A Model-Based Transformation Process to Validate and Implement High-Integrity Systems

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

Jérôme Hugues
ISAE - Toulouse University
1, place Emile Blouin, 31056 Toulouse, France
Email: jerome.hugues@isae.fr

***Abstract*—Despite numerous advances, building High-Integrity Embedded systems remains a complex task. They come with strong requirements to ensure safety, schedulability or security properties; one needs to combine multiple analysis to validate each of them. Model-Based Engineering is an accepted solution to address such complexity: analytical models are derived from an abstraction of the system to be built. Yet, ensuring that all abstractions are semantically consistent, remains an issue, e.g. when performing model checking for assessing safety, and then for schedulability using timed automata, and then when generating code. Complexity stems from the high-level view of the model compared to the low-level mechanisms used. In this paper, we present our approach based on AADL and its behavioral annex to refine iteratively an architecture description. Both application and runtime components are transformed into basic AADL constructs which have a strict counterpart in classical programming languages or patterns for verification. We detail the benefits of this process to enhance analysis and code generation. This work has been integrated to the AADL-tool support OSATE2.**

***Index Terms*—model-to-model; aadl; behavior; transformation; middleware; distributed systems.**

I. Introduction

High-Integrity (HI) Embedded systems increasingly rely on software to perform critical functions. These systems are often both security- and safety-critical in that their failure could result in the failure of the mission, or great damage. This particular class of systems comes with strong requirements to ensure safety, reliability and security properties.

Computation models and architectural profiles such as the Ravenscar Profile [1] have been developed to ensure safety. They define several restrictions of concurrency constructs to allow static analysis (schedulability, model checking, etc) of the application. It limits the complexity and avoids non deterministic constructs in HI-DRE system execution support.

Similarly, improved techniques for the production of the software components for HI-DRE systems have been developed including Model-Driven Engineering (MDE) and formal methods. The MDE techniques and tools can be used to specify, analyze, optimize, synthesize, validate and deploy application and middleware components [2]. The MDE approach defines domain-specific modelling languages (DSMLs) as AADL [3] and generative technologies to provide "correct-by-construction" components. At the modelling level, the application of the restrictions defined by profiles such as the Ravenscar Profile reinforces this notion of component "correct-by-construction" [4].

Nevertheless, model-based development introduces new complexities when composing, deploying or analysing DRE systems [2]. Usually, the same high-level model is often used by different tools. Each tool performs an abstraction of the high-level model, mapping constructs to simpler ones prior to assess one particular aspect of the system. We note that 1) these mappings may share some commonalities, and 2) consistency between these abstractions is hard to validate.

In this paper, we propose to make visible this expansion phase at model-level through iterative model-based transformations (MBT): an AADL architectural description is refined to make explicit all runtime low-level artifacts as a refined AADL model; and then to use this view as a common input for both analysis and code generation.

This approach aims at eliminating high-level abstractions in components to obtain a model close to the system implantation, while preserving analysis capabilities of the model. The benefits of this approach allow us to take into account the whole system (application and middleware) in a common model so as to reduce semantic gap between analysis and implementation and to simplify the code generation process.

The reminder of this paper is organized as follows: Section II presents the motivations of our approach; Section III shows a brief overview of AADL and its behavioral annex through an aerospace case study; Section IV gives the building blocks required by our MBT process; Section V describes the different steps of our process; and its integration in the AADL-tool support OSATE2 using ATL and EMF technologies; and Section VI presents concluding remarks and our future works.

II. Motivations

A. MDE Approach to Implement HI-DRE Systems

Model-based HI-DRE systems development process has been defined to improve the development of HI applications in terms of complexity, time and costs. The benefits of this method are: off-line analysis performed earlier in the system's life-cycle, rapid prototyping of the system, automation of analysis and code generation steps.

Several DSMLs have been developed to design DRE systems such as AUTOSAR [5], AADL [3], UML/Marte [6].

They define concepts and patterns to describe the architecture components involved in the application, and their behavior.

These modelling languages introduce high-level abstractions: components, interfaces, interaction protocol and non-functional properties to configure them. These elements are then used to perform numerous analysis such as schedulability, memory, flow analysis to ensure security, etc.

From this architectural description, it is possible to generate architectural skeletons, like the glue code to integrate user components, some middleware components [7], or even the functional code itself from its behavioral description. The information extracted from the system properties specified in the model is used to configure and deploy these components.

B. MDE Implied Complexity

MDE aims at reducing some steps in the design and implementation process, but we note it adds new levels of complexity associated to the analysis, configuration and deployment of components performed by different tools. Each tool performs some refactoring of the model prior to extract relevant features, e.g. to resolve refinements or link model elements. Such refactoring may result in contradictory abstractions, reducing the value of the analysis.

The figure 1 illustrates the analysis made in the HI-DRE development process. Off-line analysis are performed on the system model, e.g. to enforce modeling patterns. Static analysis are performed on an analytical model corresponding to the system implementation.

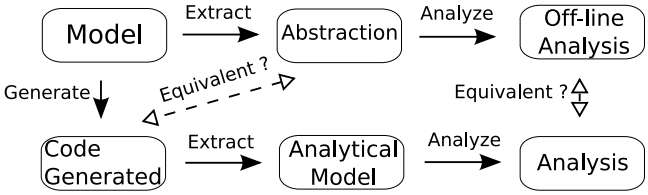

Fig. 1: Analysis on HI-DRE System Development Process

Usually, one assumes all intermediate models used for analysis faithfully reflect both initial models and generated code. However, it is difficult to relate them since they are based on different notations, semantics and levels of abstraction.

The first model describes components with strong semantics and introduce high-level abstractions of components behavior. These high-level abstractions require several transformations to produce a representation in a programming language. These successive transformations are explained by the impossibility to map directly high-level components to programming language structures. Similar concerns arise when one maps the initial model into a representation suitable to perform model checking or simulation.

The second (analytical) model describes the architecture and the behavior of the whole system (application + middleware). The execution support (middleware) is integrated later, at the implementation level, and is therefore not taken into account at the modelling level. Thus, the middleware is considered as a "black box" in several off-line analysis. Yet, it greatly impacts schedulability or resource dimensioning analysis.

C. Discussion

In previous subsections we have pointed out issues introduced by the MDE approach to design HI-DRE systems. The differences between the system model, analysis models, and the generated code introduce a semantic gap, reducing the confidence in the whole process. Differences are due to :

1) the different levels of abstraction between the initial model, intermediate analysis models, and the generated code (final model of the process), and
2) the resources of the execution support (middleware) not taken into account in relevant analysis stage.

To resolve these issues we propose a new approach (see figure 2) to analyze and then generate HI-DRE systems. To reduce the semantic gaps between the system model and the generated code our solution is to refine the system model iteratively with model-based transformations (MBT). We can find such a refinement process in approaches like CompCert [8].

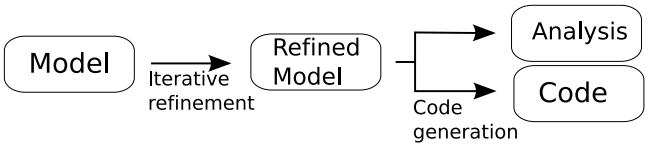

Fig. 2: Approach to implement HI-DRE System

These successive refinements allow us to assess equivalence between the intermediate models and to make explicit transformations often hidden in the model-transformation toolchain. The integration of the middleware components is performed during these refinements and thus allows to consider the whole system (application + middleware) in off-line analysis.

We chose to develop our approach using the DSML AADL and its behavioral annex AADL-BA [9]. The major advantages of AADL is to avoid the introduction of additional languages to specify the intermediate models, while enabling precise specification of components' behavior.

III. Overview of AADL and AADL-BA

AADL [3] is an architecture description language standard managed by the Society of Automotive Engineers (SAE). We have already showed the new features of the AADLv2 [7]. We give a short introduction of AADL using the *MPC* case-study and an overview of the AADL behavior annex.

A. The MPC Case-study Architecture

The figure 3 shows the AADL graphical representation of the Multi-Platform Cooperation (*MPC*) platform. The system holds three spacecrafts (nodes) with different roles. SC_1 is a leader spacecraft that contains a periodic thread which sends its position to SC_2 and SC_3 which are follower spacecrafts. They receive the position sent by SC_1 with a sporadic thread (`Receiver_Thread`), update their own position and store it in a local protected object (AADL shared data). A second thread

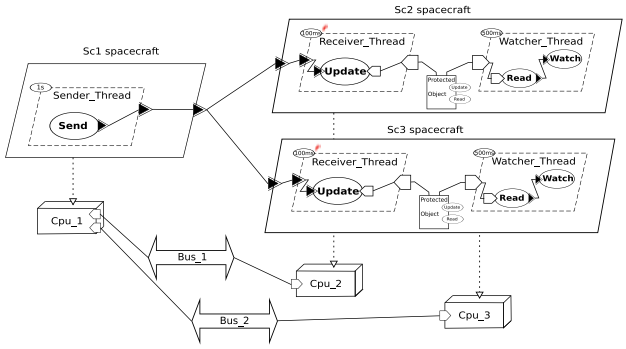

Fig. 3: Architecture view of the MPC case study

in these two spacecrafts periodically reads the position value from the local protected object, and "watches and reports" all elements at that position (e.g. earth observation, etc).

This model gathers typical elements from HI-DRE systems, with a set of periodic tasks devoted to the processing of incoming orders (`Watcher_Thread`), buffers to store these orders (*Protected Object*) and sporadic threads to exchange data (`Receiver_Thread`). These entities work at different rates specified by AADL properties, and should all respect their deadlines so that the `Watcher_Thread` can process all observation orders in due time.

Deployment and configuration are described through the specification of hardware components and the binding of software components to hardware in the same view. In MPC, each node is bound to a specific CPU and communication between nodes are bound to different buses.

B. Overview of the AADL Behavior Annex

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 and replaces property-based interpretation by concrete behavior. Thus, it is possible to attach a *behavioral_specification* to each AADL component using AADL *annex_subclauses*.

1) The Behavior Specification: A *behavior_specification* is a state transition automaton with guards and actions. Guards and actions use variables to manipulate data. Local variables (non-persistent) are used to save intermediate results. State variables can reference an AADL data component.

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]. Execution states represent intermediate states 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. A dispatch condition affects the execution of a thread based on external triggers. An execution condition models the behavior within an execution sequence of a thread, subprogram or other components. They are based on input values from ports, shared data, parameters, and behavior variable value.

When a dispatch condition is evaluated to true, the thread dispatches and transition (outgoing of a complete state) is taken. Actions associated to the transition are executed sequentially. Periodic dispatches are time-driven. Sporadic dispatches can be triggered by the arrival of event or data on ports or the call to provide subprogram access.

2) 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. Send and receive outputs through ports can also be specified.

The standard defines three ways to explicitly model critical section to access shared data. '{' and '}' characters defines a smaller action block which encapsulates the shared data. *Provides subprogram access* of the corresponding shared data component can be called in actions blocks. *Get_Resource* and *Release_Resource* runtime services specified in the runtime support of the AADLv2 standard [3] can be inserted in action blocks. The user can also provide specific implementations of these runtime services at execution platform level.

An AADL-BA behavior specification improves an AADL architecture description by refining the actions (subprogram call sequences, outputs...) executed within a component. Thus, we can specify both the architecture and the behavior of the DRE system in a same model. Finally, several approaches for defining the semantics or for interpreting the behavior annex of AADL have already been proposed [10], [11], [12].

C. The MPC Case-study Behavior

The figure 4 describes the behavior automaton attached to the `Sender_Thread` periodic thread. At every dispatch of the thread, the transition is fired and the subprogram `Send` specified in the action block is executed.

```
annex behavior_specification **{
 states
  stExec : initial complete final state;
 transitions
  tExec_Exec : stExec -[on dispatch]-> stExec
       { Send!(Data_Source); } -- action block
**};
```

Fig. 4: `Sender_Thread` periodic thread behavior automaton

The figure 5 shows the behavior automaton of the `Receiver_Thread` sporadic thread. At the reception of the `Data_Sink` data the thread dispatches. The transition is fired and the `Update` subprogram which updates the position of the spacecraft and stores it in the local protected object `Protected_Local` is executed. As we access to a shared data the `Get_resource` and `Release_resource` AADL runtime services are used to model the critical section.

IV. Building Blocks of the MBT Process

In section II) we have exposed the rationale for a MBT process to support the validation and implementation of HI systems based on an iterative refinement of architectural

```
annex behavior_specification **{
 states
  stExec : initial complete final state;
 transitions
  tExec_Exec : stExec -[on dispatch Data_Sink]-> stExec
       { Get_Resource!(Protected_Local);
         Update!(Data_Sink);
         Release_Resource!(Protected_Local); } -- action block
**};
```

Fig. 5: `Receiver_Thread` sporadic thread behavior automaton

descriptions. In previous sections, we introduced the AADLv2 and its behavioral annex.

AADL is a complete architecture description language, supporting modeling through abstractions thanks to common concepts (component categories and properties; inheritance and refinement mechanism) for static modeling. The behavior annex completes this description to allow for dynamic modeling, with a formal semantics.

To support our process, we define a minimal set of abstractions out of the full AADL language: AADL-light. The objective is to keep relevant features to maintain some level of abstraction; while rewriting the ones that induce interpretation (see next section). Then, we introduce the building blocks of our runtime as a library of AADL models: POLYORB-HI-AADL.

A. AADL-light: an AADL profile

1) Rationale: The rationale for defining AADL-light is to define a subset of AADL, without any convoluted features. This is permitted per the AADLv2 standard, in the "Profile and Extensions" section. By convoluted, we mean any concept that requires interpretation through syntactic mechanisms such as inheritance (e.g. of properties), refinement (of features or components). These interpretations are performed in any analysis tool to extract relevant information from a model, like we did in the AADL toolchain OCARINA [7].

By defining such profile, we define the target language of our MBT process: every AADLv2 input model would have a semantically equivalent model in AADL-light by rewriting.

In the following, we review the limitations we put on AADLv2 to define our profile.

2) Abstract, Prototypes and Refinement Restrictions: AADLv2 defines the concept of abstract entities such as features, component type and implementation.

Abstract components are used in early steps of the modelling process to model components without knowing the precise component category of the element (tasks, shared object, etc). It is refined later in a concrete component categories (i.e. hardware, software). Abstract features have the same characteristics and are used for modelling component interfaces (i.e. port, subprogram, etc). This is equivalent to a Java or Ada interface.

A concrete component obtained from an abstract component inherits these architectural specifications (properties, composition) and behavioral specification (automaton) through an extension/refinement mechanism. Besides, in the final model of a system, the root node of the model can only have concrete component types in its transitive closure, as subcomponents.

Thus, we can rewrite these concrete components with all information extracted from the inherited abstract component and eliminate the latter from the current model. This suppresses an indirection between components that is relevant for modeling at large, but is irrelevant when considering the final system. The same strategy can be applied on concrete components that extends other concrete components.

Prototypes represent parameterization of component. Prototypes binding allows parameterization via prototype to be propagated down the system hierarchy (as Ada generics). This concept is useful to reuse components and reduce the size of the model. As for abstract components, prototypes cannot be subcomponents of the root node of the model, but only components that contain prototypes/prototype bindings. These are rewritten with the expected component types or component implementation.

3) Data Component Restrictions: The AADL language allows to describe data component in different forms and not fully specified. It can be later refined and completed. The "Data Modelling" annex document lists possible patterns and associated properties.

To avoid the use of multiple modelling patterns as inputs in our analyzers and code generator we restricted the different forms of AADL data component specification to one. In addition, we impose the use of typed and bounded arrays accord to the HI-DRE systems restrictions.

This restriction serves two objectives: providing a uniform view of modeling patterns to ease model processing; and bounding memory consumption.

4) Port and Port Connection: AADL ports defines interactions between components. They are used to transfer event, data or both between threads, processors or devices. A port according to its properties could be equivalent of a signal, a FIFO or a buffer and associated runtime elements such as mutexes, semaphores, etc.

AADL defines low-level mechanisms to lock a resource using data components, define arrays and dispatch triggers. We take advantage of these features to rewrite ports and port connections to make all runtime elements implicit at model-level. All queues and port variables are made visible. We keep only the event port "Dispatch" mandated by the standard. All other ports are associated to a state variable used by the thread automaton (see next section).

By doing so, we allow for finer analysis by making visible all shared objects. Such visibility is mandatory for precise schedulability analysis, or to perform model or code optimisations based on buffer usage like in [13].

5) AADL Behavior Annex Restrictions: The restrictions defined for the AADL core language also have an impact on the behavioral specification. All entities defined to map ports onto queues and port variables require control mechanism that can be defined using the behavioral annex. Yet, it is permissive and define several structures to model critical section [14].

Again, to ensure uniqueness of modeling patterns per concept and ease analysis, we mandate the use of *Get_Resource* and *Release_Resource* services and force the specification of the *Release_Resource* after every *Get_Resource*. Such restriction is usually mandated by coding practices for HI systems.

B. The POLYORB-HI-AADL Middleware

1) Rationale: In the previous section, we illustrate how to map individual AADL components onto simpler ones. Yet, one needs additional runtime support to interconnect them.

We introduced the POLYORB-HI family of AADL runtimes [4] for C/RT-POSIX, C/ARIN653 and Ada/Ravenscar. They rely on similar coding patterns to support AADL runtime services. We propose to move these patterns at model-level to complete the AADL-light patterns with explicit runtime elements, so as to define a "POLYORB-HI-AADL" variant of the POLYORB-HI family. Hence, the combination of AADL-light and POLYORB-HI-AADL provides visibility over all components that may induce resource consumption (memory, time) and their interleaving.

The initial definition of POLYORB-HI relies on canonical middleware services to support event and data exchange in HI systems in both monolithic and distributed case: representation, transport, protocol, execution, naming and activation. We already shown that these services are specified by a set of customizable or generated components (i.e library). Customizable components are used to configure service which the implementation is independent of the application. Generated components are used to generate service which implementation must be generated from system properties. We propose to apply the same design philosophy using AADL-light combined with its behavioral annex as a meta-programming language.

In the following subsections we give a brief description of the AADL-light customized or generated components used by the POLYORB-HI-AADL services. Then we illustrate how these services are integrated in an AADL application model.

2) Middleware service components: Deployment (resp. configuration) information extracted from the AADL application model are generated in a package (resp. property set). They represent constant, node types, entities and enumerations used by execution, naming and activation services to identify nodes reachable by the local node.

Data types and routines required to manage thread's activity are part of the *Activation* service. It also specifies thread interface, and high-level routines as `Send_Output`, `Receive_Input` and `Await_Dispatch`. The Ravenscar Profile restrictions required a single queue for messages received by a thread. Thus, for each thread we generate a global queue (*GQ*) entity as a shared data component with subprograms to manage it (`Wait_Event`, `Dequeue`, etc). These global queues are part of the *Execution* service. Both *Execution* and *Activation* services are in charge of processing a request.

The *Naming* service defines data components used to let a node reach another node. For each node we specify a `Naming_Table` deduced from the distributed application connection topology. This structure contains all port/location information required to establish a connection with the nodes connected to the local node.

The *base type marshallers* package models all marshaller/unmarshaller subprograms for AADL basic types. For user data components we generate marshaller/unmarshaller subprograms according to the distributed application properties. Analyzing connection deep memory copy in buffer (for homogeneous system) or standard protocols as *CDR*, *XDR* (for heterogeneous system) are selected to ensure interoperability. These subprograms define the *Representation* service.

The *Protocol* service defines among other things the `Send` subprogram used by a node to send a message to another node. This subprogram called the `Send` (resp. `Deliver`) subprograms of the transport service used in remote (resp. local) communication.

In the case of remote communication it calls the `Send` subprogram provided by the low-level transport layer (in our case the *TCP_IP_Protocol* package). This instance of the *Protocol* service models a TCP/IP based protocol layer, routines and thread to be integrated with the middleware network interface.

```
data Naming_Entry
properties Data_Model::Data_Representation => Struct;
end Naming_Entry;

data implementation Naming_Entry.SC1_K  --  SC1 spacecraft
subcomponents
 Node_Port: data Node_Port {Data_Model::Initial_Value => "12002";};
 Node_Addr: data Node_Addr {Data_Model::Initial_Value => "196.124.97.65";};
end Naming_Entry.SC1_K;

data implementation Naming_Entry.SC2_K  --  SC2 spacecraft
subcomponents
 Node_Port: data Node_Port {Data_Model::Initial_Value => "0";};
 Node_Addr: data Node_Addr {Data_Model::Initial_Value => "127.0.0.1";};
end Naming_Entry.SC2_K;

data Naming_Table end Naming_Table;      --  Naming table
data implementation Naming_Table.impl
subcomponents
 SC1_K: data Naming_Entry.SC1_K;
 SC2_K: data Naming_Entry.SC2_K;
properties
 Data_Model::Data_Representation => Array;
 Data_Model::Dimension => (PolyORB_HI_Generated_SC_2_Configuration::PO_HI_NB_NODES);
end Naming_Table.impl;
```

Fig. 6: Naming service for SC_2 spacecraft: naming_table

3) Integration of the POLYORB-HI-AADL middleware: Figure 6 illustrates the `naming table` component generated for SC_2. Each entry is deduced from the DRE application connection topology. For each node connected to the local node SC_2 we generate a data component with its port and its location (i.e IP).

Figure 7 shows the integration of the POLYORB-HI-AADL middleware services for SC_1 in the case of the message emission. The `Sender_Thread` calls the `Activity.Send_Output` routine (execution service). This routine uses the representation service to marshall the data to be sent, and stores it in the generated global queue by calling the `GQ.Send_Output` routine. Then, the `Network_Sc1_Thread` retrieves the data stored in the global queue and calls the `Send` routine of the transport service.

Similarly, the figure 8 shows the integration of the middleware for SC_2 in the case of a message reception. The thread network stores the received data in the global queue

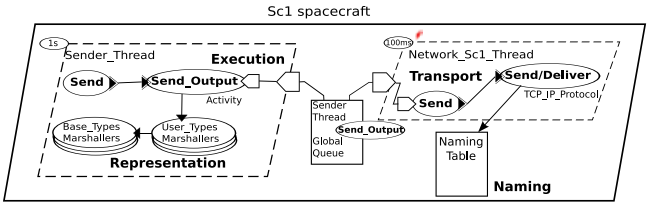

Fig. 7: Middleware integration in SC_1: send_output

and wakes up the `Receiver_Thread` by sending an event on the `Dispatch` port (activation service).

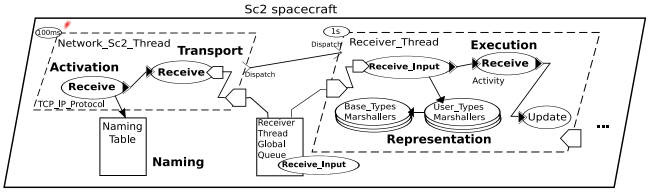

Fig. 8: Middleware integration in SC_2: receive_input

V. MBT Process to Validate and Implement HI-DRE Systems

In the previous sections, we have introduced the need to perform an iterative refinement of a high-level model describing a HI system to ease validation and code generation: each refinement makes visible all underlying runtime elements, allowing for a more precise validation of the entire system. We introduced a subset of AADL combined with the AADL behavioral annex that is sufficient to express the whole runtime environment. In this section we detail the different refinement steps of this process.

A. Model-based Transformations (MBT) Process

The figure 9 describes the MBT process. We define 4 steps to refine the initial AADL architectural description. Step 1 is the expansion of the AADL behavior specification of thread components (periodic and sporadic). Step 2 is the rewriting of high level components and a normalization of data components to AADL-light. Step 3 is the transformation of ports and port connections into a single global queue and the integration of the PolyORB-HI-AADL runtime to make visible all shared components. Finally, Step 4 is an expansion stage for the *Representation* service (i.e marshallers) and for shared data patterns whose implementations are specific to the target programming language for code generation.

1) Step 1 - Integrate Software Component Behavior: The Step 1 makes explicit the behavior of each active thread using the corresponding behavioral automaton, thus to make explicit all dispatch points. Thus, we expand the specification of threads using our AADL-BA behavior thread patterns described in the section III-C (see figures 4 and 5) .

The figure 10 describes the architecture specification of the `Receiver_Thread`. AADL properties and subprogram call sequences are used to produce the behavioral automaton showed

```
thread Receiver_Thread
features
  Protected_Local : requires data access Protected_Type.Impl;
  Data_Sink       : in event data port Record_Type.Impl;
end Receiver_Thread;

thread implementation Receiver_Thread.Impl
calls
 callseq : { Update : subprogram Protected_Type.Update; };
connections
 data access Protected_Local -> Update.This;
 parameter Data_Sink -> Update.Update_Value;
properties
 Initialize_Entrypoint_Source_Text  => "Initialize_Subprogam";
 Compute_Entrypoint_Call_Sequence   => reference (callseq);
 Dispatch_Protocol                  => Sporadic;
 Period                             => 300 Ms;
 Priority                           => 2;
end Receiver_Thread.Impl;
```

Fig. 10: Receiver_Thread architectural specification

in figure 11. Specific subprogram (executed at initialization time, dispatch, etc.) linked by AADL properties are made visible in the automaton. Some other properties remain like priority or WCET as these have no equivalent notation.

```
subprogram Initialize_Spg
 -- source_text or behavior annex subclause
end Initialize_Spg;

thread implementation Receiver_Thread.Impl
 subcomponents
  Init_Spg : subprogram Initialize_Spg;
  Update : subprogram Protected_Type.Update;
 connections
  data access Protected_Local -> Update.This;
  parameter Data_Sink -> Update.Update_Value;
 properties
  Dispatch_Protocol => Sporadic;
  Period            => 300 Ms;
  Priority          => 2;
 annex behavior_specification {**
  states
   stInit : initial state;
   stExec : complete final state;
  transitions
   stInit_Exec : stInit -[ ]-> stExec { Init_Spg!; };
   stExec_Exec : stExec -[ on dispatch Data_Sink ]-> stExec
        { Get_Resource!(Protected_Local);
          Update!(Data_Sink);
          Release_Resource!(Protected_Local); }
 **};
end Receiver_Thread.Impl;
```


Fig. 11: Receiver_Thread behavioral specification

To enrich the description, we also encapsulate the subprogram used to access/modify the *Protected Type* shared data with calls to `Get_Resource` and `Release_Resource` routines. Explicitly providing critical sections allows us to define threads dependencies and to properly evaluate blocking time.

2) Step 2 - Simplification and Normalization: In this step, we rewrite components to remove all AADL modeling constructs that rely on inheritance or refinements. By flattening this hierarchy, we make visible the actual definition of each model element. Then, we normalize all data components to rely on a uniform set of modeling patterns. The figure 12 shows how we rewrite data components. `Integer_32_RW` data component is generated from the `Integer_32` data type that extends `Base_Types::Integer`. Rewritten types have the same properties as the initial ones. Similarly, `Data_Simple_Type.Impl_RW` is generated from `Data_Simple_Type.Impl` and `Data_Simple_Type.P_Impl` by resolving all types referenced by prototypes.

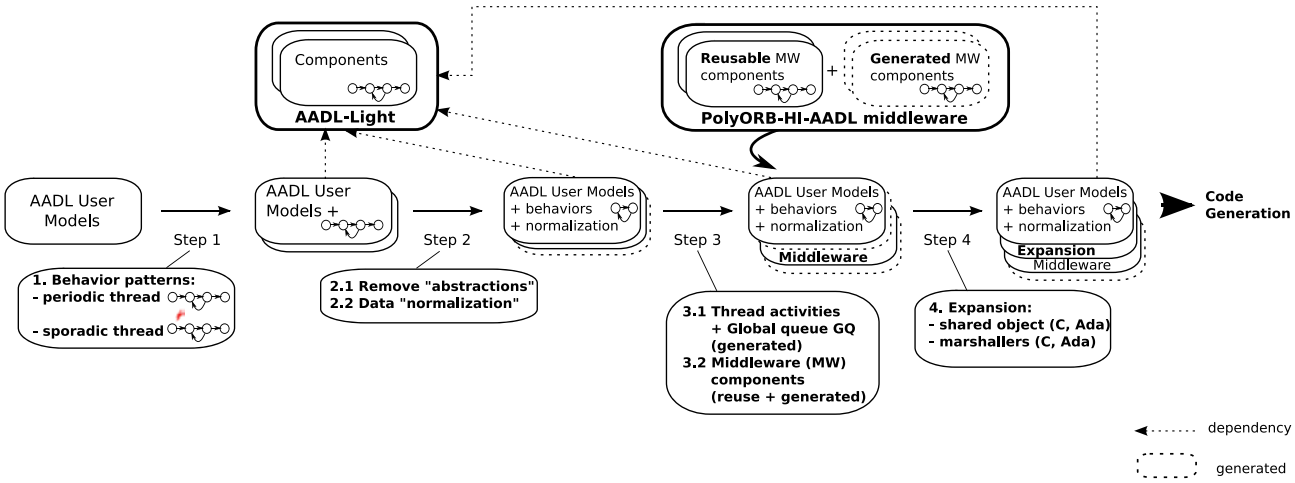

Fig. 9: Model-based Transformation (MBT) DRE System Development Process

```
data Integer_32 extends Base_Types::Integer
properties
 Data_Model::Number_Representation => Signed;
 Source_Data_Size => 4 Bytes;
end Integer_32;

data Data_Simple_Type end Data_Simple_Type;

data Data_Simple_Type.P_Impl
prototypes
 Dt : data;
subcomponents
 Field_1 : data Dt;
end Data_Simple_Type.P_Impl;

data Data_Simple_Type.Impl extends Data_Simple_Type.P_Impl
 (Dt => Integer_32)
end Data_Simple_Type.Impl;
```

(a) AADL data component with prototypes

```
data Integer_32_RW
properties
 Data_Model::Data_Representation => Integer;
 Data_Model::Number_Representation => Signed;
 Source_Data_Size => 4 Bytes;
end Integer_32_RW;

data Data_Simple_Type end Data_Simple_Type;

data Data_Simple_Type.Impl_RW
subcomponents
 Field_1 : Integer_32_RW;
end Data_Simple_Type.Iml_RW;
```

(b) AADL data component simplified

Fig. 12: AADL data component simplification

3) Step 3 - Thread Interface and Middleware Components: In this step, we apply two major transformations:

The first transformation aims at refining thread interfaces (i.e. ports) according to restrictions defined in IV-A4 and IV-B2. All the input ports of a given thread are replaced by an unique queue for this thread: the global queue (*GQ*). This queue models the queue of events/data being received by the thread. This *GQ* component is shared with other threads that produce messages; or network-related threads. This queue is made of several consecutive circular arrays, each one representing a specific port queue of the thread. Additional timestamps ensure causality between messages. Thus it ensures we reproduce the behavior from the original AADL ports. The expansion of ports impacts the thread architecture and the behavior automaton generated in the STEP 1. To stay compliant with AADL and its behavioral annex we let an unique port available called: `Dispatch`. It allows to wake up the thread after the reception of a message in the *GQ*. Upon reception, the thread automaton will walk through the *GQ* and process messages.

The second transformation is the integration of the POLYORB-HI-AADL middleware components (AADL-light library). This transformation makes visible all low-level drivers and attached resources: we select and configure some middleware services (i.e AADL-light packages). In the case of communication between remote nodes we select the network protocol required and configure the network interface. Then, we generate the AADL middleware components missing from the analyze of deployment and configuration attributes coming from user AADL models. At the end of these two expansions we obtain one single model with the whole application and configured middleware.

4) Step 4 - Programming Language Expansion: During this last step, these entities are mapped to regular structures of programming languages such as C or Ada; or other modeling notation for further analysis. Since the resulting model defines all resources (threads, locks, queues, etc), we can define a reduced one-to-one mapping between the model and corresponding artifacts at programming language level. For instance, a synchronizing operation such as a shared data access is not implemented the same way in Ada and in C. Ada 2005 mapping takes advantage of *protected objects* when C has to deal with POSIX conditional variables and mutexes. The same holds for mapping thread definition, automata as sequential code, or data types definition.

B. Integration into OSATE2

OSATE2 [15] is an extensible open source platform developed by the Software Engineering Institute (SEI) to assist the development of HI-DRE systems with AADL. It is based on Eclipse and the Eclipse Modeling Framework (EMF). It includes an AADL meta-model, an AADL front-end and

architecture analysis capabilities as Eclipse-based plug-ins. After syntactic and semantic analysis the front-end produces an instance of AADL metamodel that can be processed.

We have implemented two Eclipse-based plug-ins that exploits models built by OSATE2. The AADL model transformations (AADL-MT) plug-in implements the four steps outlined in our approach using EMF and the ATL model transformation technology [16]. ATL allows to specify a set of transformation rules with OCL [17] expression to produce or refine several target models from input models.

```
-- ATL Helper
helper context AADL2!ProcessSubcomponent def : hasLocation() : Boolean =
 self.ownedPropertyAssociation->select(e | e.property.name = 'Location')
                              ->notEmpty();

-- ATL Matching rule
rule GenerateNamingEntry {
 from cmp : AADL2!ProcessSubcomponent (cmp.hasLocation() and cmp.hasPort())
 to target : AADL2!DataImplementation (
      name <- "Naming_Entry"+cmp.name+"_K"
      ownedDataSubcomponent <- Sequence{ thisModule.generateNodePort(cmp),
                                         thisModule.generateNodeAddr(cmp) } ) }
```

Fig. 13: Naming_Entry ATL transformation rule

The figure 13 shows the ATL matching rule used to generate a `Naming_Entry` data component described in figure 6. The `from` part defines the rule source pattern, a process subcomponent, and ATL helpers to describe conditions on the source element (`hasLocation...`). The `to` part defines the target pattern of the rule. It contains the generated elements and the bindings to initialized their attributes and references.

The AADL code generator (AADL-GEN) plug-in implements our code generator targeting C/POSIX and Ada/Ravenscar languages. We use EMF to produce the C and the Ada meta-models with respect to their BNF. Then, we use ATL to transform AAXL2 models refined by our MBT process to target programming language component. Finally, we produce the textual output files.

VI. Conclusion and Future Works

In this paper we have exposed a Model-Based Tranformation (MBT) process to implement HI-DRE systems. This approach solves complexities introduced by the MDE approach by removing through successive refinements the high level abstractions of components to obtain a model close to the system implementation and integrating execution support (middleware) components.

To reduce the number of intermediate representations, we introduced the AADL-light subset of AADL to express all intermediate models, and the underlying runtime. The definition of this subset allows us to make visible every single entities used in the application, allowing for finer-grained analysis and simpler code generation.

Analysis is made more accurate since one has a faithful view of all resources used, in their context. Deriving an analytical or simulation model no longer requires interpretation of some constructs (e.g. how to read events from a queue).

This approach also eases code generation: the final model makes visible all internal data types and algorithm required to exchange information among model elements (threads, queues, etc). Besides, the integration of middleware components enables optimization at model-level.

Future work will focus on strengthening equivalence of our intermediate models by taking advantage of traceability information and semantics information of all constructs; and extend the coverage of AADL constructs we support as inputs, like modes or properties. Finally, we will work jointly with the SEI on finishing the integration of this work in OSATE2.

References

[1] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in High Integrity Systems," *Ada Lett.*, vol. XXIV, no. 2, pp. 1–74, 2004.

[2] K. B. Arvind, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "Applying model-driven development to distributed real-time and embedded avionics systems."

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

[4] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite," *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 1–25, Jul. 2008.

[5] D. Schreiner and K. M. Goschka, "A Component Model for the AUTOSAR Virtual Function Bus," in *COMPSAC'07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 635–641.

[6] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier, "First experiments using the uml profile for marte," in *ISORC'08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 50–57.

[7] 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.

[8] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://gallium.inria.fr/ xleroy/publi/compcert-CACM.pdf

[9] SAE, *Annex X Behavior Annex (AS5506-X draft-2.13)*, Aug. 2010.

[10] Y. Ma, J.-P. Talpin, and T. Gautier, "Interpretation of aadl behavior annex into synchronous formalism using ssa," *Computer and Information Technology, International Conference on*, vol. 0, pp. 2361–2366, 2010.

[11] Z. Yang, K. Hu, D. Ma, and L. Pi, "Towards a formal semantics for the aadl behavior annex," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009, pp. 1166 –1171.

[12] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Models in software engineering," M. R. Chaudron, Ed. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems, pp. 5–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01648-6_2

[13] P. H. Feiler, "Efficient embedded runtime systems through port communication optimization," in *ICECCS*. IEEE Computer Society, 2008, pp. 294–300.

[14] G. Lasnier, T. Robert, L. Pautet, and F. Kordon, "Architectural and behavioral modeling with aadl for fault tolerant embedded systems," in *ISORC*, Parador of Carmona, Spain, May 2010, pp. 87–91.

[15] SAE AADL, "Osate2," http://www.aadl.info, 2010.

[16] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Sci. Comput. Program.*, vol. 72, pp. 31–39, June 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1385689.1385713

[17] OMG, "Object Constraint Language (OCL)," http://www.omg.org/spec/OCL, 2010.