

A mapping from AADL to Java-RTSJ

[Extended Abstract] *

Bodeveix Jean-Paul
IRIT
Université Paul Sabatier
Toulouse, France
bodeveix@irit.fr

Cavallero Raphaël
IRIT
Université Paul Sabatier
Toulouse, France
cavaller@irit.fr

Chemouil David
CNES
Toulouse, France

Filali Mamoun
IRIT
Université Paul Sabatier
Toulouse, France
filali@irit.fr

Rolland Jean-François
IRIT
Université Paul Sabatier
Toulouse, France
rolland@irit.fr

ABSTRACT

In this paper, we study a mapping from AADL to Java-RTSJ. After reviewing the basic concepts of the AADL execution model, we present the basic notions of Java-RTSJ, we rely on, for our mapping. Then, we propose a mapping taking into account a given subset of AADL. A related works section reviews existing works and elaborates on some comparisons.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*description languages*; D.4.7 [Operating Systems]: Organization and Design—*real-time systems and embedded systems*; D.2.4 [Software Engineering]: Software/Program Verification—*model checking*; D.3.1 [Programming Languages]: Formal Definitions and Theory—*semantics*

General Terms

Design, Languages, Verification

Keywords

Architecture Description Languages, Real-time systems, Architecture Analysis and Design Languages (AADL), Real-Time Specification for Java (RTSJ), Java

1. INTRODUCTION

During the last years, AADL has emerged as a solution for the description of real time designs. Actually, it has been

*A full version of this paper is available as <ftp://ftp.irit.fr/IRIT/ACADIE/full-jtres07.pdf>. This study has been partially funded by CNES and ASTRIUM.

proposed and accepted as a standard of the SAE [2]; moreover, many institutions currently study it in order to adopt it for their future projects. For instance, in the french nationwide project TOPCASED [12], AADL has been selected and is the first real time specific description language to be supported by the TOPCASED development platform.

In this paper, we are interested in the mapping of AADL to real time kernels. Actually, although AADL is recognized as offering features allowing a precise analysis, to the best of our knowledge, there is no current mapping to some real time execution platform. For this purpose, we adopt a generic solution: we study the mapping to RTSJ [14]. The choice of RTSJ is motivated by two facts: RTSJ has already been considered as a possible execution kernel for space applications [8], moreover since we are also concerned by establishing the correctness of our mapping, the availability of precise descriptions as well informal [4, 14] as well formal [8, 15] was crucial.

The rest of this paper is organized as follows. Section 2 presents the main AADL aspects we are concerned with in this paper. Section 3 is an overview of the *Real Time Specification for Java*. It focuses on the features that will be used by the implementation of the mapping from AADL to RTSJ. In section 4 we describe the mapping of a subset of the AADL to real time Java by giving the main features of the kernel library. model. Section 5 is about related works: we elaborate some comparisons with the Giotto Architecture Description Language [11]. Section 6 draws some conclusions.

2. AADL

AADL is an architecture design language standardized by the SAE. This language has been created to be used in the development of real time and embedded systems. As a successor of MetaH [13], AADL capitalizes more than 10 years of experiments. MetaH is a language developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications. AADL also benefits from the knowledge on ADLs acquired at CMU during the development of several ADLs, like ACME [6] and Wright [3].

2.1 The language

AADL includes all the standard concepts of any ADL [9]: components, connectors used to describe the interface of components, and connections used to link components. The set of AADL's components can be divided in three partitions, the software components (process, thread, thread group, subprogram, and data), the hardware components (processor, bus, memory, device), and a System component. Components can communicate through ports, synchronous calls, and shared data. A process represents a virtual address space, or a partition, this address space includes the program defined by its sub-components. A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution, it's the only AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another program. A data models a static variable used in the code, they can be shared by threads or processes.

A processor is an abstraction of the hardware and the software in charge of the scheduling and the execution of threads. The memory represents any platform component that stores data or binary code. The buses are communication channels used to connect different hardware components. The devices represent interfaces between the system described and its environment.

Systems allow to compose software components with hardware components. The interactions can be defined at a logical and a physical level. At a physical level, software components are associated to hardware component, a thread to a processor, or a data to a memory for example. The logical level is used to describe the communication between hardware and software. At a logical level we can define communication connections between processors or devices and software components.

AADL uses the notion of mode to determine a set of active components. This mechanism allows to describe dynamic architectures. The set of active components can be modified by the reception of an event. The AADL standard describes a strict semantics of execution, this semantics is customizable using properties. We will present only a subset of AADL. We don't take into account the hardware components. Modes are not modeled yet, but it is planned to integrate them in our model.

2.2 Communication through ports

AADL proposes three types of ports: data, event and event data ports. A port is declared to be in an input, output or input/output mode. It can be used to transmit data or control or both. Ports are used to describe the interface of a component. Data transmitted through ports is typed. Each input port has a fresh variable to define the state of the port, if a port has not received anything between two thread dispatches this variable is set to false. A buffer is also associated with each input port, when an output port sends a data or an event it modifies these buffers. On the dispatch of a thread these buffers are copied into the local memory of the thread. Some properties permit to customize the behavior of event and event data ports. The property "Queue_size" determines the maximum number of events that can be re-

ceived. "Overflow_handling_protocol" describes the behavior of the port in case of overflow, the two default politics are drop newest and drop oldest. The "Dequeue_protocol" describe the way elements in the queue are accessed, one by one ("OneItem") or all at once ("AllItems"). Data ports have the simplest behavior, data is sent at the end of the thread's execution and is received at the next dispatch of the receiving thread. Event and event data ports have a very close behavior, they can send an event or event data anytime during the execution of a thread. Events or events data sent are queued in the destinations ports. Input event and event data ports are delivered at the dispatch of the thread. For periodic threads that are harmonic, a data connection can be declared as immediate or delayed. If the connection is delayed data is sent at the end of the period of the sending thread. If the connection is immediate the receiving thread must wait the sending thread to complete and it receives data at the start of its execution.

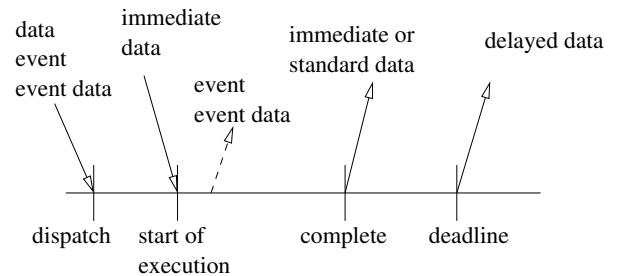


Figure 1: *communication through ports in AADL.*

2.3 Scheduling strategy

2.3.1 Thread models

Threads are the only components that have an execution semantics. AADL supports the classic types of dispatch protocols, a thread can be declared as periodic, aperiodic, sporadic or background. All the standard properties (WCET, deadline, ...) used to describe a real-time system exist in AADL. Threads have two predeclared event ports : dispatch and complete. The dispatch port is used for aperiodic or sporadic threads. If this port is connected all other ports of the thread do not trigger the dispatch. It's a very common behavior for an aperiodic or a sporadic thread to send an event on completion. In AADL, we do not specify when an event is sent. The complete event port is used to send an event at the end of the execution.

2.3.2 Basic scheduling strategy

All the thread have the same life cycle, this cycle can be represented as an automaton (see fig. 2). All threads start in the awaiting dispatch state. The dispatch condition depends on the thread's type. If the thread is periodic it will be dispatched at every period. At this time, delivery occurs for all its input ports. An aperiodic or a sporadic thread that does not have its dispatch ports connected is dispatched each time it receives an event. Delivery occurs only for the port that triggers the dispatch. If its dispatch port is connected, it is dispatched each time it receives an event on this port, and delivery occurs for all its others ports. The thread in the running state that has the maximum priority starts or

continues its execution. The priority of the thread is determined by the chosen scheduling policy (RMA, EDF, LLF). This policy is specified by a property of the model. When a thread is dispatched it can have a higher priority than the executing thread. In this case, the executing thread is pre-empted and goes back to the running state. When a thread ends its execution it goes to the “awaiting.dispatch” state until its next dispatch. At this time, all the output data ports of the thread are read and their content is sent to their respective destination ports.

3. RTSJ

RTSJ [14] is the *Real-Time Specification for JAVA* (JSR-1, 1998). RTSJ offers a new library (*javax.realtime*) for real-time compliant Java applications programming. Several free and commercial implementations are available : Timesys (reference implementation), Aicas (Jamaica), Aonix (Perc), etc.

3.1 Memory management

RTSJ offers the possibility to use memory areas that are different from the Java *heap*. These areas do not suffer from the non-deterministic activity of the *garbage collector*. In particular, RTSJ defines an *ImmortalMemory* area where allocated objects are never destroyed until the program termination. As AADL models consider only statically allocated objects that are not supposed to be destroyed until the program terminates, all the objects described later will be allocated in the *ImmortalMemory* and the *garbage collector* will not be taken into account.

3.2 Threads in the RTSJ

RTSJ's *RealtimeThread* class defines threads with three main advantages compared to Java threads :

- *RealtimeThreads* can be allocated in any memory area, including RTSJ's *non heap* memory areas like the *ImmortalMemory*. To highlight the difference between *heap* and *non heap* memory using threads, RTSJ defines a *NoHeapRealtimeThread* class that cannot be allocated in the *heap*.
- when a priority is given to a Java thread, it is only used by the scheduler as an indication. In the RTSJ, priority rules given to threads are actually enforced by the scheduler.
- RTSJ provides explicit support for periodic and non periodic threads, via the *ReleaseParameters* class and its three subclasses : *PeriodicParameters*, *SporadicParameters*, *AperiodicParameters*.

Important. Unfortunately, while periodic threads are efficiently supported by the RTSJ, aperiodic and sporadic threads are only supported as one-shot threads. The programmer must provide his own support for “reusable” aperiodic and sporadic threads (see [14, page 255]).

3.3 Events and event handlers

3.3.1 Asynchronous event

RTSJ provides support for asynchronous events (e.g. alarms) via the *AsyncEvent* class. Events can be triggered by calling the method *fire()* defined in this class.

3.3.2 Asynchronous event handler

Asynchronous events are associated with handlers using the method *addHandler(AsyncEventHandler aeh)*. An asynchronous event handler is associated with a real time thread. This implies that it can be allocated in the desired memory area, in particular *non heap* memory. The association between a handler and its thread is performed dynamically. To avoid the time overhead, a thread can be bound to a handler at initialisation using the *BoundAsyncEventHandler* class.

When an event is triggered, its associated handler's *fire-Count* is increased and the handler is released to execute the *handleAsynchEvent()* method repeatedly until the fire-Count equals zero.

3.3.3 Timers

Timers extend the *AsyncEvent* class and provide a way to trigger an event after a given time has elapsed or on a specific date. The RTSJ supports both periodic and one shot timers. Periodic timers fire repeatedly at a fixed period, one shot timers fire only once, unless they are restarted. Timers can be stopped, restarted (the timer is reset and started again) and rescheduled (the timer will not fire until the date it has been rescheduled to).

4. AADL MAPPING TO RTSJ

The purpose of this mapping is to produce a real time compliant Java library. This library could be used either to write real time AADL-compliant Java programs with respect to the execution model, or to be the target of a code generator operating on a source AADL specification.

4.1 Supported subset of the AADL

The following elements of the AADL execution model are supported:

- Threads *Dispatch_Protocol*: periodic, aperiodic and sporadic.
- Ports and port connections: data (*sampling*, *immediate*, *delayed*), event and event data.

4.1.1 Execution model

Threads. This mapping considers a simplified execution model of AADL threads (see figure 2) with three states:

- suspended awaiting dispatch: if the thread is periodic, it waits for the beginning of the next period; if the thread is not periodic, it waits for the arrival of an event or event data.
- scheduled for execution: the thread has been dispatched and now waits for the scheduler to allow its execution,
- running: the thread is actually executing.

When a thread moves from the *suspended awaiting dispatch* state to the *scheduled for execution* state, it reads the values written by other threads in its input ports. When a thread moves from the *running* state to the *suspended awaiting dispatch* state, it writes the new values it has produced to its output ports. The thread that is running can be preempted by the scheduler so that another thread with a higher priority can run, the first thread returns to the *scheduled for execution* state.

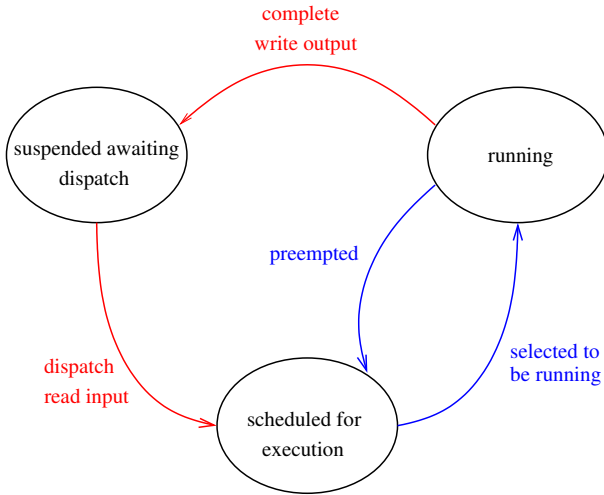


Figure 2: The simplified AADL threads' execution model.

Modes. The current mapping does not take execution modes into account. However, the current structure of the kernel is designed to allow easy transition to modes using. The system is considered to be running in one mode (the initial mode) and no mode switch can occur.

4.1.2 Communications

AADL communications semantics has gone through major changes between AADLv1 and AADLv2. The current mapping is mainly v1 compliant and we will talk only about v1 features of the kernel. However, it can be noticed that some AADLv2 features are already available, for instance: *Input_time* and *Output_time* properties for ports, *sampling*, *immediate* and *delayed* data port connections, dispatch conditions for aperiodic and sporadic threads.

4.2 Overview of the kernel

4.2.1 Global structure

The communication structure in this kernel can be described as follows (see also figure 3):

- threads that write output to *out* ports and read input from *in* ports,
- connections that link two or more ports,
- a router that centralizes all these connections and interfaces with threads to transmit data, events, and event data.

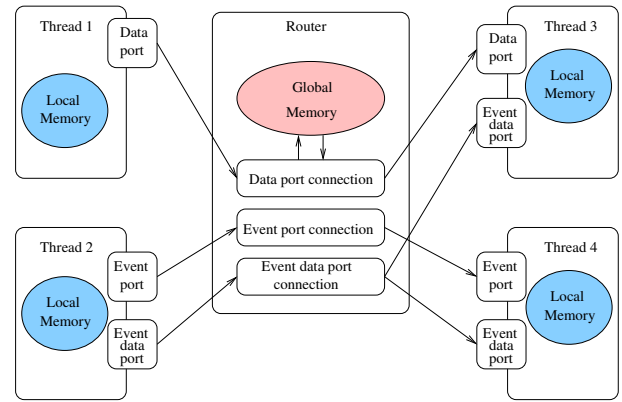


Figure 3: Overview of the communications structure.

The kernel is divided in two categories:

- User classes representing the elements that the user is allowed to manipulate. For instance, in order to :
 - create threads,
 - create ports, access ports and their local memory,
 - create port connections.
- System classes that contain only protected methods. They ensure that the behaviour of the system is that of an AADL system, essentially by
 - performing communications at the right timings,
 - forcing threads behaviour to match AADL threads' execution model.

4.3 User classes

These classes contain public methods for the user.

4.3.1 components

We give a generic interface for all user-defined AADL components allowed to have ports : the *AADLPortOwner* interface (listing 1).

Listing 1: the *AADLPortOwner* interface

```
public interface AADLPortOwner {

    /* allow connection to a routing
     * environment */

    public void
        setRouter(AADLRouter env);
    public AADLRouter getUserRouter();

    /* getters for user-defined ports: */

    public AADLOutDataPort []
        get_out_data_ports();
    public AADLInDataPort []
        get_in_data_ports();
    public AADLInOutDataPort []
        get_inout_data_ports();
    public AADLOutEventPort []
        get_out_event_ports();
}
```

```
(...)
public AADLInOutEventDataPort []
    get_inout_event_data_ports ();
}
```

However, the only AADL components with ports that we consider in this kernel are threads.

4.3.2 Threads (see 2.3.1 for AADL)

All the threads, including the “primordial” thread, and, therefore, all objects created when the application is started, are allocated in the immortal memory. Scoped memory is not used.

In our simplified execution model, threads are components that have ports (i.e. *AADLPortOwner* objects) and define:

- a method that contains the code the thread has to execute once it is running,
- predeclared ports : *Dispatch* (in event port), *Complete* (out event port) and *Error* (out event data port),

Thus, user defined threads are specified by the *AADLRunnable* interface (listing 2).

Listing 2: the *AADLRunnable* interface

```
public interface AADLRunnable extends
    AADLPortOwner{

    /** Support for AADL entry point
     * subprogram. */

    public void entry_point();

    /** getters for threads predeclared ports:
     */

    public AADLInEventPort
        getPredeclared_Dispatch_Port ();
    public AADLOutEventPort
        getPredeclared_Complete_Port ();
    public AADLOutEventDataPort
        getPredeclared_Error_Port ();
```

4.3.3 Ports (see 2.2 for AADL)

The user ports classes represent the input and output local memory of the thread. When the thread is dispatched, input ports are frozen, i.e. their content (or a part of it) is not updated until the next dispatch.

Port types. Ports that carry data (data and event data ports) are type. To have a generic approach for ports, we do not care about data types (only objects are transmitted). However, to ensure examples correctness, we will use a small AADL package where the data type *int* is declared. Ports in AADL examples will be typed using this data. Therefore, ports in Java examples will transmit integers.

```
package base
public
    data int
```

```
end int;
end basic;
```

Example. AADL threads write their output in ports and read their input from other ports. A sample AADL thread type with three different ports is given in the listing 3.

Listing 3: AADL port specifications

```
thread leThread
    features
        odp: out data port basic::int;
        iep: in event port{
            Dequeue_Protocol => OneItem;
            Overflow_Handling_Protocol => Error;
            Queue_Size => 10;
        };
        ioedp: in out event data port basic::int{
            Dequeue_Protocol => AllItems;
            Overflow_Handling_Protocol =>
                DropOldest;
            Queue_Size => 5;
        };
    end leThread;
```

The mapping of these ports to Java-RTSJ is given by the listing 4.

Listing 4: AADL ports in java-RTSJ

```
myData = new UserData ();

/* this will represent the type
 * of the ports: */

UserDataSerializer serializer =
    new UserDataSerializer ();

/* the out data port: */

AADLOutDataPort odp =
    new AADLOutDataPort (this ,
        serializer ,
        myData);

/* the in event port: */

AADLInEventPort iep
    = new AADLInEventPort (this ,
        AADLInEventPort.ONE_ITEM,
        AADLInEventPort.ERROR,
        10);

/* the in out event data port: */

AADLInOutEventDataPort ioedp
    = new AADLInOutEventDataPort (this ,
        serializer ,
        AADLInEventPort.ALL_ITEMS,
        AADLInEventDataPort.DROP_OLDEST,
        5);

/* using the ports: */

odp.send(myData);
int count = iep.getCount ();
ioedp.getValue(myData);
ioedp.raise_event(myData);
```

Data ports. Data must be serialized before it is transmitted and deserialized before it is given to the user. The serializer/deserializer provides support for this requirement and represents the data type of the port. This serializer/deserializer is *not* based on java serialization but uses statically allocated *ByteBuffer* objects to avoid memory leaks and be more flexible. We have based our serialization methods on the work done by Gérard Padiou [10].

The interface of *in* and *out* data ports is given below:

```
public AADLInDataPort(AADLPortOwner parent,
    ObjectSerializer serializer);

public void getValue(Object fillMe);

public boolean fresh();
```

```
public AADLOutDataPort(AADLPortOwner parent,
    ObjectSerializer serializer,
    Object initial_value);

public void send(Object objectToSend);
```

Event ports. Event ports trigger events. These events are queued at receiving ports. The user can access counters that represent the number of events that were queued at a port since the previous dispatch.

Different properties can be specified for the queue of *in* event (data) ports in the AADL model:

- *Dequeue_Protocol* : *OneItem* or *AllItems*,
- *Overflow_Handling_Protocol* : *Error*, *DropOldest* or *DropNewest*
- *Queue_Size*.

The interface of *in* and *out* event ports is given below ¹:

```
public AADLInEventPort(AADLPortOwner parent,
    int dequeue_Protocol,
    int overflow_Handling_Protocol,
    int queue_Size);

public int getCount();
```

```
public AADLOutEventPort(AADLPortOwner parent)
;

public void raise_event();

public void when_complete();
```

Event data ports. Event data ports act as event ports, transmitting data along with events. Data values can be dequeued, one after the other, from receiving ports.

¹Java 5 *enums* are not used for the different protocols as RTSJ is based on Java 1.4

In out ports. *In out* ports offer both *in* and *out* ports methods. The interface of an *in out* event data port is given as an example in the listing 5 below. When output is written to an *in out* port, the input value is overwritten.

Listing 5: An in out event data port

```
public AADLInOutEventDataPort(AADLPortOwner
    parent,
    ObjectSerializer serializer,
    int dequeue_Protocol,
    int overflow_Handling_Protocol,
    int queue_Size,
    Object initial_value);

public void getValue(Object fillMe);

public int getCount();

public void raise_event(Object data);

public void when_complete(Object data);
```

4.3.4 Port connections

A port connection links two or more sibling ports together. Port connections are directional. An AADL port connection maps to an *AADLxxxxConnection* object in Java-RTSJ, where *xxxx* is one of {Data, Event, EventData}. Parameters are: a source port (an *out* port) and one or more receiving ports (*in* ports), depending on the class. The kernel currently supports: one-to-one data connections and one-to-n event and event data connections.

Example: an AADL specification and the Java-RTSJ implementation. The listing 6 shows an extract of a sample AADL specification, where a process is defined. This process contains two threads that exchange data through two data port connections.

Listing 6: Connections in AADL

```
(...)
process testProcess
end testProcess;
process implementation testProcess.impl
subcomponents
t1: thread testThread1.impl;
t2: thread testThread2.impl;
connections
data_connection_12:
data port lt1.odp -> lt2.idp;
data_connection_21:
data port lt2.odp -> lt1.idp;
end testProcess.impl;

thread testThread1
features
odp: out data port basic::int;
idp: in data port basic::int;
end testThread1;
thread implementation testThread1.impl
end testThread1.impl;

thread testThread2
features
odp: out data port basic::int;
idp: in data port basic::int;
end testThread2;
```

```

thread implementation ttestThread2.impl
end testThread2.impl;

```

Figure 4 shows a standard graphical representation of the process *testProcess*.

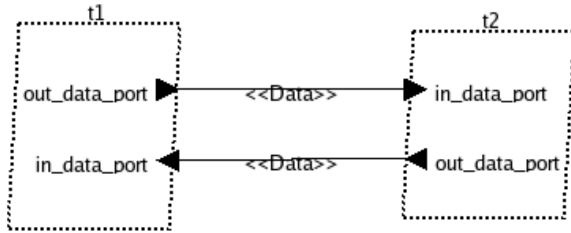


Figure 4: Implementation of testProcess

In this example, we have an AADL data port connection, therefore we use the *AADLDataConnection* class.

First, we have to create runnables for *testThread1* and *testThread2*, with the needed ports. For example, the code of the *testThread1* class that implements *AADLRunnable* is given by the listing 7.

Listing 7: An AADLRunnable implementation for testThread1

```

public class testThread1 implements
  AADLRunnable {

  private AADLRouter env;

  /* define arrays of ports: */
  private AADLOutDataPort [] odp =
    new AADLOutDataPort [1];
  private AADLInDataPort [] idp =
    new AADLInDataPort [1];

  /* use integers as data type: */
  IntegerSerializer is =
    new IntegerSerializer ();
  private AADLInteger idp_delivered =
    new AADLInteger (0);

  (...)
  public testThread1 () {
    (...)
    /* create user ports: */
    odp [0] =
      new AADLOutDataPort (this ,
        is ,
        idp_delivered);
    idp [0] =
      new AADLInDataPort (this , is);
  }

  public void entry_point () {
    /* whatever */
    (...)
  }

  /* permit the system to collect
  user ports: */

  public AADLOutDataPort []

```

```

  get_out_data_ports () {
    return odp;
  }

  public AADLInDataPort []
  get_in_data_ports () {
    return idp;
  }

```

Then, we can connect the ports of these runnables, as shown by the listing 8.

Listing 8: Creating the data port connections

```

/* testThread1, testThread2 implement
 * the AADLRunnable interface. */

testThread1 t1_r = new testThread1 ();
testThread2 t2_r = new testThread2 ();

int odp_index = 0;
int idp_index = 0;

AADLDataConnection c12 = new
  AADLDataConnection (
    t1_r.get_out_data_ports () [odp_index] ,
    t2_r.get_in_data_ports () [idp_index]);

AADLDataConnection c21 = new
  AADLDataConnection (
    t2_r.get_out_data_ports () [odp_index] ,
    t1_r.get_in_data_ports () [idp_index]);

```

4.4 System classes

These classes generally contains only private and protected methods. However, some constructors are public, as for threads.

4.4.1 The router

The router is aware of all the connections specified in the system. Therefore, connections are collected in a *ConnectionsTable* that the router takes as parameter.

```

public ConnectionsTable (
  AADLDataConnection []
  userDataConnections ,
  AADLEventConnection []
  userEventConnections ,
  AADLEventDataConnection []
  userEventDataConnections);

```

The structure of the router is given by the figure 5.

- For each data connection, memory is allocated in the router. Data will be either stored in this memory when the associated thread completes, or retrieved by receiving threads when they are dispatched.
- Events can trigger a non-periodic thread dispatch, depending on the receiving thread's state when the event occurs. Thus, the router provides methods for an output event or event data port to request the dispatch of a non periodic thread.

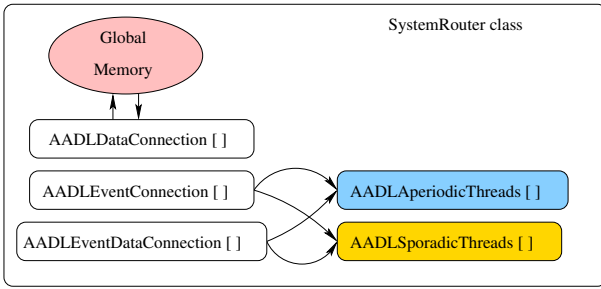


Figure 5: Inside the router.

4.4.2 Threads dispatch and scheduling

The kernel offers three types of threads : *AADLPeriodicThread*, *AADLAperiodicThread* and *AADLSporadicThread*. These three classes extend the *SystemThread* class, that is a modified *NoHeapRealtimeThread*.

RTSJ's scheduling. Most of the scheduling is performed by the RTSJ scheduler : enforcing priority rules, suspending and resuming threads. However, for a thread's execution to match the AADL threads execution model, actions must be taken manually, especially to ensure that dispatch and complete actions are performed correctly at the right timing.

Part of this can be achieved by surrounding the user entry point with the right methods in the thread's *run()* method. For instance, we know that the thread completes when the execution of the user entry point is finished, thus we write the *run()* method as follows:

```

/* run method for a periodic thread : */
public void run () {
    boolean continue = true;
    while(continue){
        userRunnable.entry_point ();
        complete ();
        continue = waitForNextPeriod ();
    }
}

```

In the *complete()* method we write the communication actions that must be performed when the thread completes. We can do this for aperiodic and sporadic threads as well.

Moreover, for periodic, aperiodic and sporadic threads, in order to achieve communication at deadline, we introduce a *OneShotTimer*: the deadline timer and its handler: the *DeadlineMonitor*. When this timer fires, the handler calls a method in the associated thread to send the output that was scheduled to be sent at the deadline, the timer will be restarted when the thread is dispatched again. There is one such timer and handler per thread. But other requirements are *Dispatch_Protocol*-specific:

Periodic threads. Periodic threads are handled by the RTSJ scheduler. Expected time values for period, deadline and WCET are given to the *PeriodicParameters* class. We know that, for periodic threads, dispatch must occur at the begin-

ning of each period. However, as soon as more than one periodic thread is active in the system, no more than one of them can actually be running at the time of the dispatch. Other periodic threads will have their actual execution delayed by the scheduler and thus, they will not be able to request dispatch communication operations on their own at the right time. Therefore, we have to use the *Dispatch_Offset* AADL property to provide off-line scheduling in order to prevent several periodic threads from being dispatched at the same time. This can be easily done by setting the *RelativeTime_start* in the thread's *PeriodicParameters* to the value of the *Dispatch_Offset* property. Eventually, for periodic threads, the dispatch occurs at the same time as the beginning of the execution and, as a consequence, the execution model for periodic threads can be redefined as shown on figure 6.

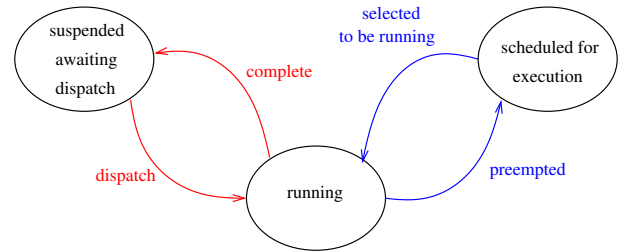


Figure 6: Simplified execution model for periodic threads.

Thus we write the *run()* method as follows:

```

/* complete run method for a periodic thread
: */
public void run () {
    boolean continue = true;
    while(continue) {
        dispatch ();
        userRunnable.entry_point ();
        complete ();
        continue = waitForNextPeriod ();
    }
}

```

Aperiodic threads. Aperiodic threads are the simplest case as their dispatch occurs when an event is received. Thus, the corresponding incoming port can dispatch the thread. If an event has been received while the thread was executing, the thread is dispatched immediately after it has completed (to do this we simply add a test at the end of the *complete()* method to re-release the thread if necessary).

Sporadic threads. As for aperiodic threads, when an event is received, the corresponding incoming port can dispatch the sporadic thread. However, the MIT (minimum inter-arrival time) has to be considered. We introduce another *OneShotTimer*: the MIT timer and its handler: the *MIT-Monitor*. If an event has been received while the thread was executing and has not been ignored or treated as an error, when the thread completes:

- if the MIT timer has fired, the thread is dispatched

immediately,

- otherwise, the thread returns to the *suspended awaiting dispatch* state until the MIT timer fires. Then the thread is dispatched again.

Example : periodic threads. We refine the example given in the listing 6 to specify that we use periodic threads. In this Example, we want to be sure that the thread testThread2 is always executed after testThread1 has completed its execution and produced its output. Thus, by performing off-line scheduling, we give testThread2's *Dispatch_Offset* the value of testThread1's *Deadline*:

```

process testProcess
end testProcess;

process implementation testProcess.impl
  subcomponents
    t1: thread testThread1.impl {
      Dispatch_Protocol => Periodic;
      Period => 100ms;
      Deadline => 40ms;
      Compute_Execution_Time => 10ms..20ms;
      Disptach_Offset => 0ms;
      Priority => 20;
    };
    t2: thread testThread2.impl {
      Dispatch_Protocol => Periodic;
      Period => 200ms;
      Deadline => 150ms;
      Compute_Execution_Time => 10ms..30ms;
      Disptach_Offset => 40ms;
      Priority => 11;
    };
    (...);
  end testProcess.impl;
  (...);

```

We use the constructor for AADL periodic threads defined in the kernel:

```

public AADLPeriodicThread(
  AADLRunnable toRun,
  RelativeTime dispatchOffset,
  RelativeTime period,
  RelativeTime cost,
  RelativeTime deadline,
  int priority);

```

Then, in a *main* class, we can write the process:

```

(...);

/* using the runnables previously created: */
AADLPeriodicThread t1 =
  new AADLPeriodicThread(t1_r,
    new RelativeTime(0),
    new RelativeTime(100),
    new RelativeTime(20),
    new RelativeTime(40),
    10);

AADLPeriodicThread t2 =
  new AADLPeriodicThread(t2_r,
    new RelativeTime(40),
    new RelativeTime(200),

```

```

    new RelativeTime(30),
    new RelativeTime(150),
    11);

/* using the connections previously created:
*/
ConnectionsTable routingTable =
  new ConnectionsTable(dataConnections,
    null, null);

/* create the router */

SystemRouter router =
  new SystemRouter(routingTable);

/* connect the threads to the router: */

t1.connectToEnvironnement(router);
t2.connectToEnvironnement(router);

```

4.4.3 Ports

System ports classes are intended for interfacing with the router and the local memory of the thread (user ports classes).

- *in* system ports act as input buffers to the thread: they receive input at any time but only deliver a part of this input to the user, at dispatch time or start time.
- *out* system data ports act as output buffers to the thread: the output written is not necessarily sent to the environment immediately but only at the right time (at thread's complete or deadline).

4.5 Summary

We have achieved a Java kernel that implements the semantics of a subset of AADL:

- Threads whose behaviour follows a simplified AADL execution model (see figure 2):
 - Periodic, aperiodic and sporadic *Dispatch_Protocols* are supported. Concerning background thread, a trivial solution is possible, using RTSJ's *ProcessingGroupParameters* (server). However, we believe that AADL tools to analyze the system could provide a better solution.
 - thread dispatch and complete states that occur as defined by the AADL standard. This is mostly achieved by using *OneShotTimers*. In the future we intend to use less handlers for deadline and MIT monitoring, using RTSJ 1.1 rich asynchronous events and handlers.
- AADL-type Communication:
 - data, event and event data ports for threads (other components not implemented yet)
 - port connections (immediate, delayed, sampling)
- Separate user and system classes and operations.

This library can be used to program AADL-type applications in Java as well as to generate code from the AADL specification, using the AADL metamodel and model transformation engines available within the TOPCASED framework [12].

5. RELATED WORKS

5.1 The Giotto language

Giotto is a time-triggered architecture description language dedicated to real-time embedded systems. Thus, Giotto models provides abstractions for usual embedded systems components: sensors, processing units, actuators and execution modes. A typical Giotto program is made of a set of port declarations (ports are typed), a set of task declarations, a set of driver declarations, and a set of mode declarations and an initial mode.

5.2 Giotto and AADL

Giotto is similar to AADL as both are ADLs that use execution modes as a base in order to build real-time applications for performance-critical (and embedded) systems. However, some strong differences exist between these two languages:

- while the Giotto execution model is pretty simple as it considers only periodic threads, the AADL execution model is very complicated since aperiodic, sporadic and background threads are also considered,
- the AADL offers a much wider range of abstract components,
- communication in Giotto is entirely synchronous, whereas in AADL communication is partially asynchronous in v1 and mostly asynchronous in v2.
- the mode switching procedure in Giotto (it has not been described in this paper) is more restrictive than in the AADL, thus much simpler.

In other words, while Giotto focuses on the basic concepts of real time applications (periodic threads only, simplified mode switching, less components etc), AADL is more intended to come up to actual needs of real time software developers and thus has a very rich execution model. It might be interesting to find a compromise between these two approaches.

6. CONCLUSIONS

In this paper, we have sketched a mapping from a subset of AADL to RTSJ. We believe that the considered subset is significant with respect to real-time programming for embedded systems. We are also currently working on a TLA [7] formal specification of the AADL execution model. Currently, we have mainly elaborated the communication model and the basic scheduling aspects (wrt AADL v1). It is interesting to remark that working both on a formal model and on an executable model has led us to raise interesting questions to the language designers. We plan to carry on our work with respect to two directions:

- make true experimentations: for the moment we have just made some experiments on top of linux machines, we plan to make experiments on Jamaica VM.
- complete the validation of the mapping. The challenge being to establish that the mapping is in fact a refinement [1, 5], of the AADL execution model.

7. ACKNOWLEDGMENTS

We would like to thank Gérard Padiou and Tanguy Leberre for their RTSJ expertise.

8. REFERENCES

- [1] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
- [2] S. Aerospace. *SAE AS5506 : ARCHITECTURE ANALYSIS and DESIGN LANGUAGE (AADL)*. SAE International, 2004.
- [3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
- [4] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, 2000.
- [5] W.-P. de Roever, K. E. with the assistance of J. Coenen, K.-H. Buth, P. Gardiner, Y. Lakhnech, and F. Stomp. *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press, 1998.
- [6] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.
- [7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [8] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking real time java using java pathfinder. In D. Peled and Y.-K. Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 444–456. Springer, 2005.
- [9] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
- [10] G. Padiou. Asynchronous calls in Java. Private communication. Describes code generation for asynchronous calls with statically allocated memory.
- [11] B. H. Thomas A. Henzinger and C. M. Kirsch. Giotto : A time-triggered language for embedded programming. Technical report, University of California, 2003.
- [12] TOPCASED. Toolkit in open-source for critical applications and systems development. <http://www.topcased.org>.
- [13] S. Vestal. *MetaH User's Manual*. Honeywell Technology Drive, 1998. <http://www.htc.honeywell.com/metah/uguide.pdf>.
- [14] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
- [15] A. Zerzelidis and A. Wellings. Model-based verification of a framework for flexible scheduling in the real-time specification for java. In *JTRES '06: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 20–29, New York, NY, USA, 2006. ACM Press.