Using an Architecture Description Language to Model Real-Time Kernels

José A. Pulido Santiago Urueña Juan A. de la Puente Juan Zamorano
DIT-UPM DATSI-UPM DIT-UPM DATSI-UPM
pulido@dit.upm.es suruena@datsi.fi.upm.es jpuente@dit.upm.es jzamora@datsi.fi.upm.es
Draft paper submitted to CEDI’05

Abstract
Real-Time embedded systems engineering needs new solutions in order to reduce costs and increase its productivity. One of the tools that can be used to achieve this goal is the Architecture, Analysis and Design Language (AADL), which enables the system architecture to be modelled in detail, thus helping to prevent errors made because of lack of accuracy in the specifications. This paper presents an AADL overview together with the model of a real time kernel. An assessment of the fitness of the language to model this kind of software is done, and some enhancements to the current AADL standard are proposed.

1 Introduction
The development of Embedded Systems has a strong handicap in the costs growth, which directly falls on the productivity. It is not possible to support this growth for ever. New ideas, concepts and methodologies must be introduced in the Embedded Systems Industry in order to be able to develop the high complexity systems required by the society but, keeping the price in a reasonable level.

These costs are directly related to the number of errors made during the development process. Furthermore, several previous studies have shown that the costs associated to errors gear up as the development process goes on. However, nowadays with the current methodologies of working, it is quite usual to specify the system architecture in a vague or few accurate way, so the errors made during the requirement capture and design phases are not discovered and, therefore, transferred to the implementation phase, which diverge from the initial idea and convert the system integration in a very, very long, ‘painful’ and expensive phase.

Figure 1: System life circle

Another influencing factor are the changes that a product use to experience along its lifetime. Again, the lack of a precise system architecture description involves a strong impact in the maintenance costs of the product.

This paper presents the Architecture, Analysis and Design Language (AADL), a language for describing the whole architecture (software and hardware) of critical, embedded, real-time systems. We summarize the components and features of the language so the reader can quickly familiarize with it. Subsequently, we describe the model of a real-time kernel (ORK) using the AADL, evaluating the different possibilities and analyzing its suitability.

2 Overview of the Architecture Analysis and Design Language
The AADL is derived from MetaH, a language designed by Honeywell for specification of real-time, fault-tolerant, multi-processor system
architectures at the early 90’s. The excellent behaviour of the costs graphic, specially in the ‘re-target’ phase has been the mainspring to develop the AADL.

As written before, the AADL describes both, the application software together with the execution platform. The core of the AADL are the components, which can be enriched through particular properties, connected among them through ports by flows and connection specifications and, modified making use of operational modes.

2.1 Components

The AADL components are defined in two steps. First, the component type specifies its external interface, declaring all the features and subcomponents that are externally accessible. Afterwards, each component type may have zero or more components implementations, which specify subcomponents and connections refinements, private parts of the components and, operational modes, that perform alternate configurations of the component.

The AADL consider the following components, arranged in groups:

- Execution platform components: denotes mainly the hardware of the system although may also include part of the software (scheduler, drivers).
  - Processor: models an abstraction of hardware and software able to schedule and execute threads.
  - Memory: represents storage devices such as RAM, ROM, disks or logical storage.
  - Device: hardware component which internals are not relevant to the system, so only the accessible interface is modeled. Typical examples of devices are displays, sensors or actuators.
  - Bus: component in charge of exchanging information among processors, memories and devices.

- Software components: model just pure software parts of the system.
  - Data: represents a data type in the source text, which may be sharable (concurrency control is assured), and may also have its own operations.
  - Subprogram: is an entrypoint in source text.
  - Thread: schedulable sequential flow of control that executes instructions within the virtual address space of a process.
  - Thread Group: is a logical group of threads. Allow simplifications when designing complex systems.
  - Process: models a virtual address space, which implementation must contain at least one thread (or thread group).

- System: is the component with a highest level of abstraction. It contains both, the execution platform and the software components (in addition to other systems as subcomponents if necessary).

- Package: a package is not a component as the enumerated above, but a way to organize them into sets of declarations by introducing separate namespaces.

2.2 Features, connections and flows

Features are part of a component type definition that specifies how the component interfaces with other elements in the system. There are four categories:

- Port: interface element used for the asynchronous transmission of data and/or control among components. They can be declared in, out or in out.

- Subprogram: as defined above.

- Parameter: represents the data values exchanged by the subprograms.

- Subcomponent access: symbolize a shared access (in or out) to data or buses.
A connection is a linkage that represents communication of data and/or transmission of control between components. There are port connections, parameter connections and sub-component access connections.

A flow is a logical stream of information through a sequence of components and connections which provides the capability of supporting different kinds of analysis such as end-to-end timing and latency. They are represented by flow specification, flow implementation and end-to-end flow declarations.

2.3 Properties

Properties are used in the AADL to provide a wide range of further information about components, features, connections, etc.

The AADL properties are declared in property sets, they have a name, a type and a value. Two of the sets are predefined and contain the most commonly used properties, but the user can declare all the needed properties in other sets, specifying to which kind of components can these new properties be applied. This characteristic makes the AADL an open language able to cover all the needs that a user can require when designing a system.

3 A real-time kernel: ORK

The Open Ravenscar Real-Time Kernel (ORK) is a small, high performance real-time kernel that provides restricted tasking support for Ada 95 programs and it is also usable from C programs.

The kernel is intended to support mission critical real-time software systems. In order to ensure that the software is highly reliable, and even in some cases go through a certification process, program constructs which are not verifiable should not be used. The exact set of language features to be avoided depends on the degree of integrity that is desired for the software and the verification methods that are to be used. Based on these considerations, language subsets for building software with different levels of integrity can be defined. In the case of Ada, there is a standard mechanism to enforce that only the required subset of the language is used by means of the pragma Restrictions and the restriction identifiers that are defined in the ALRM 'Safety and Security' annex.

Tasking has often been considered not to be safe for high integrity systems, mainly due to the difficulty of analyzing and verifying tasking programs. However, recent advances in response time analysis for fixed priority preemptive scheduling enable limited tasking mechanisms to be used even in this kind of systems.

One of the goals of the 8th International Real-Time Ada Workshop, which was held in 1997 in Ravenscar, Yorkshire, England, was to define a safe tasking model for Ada. The outcome of this work is known as the 'Ravenscar profile'. The profile was slightly modified in the following meeting, after some experience was gained on its implementation and use. It is also included in the ISO Ada 95 HIS report.

The profile defines a subset if Ada tasking that includes static tasks (with no entries) and protected objects (with at most one entry), a real-time clock and delay until statements, as well as protected interrupt handler procedures and other tasking related features.

The restrictions in Ada tasking defined in the Ravenscar profile enable tasking to be supported by a small, reliable kernel instead of a full operating system. ORK is one such kernel, which enables critical real-time systems to be executed on a bare processor with no underlying operating system.

The kernel is integrated with the GNAT compilation system. A special cross-compilation version of GNAT is included in the ORK distribution. Real-Time programs are written in a subset of Ada 95 which is consistent with the Ravenscar profile and with other, non tasking restrictions, as desired according to the degree of integrity that is required for the program. The restrictions can be enforced at compilation time by means of appropriate restriction pragmas. The code generated by the special version of GNAT can be loaded on the target hardware by means of appropriate bootstrap loaders. It can also
be executed on a target simulator for testing purposes.

3.1 Architecture of ORK

The kernel consists of the following Ada packages:

- **Kernel**: Root package (empty interface).
- **Kernel.Threads**: Thread management, including synchronization and scheduling control functions.
- **Kernel.Threads.ACTB**: Ada task control block management.
- **Kernel.Threads.Protection**: Kernel protection and thread dispatching.
- **Kernel.Threads.Queues**: Thread queues management.
- **Kernel.Time**: Clock and delay services.
- **Kernel.Memory**: Storage management.
- **Kernel.Interrupts**: Interrupt handling.
- **Kernel.Parameters**: Configuration parameters.
- **Kernel.CPU_Primitives**: Processor-dependent definitions and operations.
- **Kernel.Peripherals**: Support for peripherals in the target board.
- **Kernel.Peripherals.Registers**: Definitions related to input-output registers of the peripheral devices.
- **Kernel.Serial_Output**: Support for serial output to a console.

The kernel is not intended to be directly used from Ada programs. Instead, an interface to the GNU Ada Run-time Library (GNARL) is used so that Ada 95 tasking constructs can be directly used by the real-time application programmer.

The kernel can also be used with programs written in C. Since C has no tasking constructs, the kernel functions for creating and handling threads have to be explicitly called from C. A C API is provided for this purpose. The C interface is provided by a C interface layer, which is integrated with the GCC compilation system.

4 ORK model using the AADL

Once the reader have a notion of both, the AADL and the ORK, we straight away present a first approach of the ORK model using the AADL.

Reading the definition of the AADL processor component type, ‘A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads’, it seems that the most reasonable approach is to consider that the ORK is the software part of an AADL processor, as the ORK contains all the procedures needed to properly schedule the executing threads.

According to this view, the proposed model shown in listing ??, models all the procedures of the ORK interface as server subprograms of a processor component type. In order to facilitate the comprehension of the model, definitions of data and subprogram component types have been split into five packages (as in the Ada source code), where each group contains components related to one of the following issues: time, interrupts, memory, threads and serial output management, although the
4.1 Discussion

There are some open issues derived from this first approach:

4.1.1 Fitness of the processor component

The ORK kernel contains all the features needed to properly schedule threads, and this is its main functionality. However, ORK also manages other issues related to memory management, interrupt handling, timing, and serial output. Perhaps some of these features should be out of the processor component type definition. In fact, subprogram types are defined in packages, and the processor just has server subprograms as 'instantiations' of these types.

An alternative approach would thus be to model ORK as a system containing a processor and some other elements in order to separate the features related to threads from the other ones (see listing ??). However, we also find two problems with this approach:

- There is no satisfactory way of grouping the subprograms, and therefore a system with a large number of them could be difficult to manage. This is an issue of the AADL, which might perhaps be solved by including a *subprogram_group* feature to the language.
- Using the hierarchy of systems from such a low level could make the whole system (from physical processor to application software) unreadable.

4.1.2 Elaboration-Time

The ORK interface contains some procedures which are called only at elaboration time (e.g. *Thread_Create*, *creates a new thread*). They are important features of the ORK so we think that they should be included in the model. On the other hand, the purpose of an AADL model is to describe the static view of a system, this means, a snapshot when all the components (processes and threads) have been already created, so it is quite difficult to model this kind of procedures which create new components in AADL.

There is still an open discussion about the convenience of including this kind of procedures in the AADL model and, if the conclusion is positive, how to do it.

Our proposal consists on modeling the elaboration time making use of different operational modes. The main idea is presented in listing ???. It is a short example where the 'Thread_Create' subprogram generates an event each time a thread is created, so the process containing the threads switch the operational mode to consider one, two or three threads.

5 Conclusions

In this paper we have presented the AADL, which scope is to formally describe the architecture of embedded real time systems, allowing the analysis of their properties, automatic code generation and, predictable integration of its components. After a brief description of
the AADL in chapter 2, we have made the exercise of modelling a real-time kernel like ORK in order to evaluate the fitness of the language to achieve this goal. Next paragraph summarize our conclusions.

There are several choices to model ORK with the AADL. This means that there is no a perfect model where all the components of the model suit perfectly to the kernel, but each model has some advantages and disadvantages.

Basically there are two approaches: modelling ORK as a processor or as a system. The former is more compact and elegant, however, ORK contains some features that are out of the scope of the processor component. The latter allows to include all the features although, there are also a couple of problems:

- There is no an efficient way to group all the subprograms (we propose to add a subprogram group component similar to the thread group).
- Using the system component to model such a low level component as the ORK can produce a quite deep hierarchy of systems which would make the whole model (from application to hardware), not clear and hard to understand.

Another open discussion is, should the whole kernel be modelled? Should the elaboration-time operations be in the model? The AADL is focused on the static architecture of the systems however it seems that operational modes can be a powerful tool to model how this elaboration is performed as shown in the short example listing ??.

Thus, we think that the AADL is a practical and powerful tool, and the current version is a good start point, although some improvements should be done in order to satisfy all the needs referred above.
Listing 1: ORK modelled as a processor

package Kernel
end Kernel;

package Kernel::Time
public
subprogram Clock
features
    Result : out parameter Time;
end Clock;

subprogram Delay_Until
features
    Alarm : in parameter Time;
end Delay_Until;
end Kernel::Time;

processor ORK
features
    -- Time management operations
    Clock : server subprogram Kernel::Time::Clock;
    Delay_Until : server subprogram Kernel::Time::Delay_Until;

    -- Memory management operations
    New_Stack : server subprogram Kernel::Storage::New_Stack;

    -- Interrupts management operations
    Attach_Handler : server subprogram Kernel::Interrupts::Attach_Handler;
    Detach_Handler : server subprogram Kernel::Interrupts::Detach_Handler;
    Current_Handler : server subprogram Kernel::Interrupts::Current_Handler;

    -- Thread management operations
    Initialize : server subprogram Kernel::Threads::Initialize;
    Thread_Create : server subprogram Kernel::Threads::Thread_Create;
    Thread_Self : server subprogram Kernel::Threads::Thread_Self;
    Set_Priority : server subprogram Kernel::Threads::Set_Priority;
    Get_Priority : server subprogram Kernel::Threads::Get_Priority;
    Yield : server subprogram Kernel::Threads::Yield;
    Check_No_Mutexes : server subprogram Kernel::Threads::Check_No_Mutexes;

    -- Serial_Output management operations
    Init_Serial_Line : server subprogram Kernel::Serial_Output::Init_Serial_Line;
    Put_Char : server subprogram Kernel::Serial_Output::Put_Char;
    Put_String : server subprogram Kernel::Serial_Output::Put_String;
    New_Line : server subprogram Kernel::Serial_Output::New_Line;
    Put_line_Char : server subprogram Kernel::Serial_Output::Put_line_Char;
    Put_Line_String : server subprogram Kernel::Serial_Output::Put_Line_String;
end ORK;

processor implementation ORK.Twenty_MHz
properties
    Allowed_Dispatch_Protocol => (Periodic, Aperiodic, Sporadic, Background);
    Thread_Swap_Execution_Time => 25us .. 27us;
    Supported_Source_Language => (Ada95, C);
    Clock_Period => 50ns;
    Max_Thread_Limit => 64;
end ORK.Twenty_MHz;
Listing 2: ORK modelled as a system

```
processor ORK_Processor
features
    −− Time management operations
        Clock : server subprogram Kernel::Time::Clock;
        Delay_Until : server subprogram Kernel::Time::Delay_Until;

    −− Thread management operations
        Initialize : server subprogram Kernel::Threads::Initialize;
        Thread_Create : server subprogram Kernel::Threads::Thread_Create;
        Thread_Self : server subprogram Kernel::Threads::Thread_Self;
        Set_Priority : server subprogram Kernel::Threads::Set_Priority;
        Get_Priority : server subprogram Kernel::Threads::Get_Priority;
        Yield : server subprogram Kernel::Threads::Yield;
        Check_No_Mutexes : server subprogram Kernel::Threads::Check_No_Mutex;
end ORK_Processor;

system ORK
features
    −− Memory management operations

    −− Interrupts management operations

    −− Serial Output management operations

end ORK;

system implementation ORK.Generic
subcomponents
    Main_Processor : processor ORK_Processor;
end ORK.Generic;
```
Listing 3: Modelling elaboration-time with operational modes

thread Control_1
end Control_1;

thread Control_2
end Control_2;

thread Control_3
end Control_3;

subprogram Thread_Create
features
  Created : out event port;
end Thread_Create;

process Manager
features
  Thread_Created : in event port;
  ... 
end Manager;

process implementation Manager.Generic
subcomponents
  Control_1 : thread Control_1 in modes (One_Thread);
  Control_2 : thread Control_2 in modes (One_Thread, Two_Threads);
  Control_3 : thread Control_3 in modes (One_Thread, Two_Threads, Elaborated);
modes
  Zero_Threads : initial mode;
  One_Thread : mode;
  Two_Threads : mode;
  Elaborated : mode;
  Zero_Threads \rightarrow [Thread_Created] \rightarrow One_Thread;
  One_Thread \rightarrow [Thread_Created] \rightarrow Two_Threads;
  Two_Threads \rightarrow [Thread_Created] \rightarrow Elaborated;
end Manager.Generic;