Behavioral Modular Description of Fault Tolerant Distributed Systems with AADL Behavioral Annex

Gilles Lasnier, Thomas Robert, Laurent Pautet
Institut TELECOM - TELECOM ParisTech - LTCI
46, rue Barrault, F-75634 Paris CEDEX 13
Paris, France
Email: firstname.lastname@telecom-paristech.fr

Fabrice Kordon
LIP6 - Université Pierre & Marie Curie
4, place Jussieu, 75252 Paris CEDEX 05
Paris, France
Email: fabrice.kordon@lip6.fr

***Abstract*—AADL is an architecture description language intended for model-based engineering of high-integrity distributed systems. The AADL Behavior Annex (AADL-BA) is an extension allowing the refinement of behavioral aspects described through an AADL architectural description. When implementing Distributed Real-time Embedded system (DRE), fault tolerance concerns are integrated by applying replication patterns. We considered a simplified design of the primary backup replication pattern as a running example to analyze the modeling capabilities of AADL and its annex. Our contribution lies in the identification of the drawbacks and benefits of this modeling language for accurate description of the synchronization mechanisms integrated in this example.**

***Index Terms*—aadl; behavior; fault-tolerant; real-time distributed systems.**

I. Introduction

The *Architecture Analysis and Design Language* [1] (AADL) is an international standard intended for model-based engineering of Distributed Real-Time and Embedded (DRE) systems. It aims at modeling DRE systems with deployment, configuration and real-time information, thus allowing code generation. Both functional and non-functional features may be specified with AADL.

For systems requiring strong dependability, DRE applications usually implement both control and data acquisition services. Certification processes consider profiling, verification, test, and fault tolerance as methods to increase the dependability of the whole system. Hardware and software replication are amongst the most popular methods enabling fault tolerance. It boils to synchronize several copies of the application to improve the system overall dependability. A replicated application is a distributed multithreaded application. Thus, its design has to ensure that local synchronizations do not interfere with the fault tolerant protocol. Basically, the use of two distinct synchronization mechanisms should not affect system liveness.

The AADL standard is organized as follows: the core language defines the main semantics of systems. The core semantics definition document is available for years and covers most of the features needed to define a dependable computer system. Some behavior of an AADL specification can be inferred from the described architecture thanks to the AADL runtime model. However, this remains limited.

AADL proposes annexes to extend the core language. Among them, the AADL Behavior Annex (AADL-BA) allows one to associate behavioral description, to describe, in particular, synchronization mechanisms.

As members involved in the elaboration of these annexes, we propose a lookup on AADL-BA to be issued in spring 2010. It provides constructions to define the expected behaviors of system components described with AADL. It relies on an automata-based syntax. However, ensuring the consistency of these automata with regards to the core specification of the system is a difficult issue. It is of interest to use the new features of the standard as soon as possible to improve its capabilities.

A fault-tolerant DRE application is used as arunning example to illustrate AADL-BA modeling capabilities w.r.t. synchronization issues. This application is used to demonstrate AADL-BA efficiency.

Based on that experiment, this paper discusses the difficulties discovered while modeling this system with AADL and AADL-BA. It also proposes some design strategies to describe behavior of components in AADL-BA. In addition, we discuss about some difficulties discovered while modeling this system.

The paper is structured as follow. Section II presents the case study and highlight two modeling challenges raised by such systems. Section III presents an overview of the core language and its semantics through the AADL description of selected components of the case study. Section IV provides the key concept of the annex as well as guidelines for modeling the identified challenges. Finally, section V reports some feedbacks about AADL-BA.

II. The PBR strategy

In this section we present the Primary Backup Replication (PBR) mechanism in the context of DRE systems.

A. Passive Replication strategy for fault tolerance

One of the most efficient way to ensure fault tolerance is redundancy. Fault tolerance mechanisms then often rely on combined hardware and software redundancy to prevent system catastrophic failures. Replication-based mechanisms assume that the application is deployed on several distinct hardware nodes. A fault assumption defines the way errors occur, propagate and trigger failures in the application and the hardware. A synchronization protocol is enforced between the different copies of the application in order to preserve the application function in presence of local software or hardware failures. The design and validation of such mechanisms remain active research topics when hard real time requirements are considered [2], [3], [4].

The Primary Backup Replication is such a mechanism. It is designed to tolerate a bounded number of crashes of the hardware executing the application without paying the cost of concurrent executions of the same application. The copies of the application are synchronized with execution controllers. The role of these controllers is to enforce the replication strategy. The replicas (the copies of the application and their associated controller) is the building block of the PBR strategy as illustrated in figure 1. We summarize the strategy as follows:

1) One of the replica, called the *primary*, executes the application. The controller of this replica manages periodic snapshots of the application execution context and sends them to other replicas.
2) Other replicas are called *backups*; their local copy of the application is not executed. Their controller is waiting for execution context updates sent by the primary. These updates are stored locally. When the primary replica crashes, backups can restart the application in its most recent execution context.
3) As soon as the crash of the primary is detected by a backup, backups synchronize to elect a new primary that restarts the application at the last checkpoint.

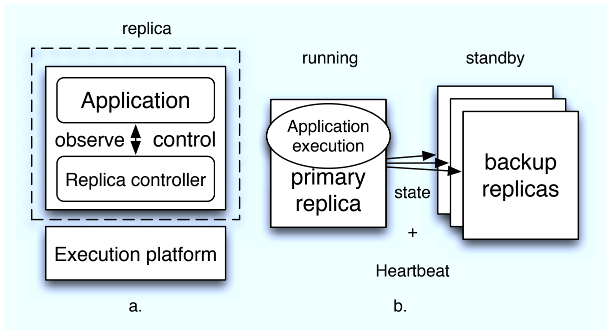

Fig. 1. Replica high-level architecture

We model both primary and backups behaviors. Three points need to be detailed in order to identify the challenges encountered when modeling such a system: detection of the primary crash, election of the new primary, restart of the application on the new primary.

a) Detecting the Primary Crash: The detection of the primary replica crash is implemented by an heartbeat protocol. Thus, ”i am alive” messages are sent periodically from the primary to backups to notify them that the primary is still running. As soon as backups no longer receive such messages, they suspect a crash of the primary. A timeout is used locally on each backup to trigger an election process.

b) Electing the New Primary: When the primary crashes, one or several backups do not receive their heartbeat in time. They broadcast to each backup a notification that the primary crashed. The election process is deterministic: backup IDs are sorted and the next primary is the backup with the smallest ID. A consensus algorithm designed to tolerate the crash fault model is used to ensure that every backup agreed on the identity of the new primary [5]. Once elected, this primary restarts the application as soon as possible.

c) Restarting the Application on the New Primary: The new primary must restart the application from the most recent checkpoint. As said above, checkpoints are periodic snapshots of the application execution context. As soon as a backup restarts the application, it uses this checkpoint to reset the application execution state. The latency introduced by restarting the application on a backup replica can be bounded thanks to the checkpointing period. Once the checkpoint is reloaded, the new primary executes the application in a state that is, in the worst case, few checkpointing periods old. The checkpoints sent by the primary are often used as heartbeat messages as they are sent at a minimal frequency.

B. Behavioral Modeling Challenges

This case study brings some interesting challenges in terms of design and model-driven engineering. We use these challenges to illustrate the features of AADL-BA.

The design of such mechanisms is complex :

- The protocol defined between replicas often rely on consensus algorithms that are rather complex to check. Checking and assessing this protocol is the first step to obtain efficient fault tolerance mechanisms.
- The controller has to be synchronized with the application for checkpointing. When naive atomic snapshots are not allowed, building the model of checkpoint mechanism becomes actually a challenge.
- Eventually, time constants are often used to tune the crash detection services. The values used have an important impact on the efficiency of the algorithms.

1) Inter-replica Protocol: Several methods exist to model and check the protocols defined between replicas. Most of these approaches are extensions or refinements of the ”state machine approach”, [6]. The issue of this seminal work was to handle the complexity of representing all behaviors in a single model. A solution is to consider operational modes to handle separately the behaviors of backups and primary. Then, the issue is to handle carefully mode switches. In [7], AADL has been used to describe a complex replication strategy and check the communications between replicas inside modes and during mode switches. Verifying mode switches highlighted weaknesses in their case study design. Such verification tasks remain at a quite high level of abstraction. This work assumes the fault tolerant protocol to be described by means of synchronizations on communications (e.g. mailbox services).

As said before, the synchronization method between the application and the controller has also to be modeled in order to check the validity of the checkpointing mechanism.

2) Synchronization for Checkpointing: For non trivial application, the state capture cannot be performed atomically without suspending the application for a long period. In most case, synchronized partial checkpoints are preferred when the application is executing several tasks. The synchronization of the partial checkpoints is necessary to obtain a global consistent checkpoint. Usually such a synchronization is implemented via runtime services like mutexes, semaphores and monitors.

First modeling challenge How to model complex thread synchronization protocols that define conditional synchronization between threads ? Is it possible to model distinct types of synchronization mechanisms at the same time (e.g. mailbox-like mechanisms and shared variables) ?

3) Timeouts and time-triggered activity: The heartbeat protocol uses watchdog or timer services to trigger the election of the new primary. A clean specification of interactions between watchdogs (or timers) and threads needs to be provided at least for the backup replica controller.

Second modeling challenge At which level of abstraction time triggered behaviors should be described ? How to describe their interactions with the rest of the system activity ?

The next section provides a brief overview of the AADL architectural description language. Then, we present the architecture of the core components of the PBR replica designed in this language.

III. Designing PBR with AADLv2

AADL [1] is an architecture description language standard managed by the Society of Automotive Engineers (SAE). This section presents a short overview of version 2.0. The software architecture of the main PBR components is specified in AADL to illustrate the standard capabilities.

A. Overview of AADLv2

AADL is a component-based description language for embedded architectures. It focuses on specifying the structure and interactions between the system components. AADL provides notions such as *processes*, *threads*, *data* for software entities and *processors*, *bus* for hardware entities.

AADL allows to define components: their interface, their structure and their properties[1] with respect to functional and non-functional requirements.

1) Main Components: Components in AADL belong to three types of component that are briefly described here.

[1]These are called attributes in other specification languages such as UML

a) Software Components: Data represent data types and may contain other data or subprograms (see shared data). A *subprogram* is an abstraction of procedures in imperative languages. A *thread* is the execution and schedulable unit in AADL. It may contain data and subprograms, as well as *subprogram call sequences*. A *Process* represents a virtual address space to store threads and data.

b) Hardware Components: AADL defines *Execution Platform Components* to describe the underlying hardware properties of a system (that can be provided by both hardware and a middleware layer). A *processor* represents a microprocessor with its scheduler. A *memory* represents a storage space (RAM/ROM or disk). A *bus* represents a hardware communication channel that links execution platform components.

c) System Component: A *System* is an hybrid component used to build hierarchies of software/hardware components.

The description of these components and their semantic implicitly have an impact on the behavioral aspect of components that are constrained by the low-level architectural description. However, refinement of the component behavior can be specified by the use of the AADL Behavior Annex.

2) Component Interactions: Components interaction are defined in a two step process. First, interfaces of components have to be defined to determine the possible interactions. The *features* of a component are *event port*, *event data port*, *data port*, *parameters* and *subprogram accesses*. They represent such interfaces that are often specific to a given type of component. Thus, *ports* provide generic mailbox services for threads. *Parameters* are interfaces of subprogram components. They can be specified *in*, *out* or *in out*.

Second, AADL defines communication mechanisms as shared data. Provides subprogram access are used in data's interface to indicate which components access such data. Connections must be specified between interfaces to show communication links.

3) Component Properties and Modes: AADL components description can be completed by *properties*. The AADL language defines a large set of specific properties to refine component definition. For instance, it is possible to refine synchronization policies between threads, management policies of critical sections on strategies to deploy and configure the system [8].

In addition to properties, AADL provides a support to describe operational modes of a system. Modes provide a nice structure to describe reconfiguration processes of existing components. Nevertheless, the set of components (a configuration) used in the system are statically defined. Connections between components and values of properties can also be mode-specific.

B. Modeling the PBR Architecture with AADLv2

Modeling a DRE system with AADL requires to identify the different roles of components and their hierarchies. In AADL, data and subprograms are located in threads. Threads are also located in processes (providing a memory space shared by all enclosed threads). System components are used to hierarchically structure the system, thus increasing the readability of the specification.

We selected *process* components, *thread*, *subprograms* to model the application and replica controller modules of a replica. The computers used in PBR is represented by a set of *processor* components and a *bus* component.

Data exchange and interaction between components are specified through AADL features as *data*, *event and event data ports* and *connections*. The *system* component allows us describing our complete PBR architecture that contains one primary replica and two backups. Then, *ports* and *data* components can both be used to model the synchronization between the application and the controller. A decision has to be made between these mechanisms.

1) The Replica System Component: The replica module is presented in listing 1. It is a *system* containing two *processes*: the application and its controller. This makes one replica.

```
processor TheCpu end TheCpu;

system Replica end Replica;

system implementation Replica.impl
subcomponents
  Appli     : process Application.impl;
  Rep_Ctrl  : process Replica_Ctrl.impl;
  CPU       : processor TheCpu;
connections
  port Appli.OutA -> Rep_Ctrl.InA;
  port Appli.OutB -> Rep_Ctrl.InB;
  port Rep_Ctrl.InA -> Appli.OutA;
  port Rep_Ctrl.InB -> Appli.OutB;
properties
    Actual_Processor_Binding =>
      reference (CPU) applies to Appli;
    Actual_Processor_Binding =>
      reference (CPU) applies to Rep_Ctrl;
end replica.impl;
```

Listing 1. AADL PBR case study : replica module

Processes are bound to the execution platform component: the CPU *processor*. The *connections* section shows how to connect *in ports* (resp. *out ports*) of the involved *process* components. The *properties* section binds processes to processors using the *Actual_Processor_Binding* property.

2) The Application Process Component: the Application process is presented in listing 2. Section *features* describes its interface: *event ports* for in/out communication to halt/resume the thread execution. Two concurrent threads, ThA and ThB, manage the application context (component `The_Shared_Data`) and take care of variables consistency.

```
process Application
features
  InA  : in  event port;
  InB  : in  event port;
  OutA : out event port;
  OutB : out event port;
end Application;

process implementation Application.impl
subcomponents
  ThA : thread thread_w_state_A;
  ThB : thread thread_w_state_B;
connections
  port InA -> ThA.InA;
  port InB -> ThB.InB;
  port ThA.OutA -> OutA;
  port ThB.OutB -> OutB;
end Application.impl;

thread thread_w_state_A;
 features
  InA   : in event port;
  OutA  : out event port;
  The_Shared_Data :
     requires data access Shared_Data.Impl;
properties
  Dispatch_Protocol        => Periodic;
  Period                   => 500 Ms;
  Compute_Execution_Time   => 0 ms .. 200 ms;
  Deadline                 => 500 Ms;
  Initialize_Entrypoint_Source_Text => "InitSpg";
end thread_w_state_A;
```

Listing 2. AADL PBR case study : Application process

Listing 2 also describes one of these thread interface (`thread_w_state_A`) and provides information: initialization subprogram, type of thread (e.g periodic), period, etc.

3) The_Shared_Data Data Component: Since the application context is manipulated by two threads, we use the AADLv2 dedicated pattern to specify shared *data* components. *Concurrency_Control_Protocol* selects a concurrency management policy supported by the AADL runtime (here, *Priority_Ceiling*). *Provides subprogram access* defines the subprograms to be used to access data that will be required by threads `ThA` and `ThB`. This is depicted in listing 3.

```
data Shared_Data
features
 Update : provides subprogram access Update;
 Read   : provides subprogram access Read;
properties
 Priority => 240;
 Concurrency_Control_Protocol => Priority_Ceiling;
end Shared_Data;

data Shared_Data.Impl
subcomponents
 State      : data;
 UpdateSpg  : subprogram Update;
 ReadSpg    : subprogram Read;
connections
  Cnx1 : subprogram access UpdateSpg -> Update;
  Cnx2 : subprogram access ReadSpg -> Read;
end Shared_Data.Impl;
```

Listing 3. AADL PBR case study : shared data

4) The Replica Controller Process Component: The replica controller process (see listing 4) contains a thread synchronizing actions. Section *connections* shows the links between between this thread and the replica controller process through *ports*.

We focus here on the description of the different execution modes of the process. Properties, components and connection can be mode-specific. The keywords *in modes* allow to specify the mode in which the component is involved.

```
process Replica_Controller
features
  InA   : in event port;
  InB   : in event port;
```

```
    OutA : out event port;
    OutB : out event port;
    SnapshotRcv : in event data State;
    SnapshotSnd : out event data State;
    IsPrimary   : in event port;
    IsBackup    : in event port;
    IsElection  : in event port;
end Replica_Controller;

process implementation Replica_Controller.impl
subcomponents
    ThA : thread thread_snap_sync
            in modes (Primary, Election, Backup);
connections
    port InA -> ThA.InA in modes (Primary);
    port InB -> ThA.InB in modes (Primary);
    port ThA.OutA -> OutA in modes (Primary);
    port ThA.OutB -> OutB in modes (Primary);
    port ThA.SnapshotSnd -> SnapshotSnd
                              in modes (Primary);
    port SnapshotRcv -> ThA.SnapshotRcv
                              in modes (Backup);
modes
  -- modes
  Primary   : initial mode;
  Backup    :         mode;
  Election  :         mode;

  -- transitions
  Backup    -[IsElection]-> Election;
  Election  -[IsBackup]->   Backup;
  Election  -[IsPrimary]->  Primary;
end Replica_Controller.impl;
```

Listing 4. AADL PBR case study : Replica controller process

The operational modes of replicas described in II are *primary*, *backup* and *election*. Mode transitions are explicitly defined by the syntax $mode\_initial - [event\_triggered] - > mode\_final$.

Mode switch is synchronized with events occurring from ports. For example, when the replica controller in *backup* mode receives a *IsElection* event, then the process switches to the *election* mode.

C. Checkpoint synchronization and watchdogs

AADL models of the application and the replica controller have been presented but there is still no description of the checkpointing and watchdog services.

The checkpointing service has to enforce a rendez-vous. It is has to block threads until all participants reached the rendez-vous. Then the controller saves the copy of the application state, and releases application thread executions.

There are two reasons for suspending a thread: when it is waiting for a dispatch trigger, or when is waiting for a shared resource. In the first case, it is easy to control how the thread is wake-up by sending an event on one of its ports. A thread will reach such dispatch state at the end of a call sequence. These particular states can be used to set up rendez-vous between threads.

In the second case, a Concurrency Control Protocol defines how critical sections associated to shared data should be protected. One of the proposed protocol uses subprograms implementing the usual lock and unlock primitives (mutexes) to enforce mutual exclusion. These primitives can be used to program more complex synchronization services. So, synchronizations can be either defined at the thread or subprogram level but through different mechanisms. Next section will highlight the fact that describing the implementation ”from scratch” with rendez-vous is easier at the thread level.

The heartbeat watchdogs are often implemented with software timers. No timer services are directly available in AADL. Nevertheless, the concept of dispatch on timeouts can be found in the AADL-BA. We show in next section how to define a watchdog as an additional dispatch condition for the replica controller thread component.

IV. AADL Behavior Specifications

The AADL Behavior Annex provides an extension to specify the behavior attached to AADL components. The intend of this annex is to refine the implicit behavior specified in the core language. Thus, it is possible to attach a description called *behavioral_specification* to each AADL component using AADL *annex_subclauses*.

The AADL-BA defines several languages. A state/transition automaton describes component behavior. A dispatch condition language refines thread dispatch behavior. An interaction operations language specifies component interactions as communications through ports, parameters, subprogram calls, etc. The action language combines basic control structures (loop and test) with subprogram calls and port accesses. Then, behavioral actions are attached to transitions of the component automaton. It can be seen as a kind of abstract code snippet bound to transitions. It is a major extension of the expressiveness of the AADL core language.

Finally, an expression language provides logical, relational and arithmetic expressions to manipulate variables.

In this section, we do not detail the expression language which syntax is very close to the one provided by Ada.

A. The Behavior Specification

A *behavior_specification* is expressed as a state transition automaton with guards and actions. Guards and actions uses variables to manipulate data.

The automaton is used to specify the sequential execution behavior of *subprogram* and dispatch protocol. Input and output behavior of AADL *threads* or *devices* can also be expressed by an automaton. Finally, it can also specify the dynamic behavior of a *process* or a *system*.

Local variables (non-persistent) can be used to save intermediate results. State variables specified by the keyword *persistent* or referencing an AADL data component can be used to reduce the size of the state automaton by keeping track of counts for instance.

A behavior automaton starts from an *initial* state and terminates in a *final* state. *Complete* state represents a suspend/resume state out of which threads and devices are dispatched [9]. Remaining states are called execution state and represents intermediate state of the automaton.

A transition represents a change from the current source state to a destination state. A transition is activated when its dispatch or execute condition is evaluated to true. Then the attached action is executed.

Dispatch condition affect the execution of a thread based on external triggers. Execute condition model behavior within an execution sequence of a thread, subprogram or other component. They are based on input values from ports, shared data, parameters, and behavior variable value [9].

1) Subprogram Behavior Specification: The initial state of the subprogram behavior automaton represents the starting point of a call. The final state represents the completion of a call. The automaton describes the execution behavior of a subprogram with one or more return points [9]. Its has one or more intermediate execution states but cannot contain a complete state.

2) Thread and Device Behavior Specifications: The behavior automaton of thread or device components describes: one initial state representing the state before initialization actions; one or more complete state representing halt/resume state; zero or more intermediate execution state and one final state representing finalization completed by thread or device completes.

The behavior of a thread dispatch is a dispatch condition evaluates to true and then the transition (outgoing of a complete state) is taken. Action associated to the transition are performed. Periodic dispatches are implicit. Sporadic dispatches can be triggered by the arrival of event, data, event data on ports or the call to provides subprogram access features. AADL-BA describes timeout for thread dispatch with the use of *on dispatch timeout* as dispatch condition. Timeout is a dispatch trigger condition raised after the specified amount of time since the last dispatch [9].

The thread behavior automaton are consistent with the thread states/actions described in the core AADL language [1]. We identify the same states named differently. The initial state corresponds to the halted state. The complete state represents the awaiting dispatch state. Final state represents the stopped thread state.

3) Other Component Behavior Specifications: The automaton of other components (process, processor, etc) starts with one initial state representing the state before initialization, one ore more complete states and one final state representing the state after finalization [9].

4) Component Interaction Behavior Specifications: As said above, AADL threads interact through shared data, connected ports and subprogram calls. AADL-BA provides mechanisms to model the behavior of *event data ports*, *data ports* or *event ports*. Thus, behaviors and policies governing *data port* and *event data port* queues (e.g dequeue protocol) can be specified.

Frozen ports mechanism can be used to ensure availability of received data on a port after thread dispatch occurs. Send and receive outputs -described by shortcut operators- through ports can be specified.

The standard defines several ways to model access to shared data subcomponents (see next subsection).

Finally, interaction between components using subprograms can also be specified by subprogram access, using the syntax *Mysubrogram!* or *Mysubrogram!(param1,...paramN)*. This call to subprogram access is defined in the actions attached to transitions.

B. Chekpointing Implementation

1) Shared Data Semantics: The standard defines three ways to model critical section in order to access shared data.

a) The smaller action block: A smaller action block encapsulates the shared data subcomponent reference with the use of '{' character and '}' character as delimiters. The annex specify that if an action block contains references to several shared data subcomponents, then resource locking will be done in the same order as the occurrence of the references to the shared data subcomponents. Resource unlocking will be done in the reverse order [9].

b) Provides subprogram access: Appropriate provides subprogram access of the corresponding shared data component can be called in actions associated to transitions. These provides subprogram access must be explicitly defined to implement the concurrency control protocol which coordinates accesses to shared data.

c) Get_resource and release_resource runtime services: *Get_resource* and *release_resource* runtime services specified in the runtime support of the AADLv2 standard [1] can be manually inserted in actions attached to transitions.

According to the AADLv2 standard, the user can also provide specific implementations of *get_resource* and *release_resource* at execution platform level.

The small block action is easy to use. It allows implicit and automatic placement of *get_resource* and *release_resource* services by the use of '{', '}'. However concerning modeling complex critical section as multiple data shared and multiple lock/unlock, the semantic defined is not precise enough.

The use of *provides subprogram access* implies to check all subprogram access and subprogram implementation to avoid run-time violation. So, systems analysis becomes more complex.

The use of *get_resource* and *release_resource* runtime services is very expressive to model access to critical section. The user can specify easily with subprogram calls where is the begin and the end of the section for simple synchronization schemes.

In the case if the user provides specific implementations of the *get_resource* and *release_resource* subprograms, then he has to insert by hand the calls to these subprograms in AADL-BA specification. Nevertheless, the standard does not provide any guidelines to specify the behavior of these subprograms.

Thus, we prefer to use AADL ports to model the synchronization barrier used in the check-pointing mechanism.

2) Modeling Challenge and Complexity: Synchronization for checkpointing requires to specify a complex synchronization mechanism between threads. We have mentioned in section III that the core AADL allows the description of synchronization mechanisms between shared resources (data). Subprogram accesses or events sent on connected ports are also involved to model these check-pointing mechanisms.

The listing 5 depicts the behavior automaton of thread thread_w_state_A contained in the application process. The thread behavior automaton has one initial state *si*, two complete states *s1*, *s2* and one final state *sf*.

```
annex behavior_specification {**
  states
    si: initial state;
    s1, s2: complete state;
    sf: final state;
  transitions
    si -[]-> s1 { InitSpg! };
    s1 -[on dispatch]-> s2 { Computation1!
                              OutA! };
    s2 -[on dispatch InA]-> s1 { Computation2! };
**};
```

Listing 5. AADL-BA PBR case study : thread behavior autamaton

When the transition *si* to *s1* triggers, the *InitSpg* subprogram specified in the actions section ('{'...'};' section) is called to initialize the thread.

Complete states *s1* and *s2* are waiting states used when the thread waits for dispatch (execution). At the first dispatch, the transition *s1* to *s2* triggers and the actions attached to the transition are executed. Thus the subprogram *Computation1* is invoked and the *OutA!* produces an event on the event port *OutA*. This signal corresponds to a situation in which the thread has completed its works and waits in state *s2* for a signal *InA* to resume.

The synchronization protocol is described through transitions triggered and actions executed between *s1* and *s2*. *s2* is the rendez-vous state. *OutA* event is the notification that a thread reach the rendez-vous. *InA* is the event received when checkpoint is completed.

C. Heartbeats Protocol Implementation

In this subsection, we describe how to use AADL-BA to model the behavior of the heartbeats protocol using AADL-BA timeout for backup replicas.

The heartbeat protocol used in the PBR architecture relies on a timeout that is triggered once the specified amount of time since the last dispatch has expired. The timeout value is given by the *Period* property of the thread.

Listing 6 depicts the behavior automaton of the thread contained in the replica controller process (backup replicas) including timeout. We give a simple description for better understanding. The *states* section declares *si* as initial state (before thread initialization), *s1* as complete state (for dispatch) and *sf* as final state.

When the thread starts, initialization is due by invoking the *InitSpg* subprogram. The transition starts from the initial state *si* and stops in the *s1* complete state. When the thread receives a InA event, the condition on dispatch InA is true, the transition between *s1* to itself triggers.

Thus, *ReceiveSnapshot* and *StoreSnapshot* subprograms are called (see actions section attached to the transition).

```
annex behavior_specification {**
  states
    si: initial state;
    s1: complete state;
    sf: final state;
  transitions
    si -[]-> s1 { InitSpg! };

    s1 -[on dispatch InA]-> s1 { ReceiveSnapshot!;
                                  StoreSnapshot!;};

    s1 -[on dispatch timeout]-> sf { OutElection! };
**};
```

Listing 6. AADL-BA PBR case study : timeout

We focus now on the dispatch timeout. According to the semantics of the AADL-BA the timeout occurs when the period of the thread expired. The thread is in state *s1* when the timeout triggers. If the backup replica controller process does not receive the snapshot (i.e *InA*) then the timeout triggers. The transition between *s1* and *sf* with the *on dispatch timeout* condition occurs. The performed action (*OutElection!*) is the emission of an event on the OutElection out event port. This event is transmitted to other backup replica controller process. Then the reception of this event triggers the mode change into replica controller process.

V. Discussions

PBR has been selected because of it complex behavior even when no crash occurs. This section reports comments and advices for people already using the core language and interested in using the behavioral annex.

A. Synchronization on Port or Resources

The core language explains how threads can be synchronized with dispatched events. A reception on a port can be used to release an execution. AADL-BA provides an automata based representation to describe the call sequences executed by threads and the state on which they are waiting for dispatch. This state base representation clearly identifies states where the application can be blocked. This annex strongly improves the readability of thread behaviors in a specification. However, it does not help to decide when shared resources should be preferred to complex synchronization protocols and vice-versa.

Thus, design strategies are required to understand consequences of describing synchronization patterns inside threads behavior. If we want to define reusable synchronization services, then ports and dispatch conditions are not satisfactory. Thread behavioral models are not modular enough: all dispatch conditions have to be defined in the same automata. Only subprograms models can be nested and reused in other subprograms. A rendez-vous protocol defined in behavioral models of subprograms can only rely on the semantics of the locking primitives Get_Resource and Release_Resource.

B. Get_Resource, Release_Resource Semantics

AADL proposes to infer critical regions from the architectural description, or from AADL-BA action blocks. The standard let also to designers the opportunity to insert explicit calls to Get/Release to suspend/resume threads. If Get/Release primitives are not necessarily paired in subprogram descriptions, then the rendez vous could be implemented. Note that POSIX implementations [10] consider this construct as erroneous. If designers follow this strategy, then the standard explains that it may not be compatible with the AADL semantics of critical sections protection. Then, the whole synchronization policy has to be checked again.

Despite this modeling issue, AADL-BA provides an interesting additional strategy to define critical regions: smaller action blocks. Smaller action blocks define boundaries of critical regions for shared data. Action blocks can be seen as simplified programs that refine the notion of call sequences from the core language. Such critical regions have the same problem than the ones defined by require/provide data access. If critical regions of distinct shared data overlap, then Get/Release have to be inserted in the correct order to avoid deadlocks.

C. *Timeout on Thread Dispatch*

In AADL, a timeout allows triggering call sequences when a dispatch is not performed in time. The deadline is defined respectively to the previous dispatch of the thread. This mechanism has been used to implement successfully a watchdog for the heartbeat protocol. The timeout dispatch condition in the annex seems dedicated to avoid threads to be blocked waiting for an event issued from their ports.

Watchdogs are also used to detect and interrupt programs exceeding their worst case execution time. Timeouts do not provide such a semantics. During the execution of an action block, even if a timeout condition is met, threads have to complete the execution of the transition prior to trigger the timeout reaction. Such a semantics is safer than interrupting subprograms that may hold locks.

VI. Conclusion

Due to the recent publication of AADLv2 and AADL-BA (this annex will soon be in informal ballot), it is of interest to check if engineers can use both AADLv2 and AADL-BA safely (e.g. in a consistent way). For that purpose, we model the PBR (Primary Backup Replication) strategy that is a typical fault-tolerant mechanisms for Distributed Real-Time and Embedded (DRE) systems.

Unsurprisingly, we successfully modeled the PBR architecture. However, it was more difficult for the behavioral part. For instance, we had to relax some of our initial requirements to reach our objectives. The main difficulty resides in the design of complex synchronization mechanisms such as await dispatch or mutual exclusion that are commonly required in distributed system design.

There seems to have two approaches driving the definition of AADL-BA. The first one proposes very strict semantics rules that are consistency with the core language at the architectural level. The second one seems to significantly relax the consistency rules (lock/unlock, timeout). The behavioral part of a component may be incompatible with the initial semantical expectations of an AADL component.

Of course, tools may help designers to explore their design from the behavioral description as it is now the case for the architectural part. However, in a MDE-based approach, it may be unsafe to have semantical rupture when engineers come to define precisely the behavior of the system.

This paper is a contribution to outline such difficulties and serve as a basis to enhance the standard through discussion in the standardization committee.

References

[1] SAE, *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008.

[2] H. Zou and F. Jahanian, "Real-time primary-backup (RTPB) replication with temporal consistency guarantees," in *18th Int. Conference on Distributed Computing Systems (18th ICDCS'98)*. Amsterdam, The Netherlands: IEEE, May 1998.

[3] P. M. Melliar-Smith and L. E. Moser, "Progress in real-time fault tolerance," in *SRDS'2004*. IEEE Computer Society, 2004, pp. 109–111.

[4] T. Ayav, P. Fradet, and A. Girault, "Implementing fault-tolerance in real-time programs by automatic program transformations," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 4, pp. 45:1–43, 2008.

[5] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *JACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980. [Online]. Available: http://research.microsoft.com/users/lamport/pubs/reaching.pdf

[6] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv*, vol. 22, no. 4, pp. 299–319, 1990.

[7] D. de Niz and P. H. Feiler, "Verification of replication architectures in AADL," in *ICECCS*. IEEE Computer Society, 2009, pp. 365–370. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2009.18

[8] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications," in *Reliable Software Technologies'09 - Ada Europe*, Brest, France, jun 2009.

[9] SAE, *Annex X Behavior Annex (AS5506-X draft-2.11)*, September 2009.

[10] The Open Group, "The Open Group Base Specifications Issue 6 IEEE Std 1003.1," http://www.opengroup.org/onlinepubs/000095399/functions/pthread_mutex_unlock.html, 2004.