Architectural and Behavioral Modeling with AADL for Fault Tolerant Embedded Systems

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

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

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

***Keywords*-aadl; behavior; fault-tolerant; distributed systems.**

I. Introduction

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

For systems requiring strong dependability, embedded software applications usually implement both control and data acquisition services. Certification processes consider analysis, verification, test methods, and fault tolerance as mechanisms to increase the dependability of the whole system. Real-time and fault-tolerance requirements are tightly related as they both require control on the software execution flow. The verification and code generation services enabled by AADL improve the system overall dependability in a MDE-based approach.

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. The AADL Behavior Annex (AADL-BA) allows one to refine and attach additional behavioral information useful for a finer systems analysis.

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. To do so, we consider a running example of a non trivial DRE architecture that we describe with AADL: an application integrated to a passive replication architecture to be tolerant to crashes.

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

The paper is structured as follow. Section II presents the case study and highlight two modeling challenges raised by such systems. Section III presents the AADL description of selected components of the PBR case study. Finally, Section IV provides the key concept of the annex as well as guidelines for modeling the identified challenges.

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

In most safety-critical systems, fault tolerance is implemented by duplicating the application on separate hardware platforms. *Replication-based* fault tolerance architectures necessitate a synchronization protocol between application copies. This protocol is in charge to detect theirs failures and ensure the fault tolerant application continue to deliver its service. The Primary Backup Replication mechanism presented in this paper is an adaptation of the fault tolerance mechanism described in [2]. The synchronization logic is distributed in each site of the distributed architecture of the mechanism in components called controller). The replicas (the copies of the application and their associated controller) are the building blocks of the PBR strategy. The controllers enforce the following behavior between the replicas:

1) One distinguished replica, called the *primary*, executes the application. The controller of this replica performs periodic snapshots of the application execution context, and broadcasts them to backups.
2) Other replicas do not execute their copy of the application, They are called *backups* and store snapshots of the application execution context sent by the primary.
3) When the crash of the primary is detected by a backup, backups start the election of a new primary that restarts the application thanks to the last snapshot received.

Modeling such an architecture relies on three key functions: the crash detection mechanism, the election protocol for a new primary, and the reconfiguration of the elected backup to restart the application.

B. Behavioral Modeling Challenges

This subsection provides details on the challenges that represents each points identified as a key function of the replication mechanism.

a) The crash detection mechanism: Replica crash detection 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 correctly. As soon as backups no longer receive such messages, they suspect a crash of the primary. 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.

b) Primary election protocol: When the crash of the primary is detect by backups, they start an election protocol to determine which backup will be the new primary. We preferred a deterministic protocol: 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 [3]. Once elected, this primary restarts the application.

c) Checkpoints capture and reload: Snapshots of the application execution context cannot be performed atomically for the whole application. In this context, we use a synchronized checkpoint mechanism. Application threads execution context are saved separately and combined to define the application global state. Each thread needs to wait on a synchronization barrier in order to capture a consistent global state. It often requires monitors or condition variable like services.

In [4], AADL has already been used to describe the reconfiguration protocol used on replicas to tolerate faults. In this paper, we plan to complete this approach showing how the behavioral annex can describe the behavior of threads implementing the replica controller.

III. Designing PBR with AADLv2

The AADL standard [1] is managed by the Society of Automotive Engineers (SAE). for lack of space, we do not remind the new features of AADL version 2.0 that are presented in [5]. This section illustrate the standard capabilities with the description of the software architecture of PBR components.

A. 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*, *thread*, *subprograms* components 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 *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.

```
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. PBR case study in AADL: replica module

Processes are bound to the CPU *processor*. The *connections* section shows how to connect *in/out ports* of the involved *process* components. The *Actual_Processor_Binding* property binds processes to processors.

```
process Application
features
  InA  : in  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 ThB.OutB -> OutB;
  ...
end Application.impl;

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

Listing 2. pbr case study in AADL: application process

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.

Listing 2 also describes one of these thread interface (`thread_w_state_A`) and its properties (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. PBR case study in AADL: shared data

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

```
process Replica_Controller
features
  SshotRcv : in event data State;
  SshotSnd : 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 ThA.SshotSnd -> SshotSnd in modes (Primary);
  port SshotRcv -> ThA.SshotRcv in modes (Backup);
  ...
modes
 Primary  : initial mode;  -- modes
 Backup   :         mode;
 Election :         mode;

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

Listing 4. PBR case study in AADL: Replica controller process

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.

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

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

B. Checkpoint synchronization and watchdogs

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.

The two reasons for suspending a thread are when it is waiting for a dispatch trigger, or 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 assciated 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 highlights 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 is an extension to specify the behavior attached to AADL components. It intends to refine the implicit behavior specified in the core of the language. Thus, it is possible to attach a *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. A behavior action language describes actions to be processed when transition triggers. 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 use variables to manipulate data.

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

Local variables (non-persistent) are used to save intermediate results. State variables referencing an AADL data component or *persistent* 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 [6]. 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 models 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 [6].

1) Subprogram Behavior Specification: The initial state 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 [6]. Its has one or more intermediate execution states but no complete state.

2) Thread and Device Behavior Specifications: The behavior automaton of thread or device 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.

The behavior of a thread dispatch is a dispatch condition evaluates to true. Then, the thread dispatches and transition (outgoing of a complete state) is taken. Actions associated to the transition is 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 [6].

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 [6].

4) Component Interaction Behavior Specifications: AADL threads interact through shared data, connected ports and subprogram calls. AADL-BA provides mechanisms to model the behavior of *event data*, *data* or *event* ports. Thus, behaviors and policies governing *data* and *event data* ports 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 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 supprograms can be specified by 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 '{' and '}' characters as delimiters. If an action block contains references to several shared data subcomponents, then resource locking (resp. unlocking) will be done in the same (resp. reverse) order as the occurrence of the references to the shared data subcomponents [6].

b) Provides subprogram access: Appropriate provides subprogram access of the corresponding shared data component can be called in actions associated to transitions. They 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 implementation to avoid run-time violation. So, systems analysis becomes more complex.

The use of *get_resource* and *release_resource* run-time 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.

However the use of multiple critical section and/or multiple shared data is not trivial to model. Subprogram behavior automaton without complete state reflected the fact that a subprogram can not be blocked. So, if the user specifies it owns *get_resource* and *release_resource* implementation then it is not possible to describe the subprogram behavior. This is a problem for the system analysis.

Finally, according to different semantic of the annex, we find that modeling a critical section is a complex problem. So, in our case we choose to use the AADL port and its semantic to model the checkpointing 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. PBR case study in AADL-BA: thread behavior autamaton

When the transition *si* to *s1* triggers, the *InitSpg* subprogram specified in the actions section ('{'...'};') initialized 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 releases the thread which has completed its works and waits in *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

This subsection describes 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. PBR case study in AADL-BA: 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. Conclusion

Due to the recent publication of AADLv2 and AADL-BA, 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 Primary Backup Replication strategy (PBR) that is a typical fault-tolerant mechanisms for Distributed Real-Time and Embedded systems.

We successfully modeled the PBR architecture but the point was to detail the behavior of its building blocks, *i.e.* the replica controllers. Both thread and subprogram behavioral models should be accurate enough to describe complex synchronization scenarios. We detailed the PBR checkpointing mechanism through thread and subprogram models. Yet, we identified potential issues in subprogram behavioral models. We plan to propose errata to clarify the behavior of the subprogram synchronization primitives.

References

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

[2] H. Zou and F. Jahanian, “Real-time primary-backup replication with temporal consistency guarantees,” in *18th Int. Conf. on Dist. Computing Systems (18th ICDCS’98)*. The Netherlands: IEEE, May 1998.

[3] R. S. M. Pease and L. Lamport, “Reaching agreement in the presence of faults,” *JACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[4] D. de Niz and P. H. Feiler, “Verification of replication architectures in AADL,” in *ICECCS*. IEEE Computer Society, 2009, pp. 365–370.

[5] 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.

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