

# A Generative Approach to Building a Framework for Hard Real-time Applications

Irfan Hamid, Bechir Zalila, Elie Najm, Jérôme Hugues  
GET-Télécom Paris – LTCI-UMR 5141 CNRS  
46 rue Barrault, F-75634 Paris CEDEX 13, France  
Email: {firstname.lastname}@enst.fr

## Abstract

*The communication and tasking infrastructure of a real-time application makes up a significant portion of any embedded control system. Traditionally, the tasking and communication constructs are provided by an RTOS. We present an approach to automatically generate a robust framework for a single-node application from its architectural description. This framework sits atop a Ravenscar-compliant runtime as opposed to a standard RTOS. Finally, we present an extension of our approach to support code generation for distributed applications.*

## 1 Introduction

Control systems deployed on air and space platforms represent one of the most safety-critical categories of software. There are stringent standards of code review and certification [10] that must be met before deployment onboard the platform. Guaranteeing the schedulability and safety properties of the application is one of the costliest aspects of this certification process.

The Ada programming language provides a rich set of tasking as well as inter-task communication constructs and is well-suited to real-time systems development. The Ravenscar Profile [4] defines a subset of Ada that provides schedulability and safety guarantees by restricting the features of the language, this makes it even more amenable to the development of such systems.

The AADL (Architecture Analysis and Design Language) [11] is an architecture description language targeted to real-time, safety-critical embedded systems. It is an evolution of MetaH [16] and uses system design constructs such as processes, threads, processors and buses to model an application. This model may be analyzed for semantic correctness, schedulability, the violation of certain safety properties as well as the generation of application code.

In this paper we present our work on the generation of Ada Ravenscar code from AADL models. We present a method of automatically creating an executable framework corresponding to the AADL system model that runs on top of a Ravenscar-compliant kernel. Once this framework is automatically generated the functional behaviour—the only missing element—can be plugged into the framework to create a working application. An open source Eclipse plugin has been developed that implements these code generation rules (available at <http://aadl.enst.fr/arc/>). We are also developing PolyORB-HI, a lightweight distribution middleware for high-integrity systems to provide mechanisms to build distributed Ravenscar applications.

Section 2 gives an introduction to both AADL and the Ravenscar Profile. Section 3 provides an overview of previous and related work in the same field. Section 4 details the generation of a framework for a single-node real-time system. Section 5 presents a case study using our approach and tools. Section 6 presents how this approach can be extended to distributed real-time systems. Section 7 finishes the article with conclusions and future directions of our work.

## 2 An Overview of AADL & Ravenscar

### 2.1 AADL

The AADL is an architecture description language that uses a component-centric model to define the system architecture, i.e.: a system is a collection of inter-connected components. It allows the modeling of software components (processes, threads, data and subprograms) as well as execution platform components (hardware such as processors, memories, buses and devices). Only non-functional aspects of components can be specified in AADL, e.g.: non-functional properties, interfaces and how they are inter-connected. Behavioural or functional properties must be given separately. This implies giving the source code for software components in a programming language. The

AADL standard [11] defines a textual as well as graphical representation of the language.

Components are elements that require and offer services according to a given specification. AADL differentiates between component *types* and component *implementations*. Component types define the interface (*features* in AADL), alongwith properties of the component. Component implementations define the internals of a component; the subcomponents it contains and the connections among the features of those subcomponents etc. There may be zero or more implementations per component type. A brief description of various AADL component categories is given.

*Process* represents a virtual address space, it may contain threads and data. *Thread* represents the basic unit of execution and schedulability in AADL. *Data* represents a data type (in the declarative part) or a data instance (as a subcomponent). *Subprogram* is a representation of the procedure/function from imperative programming languages. *Processor* represents a microprocessor together with a scheduler for dispatching the threads it hosts. *Memory* represents a storage space. *Bus* represents a hardware communication channel. *Device* represents a hardware with a known interface that can interact with the environment and/or the system under consideration. *System* is a component with no semantics that is used to organize the model.

Component types in AADL may declare features. These are the interface points that the component exposes. Different types of features are possible in AADL.

- *Ports* are data or control transfer-points. Data ports transfer data, event ports transfer control and event data ports transfer control with accompanying data. Furthermore, ports have directional qualifiers (*in*, *out*, *in out*)
- *Data accesses* represent access to a data subcomponent by an external component. This clause is used to model shared data among distant components
- *Subprograms* in the features of a data component represent access procedures (similar to methods of a class), in the features of a thread they represent RPCs

Properties are name/value pairs that can be applied to components, instances, connections and ports. In AADL properties are used to define aspects of the system entities that cannot be expressed otherwise. Examples of properties include `Dispatch_Protocol` and `Period` for threads. A set of pre-defined standard properties exists. The language also has the capability to describe new properties. Properties may be applied to a component type, a component implementation or a subcomponent.

## 2.2 The Ravenscar Profile

Ada tasks and protected objects will be explained briefly before introducing the Ravenscar Profile [4].

Ada tasks [1] (sec. 9.1) are akin to POSIX threads, they are executed independantly in the address space of the host process. Associated with each task is its priority as well as the size of its stack. Ada provides protected objects to allow communication among tasks.

An Ada protected object [1] (sec. 9.4) is akin to a C++ class, in that it exposes operations and contains private data. The major difference being that its operations are concurrency-safe and are the only means of accessing its internal data. A protected object may expose three types of operations; *functions* may not change the internal state but can return a value, *procedures* may change the internal state and may also return values, *entries* may change the internal state and may return values. In addition, an entry has a boolean *barrier* which must evaluate to true before that entry can be executed. If the barrier evaluates to false then the task calling the entry is suspended and is put on the *entry queue*. Furthermore, these barriers must be composed of variables that are part of the private data of the protected object, implying that a procedure of the protected object must be used to change the private data that unblocks the entry.

The Ravenscar Profile seeks to simplify the analysis of real-time systems developed with Ada. Constructs such as rendezvous, task entries and protected objects with multiple entries are prohibited. Tasks and protected objects must be statically declared and tasks may not terminate. Tasks may be either periodic or sporadic. This ensures a static execution model that can be analyzed *a priori* for schedulability using either RMA [14] or RTA [5]. Access to protected objects must be based upon the priority ceiling protocol [15]. This ensures absence of deadlock and also provides an upper bound for priority inversions within the system.

In a Ravenscar system, all communication among tasks must be done through protected objects. Only asynchronous communications are allowed (RPCs are prohibited). The consequence of these restrictions is that a Ravenscar-compliant runtime is much simpler to construct and much smaller than a standard Ada runtime. Aonix RAVEN [2] is a commercial implementation, whereas ORK [7] is an open source implementation.

## 3 Related Work

High-integrity application code generation from models is not limited to AADL. In [3], the authors state that generating code minimizes the risk of several semantic breaches when translating the model towards code. The developer is exposed to these breaches when hand-coding the application. They propose some guidelines to generate Ravenscar-compliant Ada code from HRT-UML. However, extensive use of generic packages in their approach causes code bloat.

More closely to this paper's scope, the Annex D of the AADL [12] describes some coding guidelines to translate

software components into source code (Ada and C). These rules are not complete mapping specifications, but they provide guidelines for those who want to generate code from AADL models. Some of these rules are compliant with the Ravenscar Profile; we integrate them in our code generation rule set. However, these guidelines assume that an “AADL executive” is present to which the generated code makes calls for creating runtime entities that correspond to the model entities.

More concretely, Stood, a tool developed by Ellidiss Software [8] allows users to model their real-time applications using AADL or HOOD [6]. Stood allows the transformation of an AADL model to HOOD, and there exists a mapping from HOOD to Ada. Therefore, users can generate code from AADL to Ada. However, the mapping from HOOD to Ada is semantically similar to that from HRT-UML to Ada, resulting in code bloat.

## 4 Generated Execution Framework

Conventionally, real-time applications have been created on top of real-time operating systems. The RTOS provides services such as thread creation, communication primitives, semaphores etc. as a set of APIs. Listing 1 shows an example of how to create a thread in this paradigm by using the API. This approach is perfectly applicable and mature. However, it cannot be applied to the development of a Ravenscar system. In a Ravenscar system all the tasks (threads) and protected objects (semaphores) must be declared at *library level*, i.e.: statically defined at compile-time. Thus, all tasking and communication constructs are directly manifest in the code rather than being abstracted away as API calls. Listing 2 shows how to create a periodic task in a Ravenscar system.

```

/* Create a periodic thread */
create_periodic(priority , period , &worker);

void worker (void *data) {
  /* Periodic response code */
  ...
}

```

**Listing 1. C code to create a thread**

```

task body Periodic_Task is
  Period : Ada.Real_Time.Time_Span;
  Next_Dispatch : Ada.Real_Time.Time;
begin
  Next_Dispatch := Ada.Real_Time.Clock;
  loop
    delay until Next_Dispatch;
    — Periodic response code
    Next_Dispatch := Next_Dispatch + Period;
  end loop;
end Periodic_Task;

```

**Listing 2. Periodic task in Ada Ravenscar**

It is apparent from listing 2 that building a non-trivial Ravenscar system by hand is a difficult task. The chance for error is significant, added to which is the ungainly nature of the exercise: application-level constructs to implement every task and inter-task communication entity. Fortunately, architecture description languages (ADLs) provide an easy-to-use method of modeling these constructs. Hence an appropriate architecture description language coupled with a custom-built code generator that produces Ravenscar-compliant code seems to be an attractive alternative.

We explored this using AADL and found it to be a viable approach. AADL was selected as the natural candidate for describing Ravenscar system architectures as it is geared toward real-time embedded systems, and has a similar *concept space* as the Ravenscar Profile. Software components such as process, thread, data, and subprogram map naturally to Ada program, task, data type/instance, and procedures.

The static nature of the Ravenscar Profile fits well with AADL, which itself does not allow the creation/destruction of components at runtime. There are, however, certain semantic impacts from Ravenscar towards AADL:

- Threads may only be periodic or sporadic
- Thread priorities are static
- No synchronous communication among threads
- The use of protected objects to implement data ports impacts the dynamic semantics of data ports defined in the AADL standard [11] (sec. 9.1.1)
- The scheduler must be that defined by the Ada standard’s specification for `FIFO_Within_Priorities`
- Shared data must conform to the ceiling protocol

### 4.1 Ravenscar Building Blocks

Reduction of the amount of generated code is a major aim of any code generation approach. Factorizing common components into a library or runtime reduces the complexity of the code generator. However, the static nature of the Ravenscar Profile prohibits a traditional component library/runtime where generating function calls according to the software architecture creates executable entities.

To circumvent this problem, we make use of Ada generic packages to create a set of reusable components. We instantiate these generic packages with parameters corresponding to our system model. The instantiated packages, while being static elements, are nonetheless a form of meta-API. However, extensive use of generic packages results in code bloat, and increases the size of the resulting executable. It is a tradeoff and we try to strike a balance between code reuse and performance by using generic packages only for tasks and event [data] port constructs.

## 4.2 AADL to Ravenscar Transformations

In this section we present the transformation rules from AADL towards Ravenscar-compliant Ada source code. The AADL model that serves as a source for the transformation must conform to certain constraints. There must be no RPC constructs—as only asynchronous communication is allowed—which restricts the use of `provides` | `requires` subprogram access clauses in the features section of threads and processes.

A valid AADL source model for a single-node Ravenscar application consists of a single system implementation which contains a single process implementation. The process implementation must contain at least one thread subcomponent and zero or more data subcomponents. Threads may have any combination of `[event]` `[data]` port clauses in their features. Furthermore, data access clauses in thread features are also legal, as they represent access to shared data declared at process level.

### 4.2.1 Data Components

Rules for transforming AADL data components to Ada data types and data subcomponents to variables in the source code are the same as described in [17]. Data components are transformed by analysis of their properties:

- `Data_Type`: An enumeration from `{Integer, Boolean, Character}`, defines the primitive Ada type corresponding to this data component
- `Length`: An integer to specify the length of an array if the data component is to be transformed to a vector
- `Element_Type`: Designates the element type of an array, it is a data component reference
- `Concurrency_Control_Protocol`: An enumeration from `{NoneSpecified, Priority_Ceiling}`, specifies if a data component has protected access

```
— Integer
data Int_Type
properties
  Data_Type => Integer;
end Int_Type;

— Integer vector
data Int_Vector
properties
  Length => 10;
  Element_Type => reference Int_Type;
end Int_Vector;
```

Listing 3. Some AADL data definitions

Listing 3 gives the AADL code for two different data components, one of which is transformed to a primitive type and the second to a vector. The Ada code after the transformation is shown in listing 4.

```
— Int_Type
type Int_Type is new Integer;

— Int_Vector
type Int_Vector is array (1 .. 10) of Int_Type;
```

Listing 4. Corresponding Ada data types

### 4.2.2 Processes

A process component is transformed to an Ada main unit. The sole purpose of this compilation unit is to include the package that declares the Ravenscar tasks. The application entrypoint simply goes into an infinite loop. Being the lowest priority thread, it serves also as the background task.

### 4.2.3 Threads

Each AADL thread subcomponent of the sole process implementation is transformed to an Ada task. The AADL property `Dispatch_Protocol` can be `Periodic` or `Sporadic`. For each AADL thread we instantiate an appropriate generic package among `Ravenscar_Periodic` or `Ravenscar_Sporadic`. Other thread properties are used to parametrize the instantiation of the generic package. These include `Thread_Priority`, `Source_Stack_Size` and `Deadline`. The `Period` property represents the period for a periodic thread or the minimum inter-arrival time of dispatching events for a sporadic thread. The `Compute_Entrypoint` thread property gives the name of the procedure to call upon dispatch of the thread.

The specification of the `Ravenscar_Periodic` generic package is given in listing 5. Figure 1 shows an example of how this transformation would take place. The dashed parallelogram is the standard graphical notation of a thread in AADL. The property associations given inside the thread are for clarity, it is not part of the standard notation.

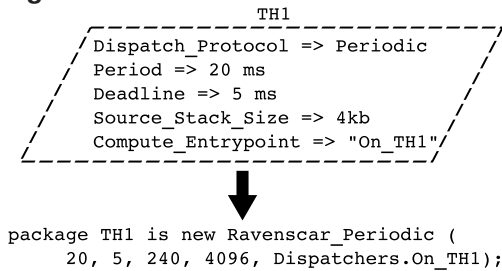
```
generic
  Period_P : Ada.Real_Time.Time_Span;
  Deadline_P : Ada.Real_Time.Time_Span;
  Priority_P : System.Any_Priority;
  Stack_Size_P : Natural;
with procedure Dispatch;
package Ravenscar_Periodic is
  task Task_Instance is
    pragma Priority (Priority_P);
    pragma Storage_Size (Stack_Size_P);
  end Task_Instance;
end Ravenscar_Periodic;
```

Listing 5. The `Ravenscar_Periodic` package

### 4.2.4 Data Subcomponents

Data subcomponents in threads or processes are transformed to instances of protected types generated specifi-

**Figure 1. Periodic thread transformation**



cally to conform to the data component classifier's specification. For data subcomponents of a thread, we declare a variable having the name of the subcomponent and the type mapped from the corresponding data component. For data subcomponents of a process component, the variable is declared in a package visible to all tasks in the application.

This approach works well for monolithic applications. For distributed applications, and due to the restrictions of the Ravenscar Profile, the transformation of shared variables has to be amended (see section 6).

#### 4.2.5 Ports

AADL provides data ports, event ports and event data ports. Data ports and event data ports have an associated data type, called a classifier in AADL terminology. This classifier may be a data component type, or a data component implementation. Ports can be connected to other ports (or left unconnected). Connections are legal between two data ports, between two event ports or between two event data ports. Furthermore, for data and event data ports the classifier (their data type) must be the same.

We call the *peer* of a port the port(s) to which the former is connected. In addition to being divided into the three categories described, ports are also qualified with a direction (in, out or in out). An in port may be connected to an out port or an in out port. An out port may be connected to an in port or an in out port. An in out port may be connected to any compatible port irrespective of its directional qualifier.

**Data Ports** A connected data port in AADL represents a state. Writing a value on an out data port transfers the value to the peer in data port. A data port does not have a queue, i.e.: two successive writes without an intervening read imply the first value written is lost for the destination thread. It is for this reason that multiple incoming connections (fan-in) to data ports are illegal.

We transform two connected data ports on co-located threads to an *exchanger*. We define an exchanger to be an Ada protected type with no entry and two procedures, `Get_Value` and `Set_Value`. The system model is traversed

and an exchanger *type* is generated for each unique data port classifier. For each [in | in out] data port of a thread an exchanger of the correct type is instantiated. The private data part of the exchanger contains a single variable of the same type as the *classifier* of the data port it represents.

The specification of an Integer exchanger is given in listing 6. It serves as a 1-place exchange buffer for an Integer. An auxiliary internal variable `Fresh` keeps track of whether the current update—a consequence of the last invocation of `Set_Value`—has been read. The parameter `Priority_P` specifies the ceiling priority of this exchanger.

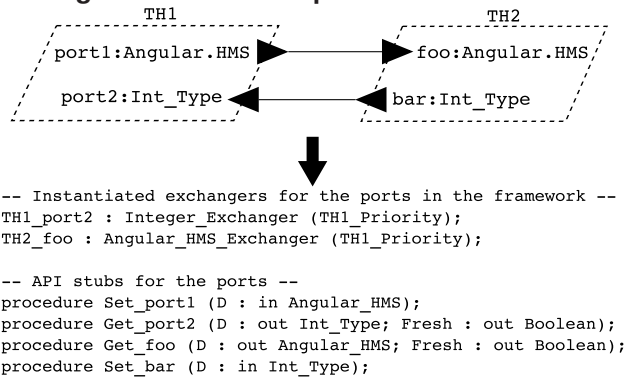
We instantiate an exchanger with a unique name for each in data port of a thread along with an accessor stub `Get_<portName>`. For the thread with the peer out data port, a stub named `Set_<portName>` is generated that writes to the exchanger. An example of this transformation is shown in figure 2. The graphical notation for a data port is ▶

```

protected type Integer_Exchanger
  (Priority_P : System.Any_Priority) is
  procedure Set_Value (D : in Data_Type);
  procedure Get_Value (D : out Data_Type;
    F : out Boolean);
private
  pragma Priority (Priority_P);
  Data : Integer;
  Fresh : Boolean := False;
end Integer_Exchanger;
  
```

**Listing 6. Integer\_Exchanger type**

**Figure 2. AADL data ports transformation**



**Event and Event Data Ports** A connected event [data] port represents a channel for control-flow. Event ports transfer events asynchronously from the source to the destination thread whereas event data ports transfer events *and* associated data. Only sporadic threads may have incoming event [data] ports, and must have at least one to enable dispatching.

For each sporadic thread, an enumeration is generated

that names every in event [data] port in its features section. A variant Ada record type is also generated, with a discriminant `Port` and whose variant parts are all the event types the sporadic thread may receive. We call this variant record the *event interface block* (EIB) for the task. An example of this transformation is depicted in figure 3. Event ports are shown as  $\triangleright$  and event data ports as  $\triangleright\triangleright$ .

For each sporadic task a single *synchronizer* is instantiated in the framework. We define a synchronizer to be a generic Ada package containing a single protected object that has one entry `Await_Event` and a procedure `Send_Event`. Its specification is given in listing 7. A synchronizer has an n-place circular buffer as internal data which stores the events (and associated data for event data ports) received for its associated sporadic task. The barrier to the entry `Await_Event` is true whenever there is at least one event in the internal event queue. The sporadic task will call the `Await_Event` of its synchronizer as its first statement and thus is released only if an event is available.

A single behaviour may be attached to periodic threads at the AADL model level. This is either the unique call sequence given for the thread or the procedure given as the `Compute_Entrypoint` property of the thread. However, a sporadic thread may have separate behaviours attached to each of its in event [data] ports. This is accomplished by setting the `Compute_Entrypoint` property of each incoming event [data] port to the appropriate procedure that should be called upon its reception.

```

generic
  Q_Length : Integer;
  Is_Hot_Overflow : Boolean;
  type EIB is private;
package Ravenscar_Synchronizer is
  type Q_Type is array (0..Q_Length-1) of EIB;
  protected Synchronizer is
    procedure Send_Event (Event_Data : in EIB);
    entry Await_Event (Event_Data : out EIB);
  private
    Event_Queue : Q_Type;
    Head : Integer := 0;
    Tail : Integer := 0;
    Event_Present : Boolean := False;
  end Synchronizer;
end Ravenscar_Synchronizer;

```

**Listing 7.** Synchronizer meta-component

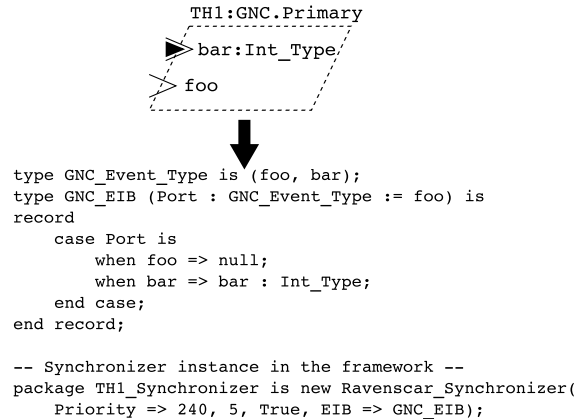
#### 4.2.6 Dispatch Procedures

For each sporadic task, a procedure named `<threadInstance>Dispatch` is generated. This procedure is called at every dispatch of the task. It carries out the following actions on each dispatch:

1. Calls `Await_Event` entry of its task's synchronizer
2. If there is a call sequence corresponding to the dispatching event, its procedures are called

3. If no corresponding call sequence is defined, then the `Compute_Entrypoint` of the event [data] port that dispatched the task is called
4. Suspends self until `dispatch_time + interarrival_time`

**Figure 3.** Sporadic thread transformation



## 5 Case Study

We provide a case study in order to illustrate the approach. We design a simple flight management computer in AADL. Our code generator gives us the source code of the Ravenscar-compliant framework. We then *flesh out* the framework with hand-written behavioral code.

### 5.1 Problem Statement

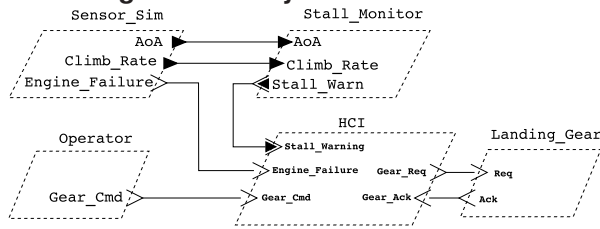
We describe a hypothetical aerial platform that has three sensors. One sensor updates the climb-rate of the platform every 20 ms. Another sensor gives the angle-of-attack—angle between the long axis of the fuselage and the direction of airflow, abbreviated *AoA*—every 20 ms. The third sensor raises an interrupt in case of engine failure.

The platform has a landing gear subsystem. This subsystem can be sent an event which causes it to raise or lower the landing gear. When the gear is raised/lowered and locked, it sends an event confirming the action.

An aircraft is said to be in a *stall* when there is a loss of effectiveness of the aerodynamic control surfaces. In our platform, the flight management computer should sound a soft stall warning when  $AoA > 40^\circ$  and a hard stall warning when  $AoA > 22^\circ$  and  $climb\_rate < 10\text{feet/sec}$ . The operator has an HCI system that consists of:

- A button to raise/lower the landing gear
- An LED that blinks while landing gear is in transit
- An audio alarm that sounds with different frequencies in case of a soft stall, hard stall or engine failure

**Figure 4. The system architecture**



## 5.2 Design

A graphical representation of the AADL system architecture is shown in figure 4. A data port is represented by  $\blacktriangleright$ , an event port by  $\triangleright$  and an event data port by  $\blacktriangleright$ .

Stall\_Monitor is a periodic thread that monitors the data from the AoA and climb-rate sensors. It has a period of 20 ms and receives the data through two data ports: AoA and Climb\_Rate, both of type Integer. The *functional unit* of this thread—defined as its Compute\_Entrypoint property—carries out the calculations according to the conditions defined in the problem statement. In case of a soft stall it raises an event on the Stall\_Warn port with 1 as the argument, for a hard stall with 2 as the argument.

HCI is a sporadic thread that manages the display and audio alarm system, it has a minimum inter-arrival time of 10 ms for events. An Integer event data is received on Stall\_Warning, with the value set according to the severity of the stall (hard or soft). An event is received on Engine\_Failure if the sensor raises an interrupt. Gear\_Cmd is the event received from the operator whereas Gear\_Req and Gear\_Ack are the event interfaces with the landing gear subsystem. Each incoming event [data] port is assigned its own Compute\_Entrypoint.

Landing\_Gear represents the landing gear subsystem. It is a sporadic thread with a minimum inter-arrival time of 3 seconds for events. When it receives an event on the event port Req it starts the landing gear raising/lowering operation. Upon completion of the operation it sends an event Ack back to HCI (nominally after 3 seconds).

Sensor\_Sim is a periodic thread that simulates the sensors by updating the sensor data every 20 ms. This thread also has an out event port that simulates the engine failure sensor. Similarly the thread Operator simulates the pilot. It is a periodic thread with a period of 10 seconds, at each dispatch it sends the order to raise/lower the landing gear via its event port Gear\_Cmd.

We assign a deadline  $D$  equal to the period or inter-arrival time for each thread except Sensor\_Sim, which is assigned a deadline of 15 ms while it has a period of 20 ms. Table 1 summarizes the tasking configuration. The code generator automatically assigns priorities according to the deadline monotonic approach. Code is generated for the

ERC32 processor where a higher numeric value denotes a higher priority.

**Table 1. Tasking configuration**

Task	Deadline	Priority	WCET
HCI	10 ms	240	3 ms
Sensor_Sim	15 ms	239	$\approx 7$ ns
Stall_Monitor	20 ms	238	1 ms
Landing_Gear	3 secs	237	3 secs
Operator	10 ms	236	$\approx 3$ ns

The sensor simulation and monitoring threads have the same period yet we give a higher priority to the sensor simulation thread (via an earlier deadline) so that it sends new data to the stall monitoring thread at every cycle. The WCETs for the sensor and operator simulation threads are minuscule.

**Table 2. Protected object configuration**

Name	Source Artifact	Priority
Stall_Monitor_AoA	port AoA	239
Stall_Monitor_Climb_Rate	port Climb_Rate	239
HCI_Synchronizer	HCI	240
Landing_Gear_Synchronizer	Landing_Gear	240

## 5.3 Generated Code

We obtain a number of source files when we pass the AADL model through our code generator. These files constitute the generated framework as well as the *holes* (callbacks) where the system designer plugs in functional code. These holes manifest themselves as the skeletons of the various procedures declared as the Compute\_Entrypoint of threads and event [data] ports. The entrypoint(s) for each thread are grouped together in a package called  $\langle$ threadInstance $\rangle$ \_Funit. As an example, the Ada code for the entrypoint procedure of the Stall\_Monitor thread is given in listing 8. The skeleton is generated automatically, the code inside the procedure is written by the system designer. The various [xxx] START/STOP comments allow the code generator to save the user code between successive code generations. The listing also shows the use of the generated stubs to manipulate the ports this thread has.

Another important generated package is for the dispatchers of sporadic threads. Dispatch procedures are not meant to be edited by the system designer. But the code of the HCI dispatch procedure shown in listing 9 clarifies how the EIB and event [data] ports are used to dispatch events.

```

— Entrypoint for Stall_Monitor —
procedure On_Stall_Monitor is
— [proc On_Stall_Monitor decls] START
CR : Integer;
AoA : Integer;
Fresh : Boolean;
— [proc On_Stall_Monitor decls] STOP
begin
— [proc On_Stall_Monitor code] START
Get_Climb_Rate (CR, Fresh);
Get_AoA (AoA, Fresh);

if AoA > 40 then
  Raise_Stall_Warn (2);
elsif AoA > 22 and CR < 10 then
  Raise_Stall_Warn (1);
end if;
— [proc On_Stall_Monitor code] STOP
end On_Stall_Monitor;

```

### Listing 8. Stall\_Monitor response code

The communication infrastructure is generated in the `Local_Comm` package. An exchanger is instantiated for each in data port and a synchronizer for each sporadic thread. Table 2 gives the names and ceiling priorities for the various communication constructs generated.

```

procedure HCI_Dispatcher is
S : Task_Features.HCI_T_EIB;
Next_Dispatch : Ada.Real_Time.Time;
use Ada.Real_Time;
begin
HCI_Synchronizer.Synchronizer.Await_Event (S);
Next_Dispatch := Clock + Milliseconds (10);
case S.Port is
when Task_Features.Engine_Failure =>
  HCI_Funit.On_Engine_Failure;
when Task_Features.Gear_Cmd =>
  HCI_Funit.On_Gear_Cmd;
when Task_Features.Gear_Ack =>
  HCI_Funit.On_Gear_Ack;
when Task_Features.Stall_Warning =>
  HCI_Funit.On_Stall_Warning (S.Stall_Warning);
end case;
delay until Next_Dispatch;
end HCI_Dispatcher;

```

### Listing 9. Dispatcher procedure for HCI

## 5.4 Schedulability Analysis

We perform schedulability analysis on the given task-set according to the Response Time Analysis (RTA) methodology described in both [4] and [5] (pages 475—479).

The *interference time*  $I$  for a task is the time spent executing higher priority tasks when the task in question is ready. The *blocking time*  $B$  of a task is the time a task may be blocked by a lower priority task due to priority inversion. If both are known then the well-known result of equation 1 can be used to compute the response time  $R$ , where the summation term gives the interference time.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (1)$$

For a system with a priority ceiling protocol, the term  $B_i$  can be calculated using equation 2. Where  $usage(k, i) = 1$  if resource  $k$  is used by at least one task with a priority less than task  $i$  and at least one task with a priority greater than or equal to task  $i$ ; otherwise  $usage(k, i) = 0$ .  $C(k)$  is the WCET of critical section  $k$ .

$$B_i = \max_{k=1}^K (usage(k, i) \times C(k)) \quad (2)$$

We calculated the execution times for exchangers and synchronizers. The two exchangers in the system are of type `Integer`, conformant to the classifiers of the two data ports. The WCET for an `Integer` exchanger `Set` operation is  $2 \times 10^{-3}$  ms. For an `Integer` exchanger `Get` operation it is  $2.5 \times 10^{-3}$  ms. For both `Send_Event` and `Await_Event` operations of the two synchronizers—for the tasks `HCI` and `Landing_Gear`—the WCET is  $2.5 \times 10^{-3}$  ms.

These obtained WCET values are applied as properties to the various AADL entities to enable detailed, automatic schedulability analysis. We define a new AADL property set `Ravenscar`, which includes (among others):

- `Read_WCET/Write_WCET`: Applies to data components and gives the WCET for a `Get/Set` operation on an exchanger (data port) of this type
- `Send_Event_WCET/Receive_Event_WCET`: Applies to threads and gives the WCET for a `Raise_Event/Await_Event` operation on that thread's synchronizer (which implements all in event [data] ports for that thread)

Using these properties the code generator will be able to perform a precise response time analysis. If the task set is not schedulable, the analyzer will be able to pinpoint the task(s) that will not meet their deadlines.

## 6 Extending to Distributed Systems

In this section we give the extension of the `Ravenscar`-compliant code generation rules to transform AADL models to distributed applications. These applications rely on our middleware for high-integrity systems, `PolyORB-HI`.

### 6.1 Distribution Model for HI Systems

High-integrity (HI) distributed systems have to be designed with respect to a set of requirements:

- *Language constructs*: All language constructs used in `PolyORB-HI` be `Ravenscar`-compliant

- *Memory model:* Dynamic memory allocation be avoided to guard against memory exhaustion and the non-deterministic behavior of allocators at runtime
- *Tasking and concurrency model:* A large part of the Ravenscar Profile addresses tasking issues, compliance to which ensures schedulability analysis, absence of deadlocks and bounded priority inversion
- *Transport layer:* This part of the system must not affect the properties obtained by respecting the Ravenscar Profile: in particular, it should bound priority inversion when accessing communication channels, which must be created at system startup and be non-terminating

From the above requirements, we derive a distribution model where the application is made of pairs of “caller/callee” tuples, each located on a different node. The caller marshals the data, retrieves the end-point to the callee, and sends the data through the low-level transport layer. A protocol handler task awakes on the callee side which receives the data, unmarshals it and calls the corresponding processing code. This model is notionally similar to typical RPCs, with the following limitations imposed to avoid blocking:

- There is one communication channel per caller/callee tuple, and one handler task per transport end-point
- Requests are asynchronous
- Data size is bounded and known at compile time

We implement this model in PolyORB-HI, a core that contains the features necessary for asynchronous distributed applications. PolyORB-HI is structured in layers (transport, protocol and application) but without dynamic creation of components. It has a small memory footprint ( $\approx 200Kb$  including the realtime kernel using GNAT for the ERC32).

## 6.2 Code Generation for Distribution

Most of the rules to generate code for monolithic distributed applications have been created with the distributed case in mind. This means that the code generated for a distributed HI application is similar to the code generated for a monolithic one and compliant with both Ravenscar and the *High Integrity Systems Restrictions* (Annex H of [1]).

**Data Exchange** Data must be converted into a representation suitable for network transfer. Data transfer between nodes is request-oriented: message-sending in the AADL model is mapped to request invocations in code. Request invocations are asynchronous to comply with Ravenscar.

For each distributed application, a variant record type is generated. The discriminant of this record is the operation that is called by a node. For each operation that could be invoked remotely, a list of components corresponding to the

operation parameters is added to the request type. A generic marshalling package is instantiated that provides platform-neutral data conversion operations for transmission over the network, these operations are deterministic ( $O(\text{data size})$ ).

### Communication Channels and Request Dispatching

All aspects of the distributed system must be statically known at compile time to conform to HI requirements. Therefore, the entire task-set and all communication channels between nodes have to be created at system initialization. To achieve this, a deployment package is generated for each node that contains the deployment information this node has concerning other nodes. This information constitutes, in conjunction with the generated naming table (see below), the means for a node to send or receive invocations from another node. This mechanism is completely static and does not use dynamic component creation. The deployment package contains two Ada enumeration types:

- *Node\_Type:* for each node we create an enumeration literal that lists all nodes reachable by the application. A literal is allocated for the local node (*My\_Node*)
- *Port\_Type:* for each `in port`, we declare an enumeration literal (see 4.2.5)

The naming table is statically deduced from the connection topology among components in the AADL model. It contains information for each node to establish a connection with and to send/receive requests to/from another node. As this table is statically generated it avoids dynamic fetching of connection end-points and thus enforces compliance to HI system requirements.

**Client Side Remote Activities** The communication between nodes can be subsumed to “method calls”: the end-point of a `data port` connection or an `event data port` connection is generally a procedure parameter. Hence we can transform the message sending into a call to a *stub* that builds a request from the parameters, marshalls this data in the request and transmits it through the network to the receiver subprogram.

For a single instance of interaction, the caller side is seen as a client; the called node a server. Furthermore, the compliance to Ravenscar reduces the invocation call protocol set to the single asynchronous method. Therefore, our code generator produces an asynchronous stub for each “method call”. It also checks that the data flow between components is suitable for asynchronous communication (remotely invoked subprograms must not have `out` parameters).

However, the use of asynchronous communication alone is not sufficient to guarantee correctness of distributed applications sharing data across nodes. We are also designing a Ravenscar Profile compliant consensus algorithm between the different nodes to guarantee data consistency.

**Server Side Remote Activities** The receiver of a request is usually a sporadic AADL thread. Each dispatch of a sporadic thread can be seen as the invocation of a request handler. As stated in [9], an extra thread plays the role of a protocol handler and dispatches the incoming requests to their respective handlers. The generated code for the request handler consists of an instantiation of a generic package which factorizes the code common to all sporadic tasks. Generic instantiation does not cause significant overhead in our case because we use it only for tasks and datamarshallers.

**Assessments** We tested our middleware and code generator on a case study similar to that in section 5. We generated the code, compiled it to the target using the GNAT compiler and simulated it locally using the `tsim` simulator for ERC32. The total size of the executable, including real-time kernel, middleware and the application is 310kB. It fits the requirements for minimal embedded systems, and is under the typical memory range for API-based middleware such as nORB or microORB, which are above 450kB.

## 7 Conclusion and Future Work

We presented a method of generating a customized framework for hard real-time applications. A case study was presented to illustrate the approach with a concrete example. Finally we showed how our approach can be extended in a Ravenscar-compliant manner to generate code for distributed, hard real-time systems. A clear advantage of the approach is the separation of concerns between the non-functional architecture of the real-time system and the functional or behavioural part. It gives system designers a block modeling approach to their problem with clear and well-defined interfaces among those blocks. The behaviour of the individual blocks is given separately. The automatic generation of code ensures conformance to interfaces and non-functional properties as a result of their direct transformation from the model towards code.

We have developed ARC (available at <http://aadl.enst.fr/arc/>), an open source Eclipse plugin that implements the code generation rules for a single-node application. The tool is based on the OSATE AADL parser [13]. The OSATE toolkit represents the AADL abstract syntax tree as an EMF metamodel. We have also defined our own Ravenscar domain model. The tool translates the AADL model to an instance of the internal Ravenscar domain model. Code generation is done by traversal of the Ravenscar domain model instance. We also described PolyORB-HI, a light-weight middleware to generate Ravenscar-compliant code for distributed applications.

Future directions for our work include the detailed exploration of distributed Ravenscar-compliant applications and the amount of coupling we can achieve with them. Also,

since the code generation rules provide a complete bijection among the model and code, we intend to implement a hypertext reporting system that gives complete traceability between the code generated and the model element that it is a consequence of. This traceability documentation is very useful for obtaining certification of flight software.

## References

- [1] Ada Working Group. *Ada Reference Manual*. ISO/IEC, 2005. <http://www.adaic.com>.
- [2] Aonix. Real-Time RAVEN. <http://www.aonix.com/>.
- [3] M. Bordin and T. Vardanega. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 59–67, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] A. Burns, B. Dobbins, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004.
- [5] A. Burns and A. Wellings. *Real-time Systems and Programming Languages*. Addison-Wesley, 3 edition.
- [6] A. Burns and A. J. Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Syst.*, 6(1):73–114, 1994.
- [7] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An Open Ravenscar Real-Time Kernel for GNAT. In *Ada-Europe'00: Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, London, UK, 2000. Springer-Verlag.
- [8] Ellidiss Software. STOOD. <http://www.ellidiss.com/>.
- [9] J. Hugues, B. Zalila, and L. Pautet. Middleware and Tool suite for High Integrity Systems. DEC 2006.
- [10] RTCA and EUROCAE. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.
- [11] SAE. *Architecture Analysis & Design Language (AS5506)*, September 2004. <http://www.sae.org>.
- [12] SAE. *Language Compliance and Application Program Interface*. SAE, 2005. The AADL Specification Document Annex D.
- [13] SEI. Open Source AADL Tool Environment. <http://la.sei.cmu.edu/aadl/currentsite/tool/osate.html>.
- [14] L. Sha, M. H. Klein, and J. B. Goodenough. Rate Monotonic Analysis for Real-Time Systems. *Computer*, 26(3):73–74, 1993.
- [15] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [16] S. Vestal. *Software Programmer's Manual for the Honeywell Aerospace Compiled Kernel (MetaH Language Reference Manual)*. Honeywell Technology Center, 1998.
- [17] B. Zalila, I. Hamid, J. Hugues, and L. Pautet. Generating Distributed High Integrity Applications from their Architectural Description. In *12th International Conference on Reliable Software Technologies*, 2007.