

Some research projects also use AADL as a modeling language for distributed systems. PolyORB-HI [22] is a middleware for High-Integrity systems, modeling distributed high-integrity systems using AADL, allowing a large part of the distributed application code to be generated automatically, and making Ada code generation from AADL model. AADL2Fiacre [23] deals with the transformation of AADL models into Fiacre models to perform formal verification and simulation.

Some related approaches are proposed to modeling non-synchronous systems using synchronous languages and developing system level design methodology. For instance, AADL2SYNC [24] tool is an AADL to synchronous programs translator, which is extended in the framework of the European project ASSERT, resulting in the system-level tool box translating AADL to LUSTRE.

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 dependability 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 loosedly time-triggered architecture (LTTA) [25]. A big improvement of our current approach would be the support for automatic distribution of the PARTITIONS. Secondly, distributed real-time executives are expected to provide important fault-tolerance facilities.

References

[1] P.H. Feiler, D.P. Gluch, J.J. Hudak. **The Architecture Analysis & Design Language (AADL): An Introduction**, Technical Note CMU/SEI-2006-TN-011, February 2006

[2] P.H. Feiler, D.P. Gluch, J.J. Hudak. **Embedded system architecture analysis using SAE AADL**, Technical note cmu/sei-20040tn-005, Carnegie Mellon University, 2004

[3] http://www.assert-online.org

[4] http://www.LAAS.fr/COTRE

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, **The synchronous languages 12 years later**, Proc of the IEEE, 91(1), January 2003

[6] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, **Concurrency in synchronous systems**, ACSD'04, June 2004

[7] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, **Programming real-time applications with Signal**, Proceedings of the IEEE, v.79, September 1991

[8] N. Halbwachs et al. **The synchronous data flow programming language LUSTRE**, Proceedings IEEE, 79-9, 1991

[9] B.De Simone, **The ESTEREL language**, Proceedings IEEE, 79-9, 1991

[10] TopCased project, http://www.topcased.org

[11] http://www.irisa.fr/espresso/Polychrony

[12] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, **Polychrony for system design**, Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design, World Scientific, April 2003

[13] Airlines Electronic Engineering Committee. **Design Guidance for Integrated Modular Avionics**. ARINC Report 651-1, November 1997

[14] Airlines Electronic Engineering Committee. **Avionics Application Software Standard Interface**. Arinc Specifcation 653, January 1997

[15] A. Gamatié, and T. Gautier, **Synchronous modeling of avionics applications using the Signal language**. In Proc of the 9th IEEE RTAS'2003, May 2003, IEEE Press

[16] J. Hudak, P. Feiler, **The SAE Architecture Analysis & Design Language (AADL) Standard: A Language Summary**, AADL Standard Document, 2006

[17] Y. Ma, J-P Talpin, T. Gautier, **Vitual prototyping AADL architectures to a synchronous model**, MEMOCODE'08, June 2008

[18] J-P. Talpin, J. Ouy, L. Besnard, P. Le Guernic. **Compositional design of isochronous systems**, Report RR-6227. INRIA, 2007. URL http://hal.inria.fr/inria-00156499

[19] J. Dongarra, D. Walker, E. Lusk, **MPI: A Message-Passing Interface Standard**, MPI Standard

[20] P. Dissaux, **AADL model transformations**, Proc DASIA 2005 Conference in Edinburgh, UK, 2005

[21] A. Bertolino, R. Mirandola, **Modeling and Analysis of Non-functional Properties in Component-Based Systems**, Electronic Notes in Theoretical Computer Science 82(6), 2003

[22] http://aadl.enst.fr/polyorb-hi/

[23] B. Berthomieu, J-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, **Fiacre: an Intermediate Language for Model Verification in the Topcased Environment**, ERTS'08, France, 2008

[24] AADL2SYNC project, available from http://www-verimag.imag.fr/~synchron/index.php?page=aadl2sync

[25] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J-P. Talpin, and S. Tripakis, **A protocol for loosely time-triggered architecture**, EMSOFT, 2002

A The SIGNAL Language

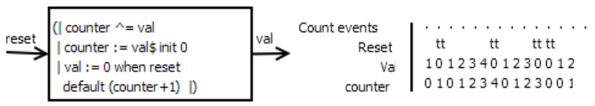
SIGNAL is a data-flow synchronous language. A SIGNAL program, also called *process* (to distinguish from the AADL "process" and ARINC "PROCESS"), is a system of equations over *signals* that specifies relations between values and clocks of the *signals*. SIGNAL is associated with a design environment, called POLYCHRONY, which offers a graphical user interface, a compiler and a model-checker.

Syntax In Signal, a process (written P or Q) consists of the synchronous composition (noted $P \parallel Q$) of equations on signals (written $x = y\ f\ z$). A signal x represents an infinite flow of values. It is sampled according to the discrete pace of its clock, noted $\hat{x}$. 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. A process defines the simultaneous solution of the equations it is composed of.

$$P, Q \ ::= \ x = y\ f\ z \mid P \parallel Q \mid P/x$$

There are several kinds of equations. A functional equation $x = y\ f\ z$ defines an (arithmetic or boolean) relation f between its operands y, z and result x. A delay $x = y\,\mathsf{pre}\,v$ initially defines x by the value v and then by that y had last time it was evaluated. A sampling $x = y\,\mathsf{when}\,z$ defines x by y when z is true. A merge $x = y\,\mathsf{default}\,z$ defines x by y when y is present and by z otherwise. The process P/x restricts the lexical scope of the signal x to the process P.

Example We consider the definition of a counter. It accepts an input **reset** *signal* and delivers the integer output *signal* **val**. The local variable **counter** is initialized to 0 and stores the previous value of the *signal* **val** (equation **counter := val$ init 0**). When an input **reset** occurs, the signal **val** is reset to 0 (expression (**0 when reset**)). Otherwise, the *signal* **val** takes an increment of the variable **counter** (expression (**counter+1**)). The activity of **Count** is governed by the clock of its output **val** which has higher frequency than its input **reset**.



```
(| counter ^= val
 | counter := val$ init 0
 | val := 0 when reset
   default (counter+1)  |)
```

```
Count events | . . . . . . . . . . . . . .
       Reset |  tt       tt      tt tt
         Va  | 1 0 1 2 3 4 0 1 2 3 0 0 1 2
     counter | 0 1 0 1 2 3 4 0 1 2 3 0 0 1
```

Figure 9. A counter and its trace

Semantics The semantics of SIGNAL is based on the polychronous MoC. SIGNAL handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as x and implicitly indexed by discrete time. At any instant, a *signal* may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. The set of instants where a *signal* x is present represents its *clock*, noted $\hat{x}$. SIGNAL relies on six primitive constructs that define *elementary processes*:

• *Relations.* `y:= f(x1,...,xn)` $\equiv^{def}$ $\forall t$: $(y_t = \perp \Leftrightarrow \forall i\ xi_t = \perp \wedge \exists i\ xi_t = \perp \Rightarrow \forall i\ xi_t = \perp \wedge \exists xi_t \neq \perp \Rightarrow y_t = f(x1_t, ..., xn_t))$

• *Delay.* `y:= x$1 init c` $\equiv^{def}$ $\forall t > 0, x_t \neq \perp \Leftrightarrow y_t \neq \perp \ \wedge x_t \neq \perp \Rightarrow y_t = x_{t-1}, y_0 = c$.

• *Undersampling.* `y:= x when b` $\equiv^{def}$ $y_t = x_t$ if $b_t = true$, else $y_t = \perp$.

• *Deterministic merging.* `z:= x default y` $\equiv^{def}$ $z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.

• *Composition.* `P1|P2` is the conjunction of the equations in `P1` and `P2`.

• *Hiding.* `P where x` restricts the scope of `x` to the *process* `P`.