Modeling and verification of memory architectures with AADL and REAL

Stéphane Rubini, Frank Singhoff
LISyC - University of Brest - UEB
20, Avenue Le Gorgeu, CS 93837
29238 Brest Cedex 3, France
{*stephane.rubini,frank.singhoff*}*@univ-brest.fr*

Jérôme Hugues
Université de Toulouse, ISAE
10, Avenue E. Belin
31055 Toulouse Cedex 4, France
jerome.hugues@isae.fr

***Abstract*—Real-Time Embedded systems must respect a wide range of non-functional properties, including safety, respect of deadlines, power or memory consumption. We note that correct hardware resource dimensioning requires taking into account the impact of the whole software, both the user code and the underlying runtime environment. AADL allows one to precisely capture all of them. In this article, we evaluate the AADL modeling to define memory architectures, and then verification rules to assess that the memory is correctly dimensioned. We use the REAL domain-specific language to express memory requirements (such as layout or size) and then validate them on a case-study using the VxWorks real-time kernel.**

***Keywords*-AADL, constraint language, REAL, architecture, verification, memory architecture**

I. Introduction

Real-time embedded systems usually have a limited amount of resources and specially a limited amount of memory resources. Besides, embedded micro-controllers memory mixes different kinds of storage: volatile or permanent, data or code, etc. Thus, engineers need to optimize memory usage, and thus to model precisely both the memory layout of the target system, and the requirements of their applications prior to check they actually match.

In this article, we investigate how such configuration and verification operations can be made at early stage in a model-driven engineering process. We discuss the use of AADL (Architecture and Analysis Description Language) to model a real-time embedded system, including specific properties to model its memory layout.

AADL is a textual and graphical language for model-based engineering of embedded real-time systems that has been approved and published as SAE Standard AS-5506A [1]. This language provides several components categories that define the physical storage components such as hard disks, ROM or RAM components. Software components (processes, data) can be attached to memories, defining resource allocation. AADL components may have properties. Information provided by component properties can be related to the component behavior, the way it will be implemented or anything else that make it possible to perform AADL model analysis. AADL properties are defined in pre-defined property sets but AADL also allows designers to define new component properties in user-defined property sets.

Yet, we note that the pre-defined property sets do not help defining precisely the kinds and layout of memory components. We propose additional property sets to model the logical design of memory components, like permanent storage (flash memory, hard disk), memory pages or memory segments of RAM, etc. This would help engineers to model specific memory layouts, data mapping onto memory or memory segments that are shared by several processors in multiprocessor architectures.

In addition to this set of properties, we specify legality rules in order to check that AADL models of memory layout are compliant with the targeted systems. For this purpose, we use the REAL domain specific language. We illustrate this approach with a complete case study on the VxWorks operating system.

This article is organized as follows. In the next section, we give an overview of memory organization, by focusing on the distinctive features of embedded systems. Section III gives some guidelines to model address space layout with AADL. Section IV discusses verifications that we can expect from the type of models presented in previous section, using the REAL AADL annex language. Finally, section V concludes this article and briefly describes some possible further works.

II. About memory layout models

In a typical computer architecture, the processor communicates with the physical devices which are mapped by the decoding hardware within a single address space. Devices may have different kinds of memory components, or memory-mapped I/O registers. Usually, the width of the address bus limits the range of addresses that the processor may access to. Within the address space, some areas may be forbidden, because no memory device is associated with them or because processor implementation do not allow such address to be used.

The speed of today's high-performance processor is limited by the main memory slowness (typically composed of DRAM cells). Hierarchical structures aim to mask this performance gap, by exploiting the spacial and temporal locality

of memory accesses. Within this structure, caches memories constitute an intermediate level which stores a subset of the memory references, that have the more chance to be accessed in a near future, according to the principle of locality. The virtual memory management completes the hierarchy with mass storage devices like hard drives; it provides a large capacity and some memory protection features. Today, this hierarchical memory structure is supported by all desktop computers.

But, embedded systems use a larger range of memory organizations. To minimize power dissipation, cost, or to respect real-time constraints such systems blend various memory technologies. Within a same address space may reside permanent ROM or EEPROM/FLASH memories, rapid scratch-pad SRAMs or large capacity DRAMs. Cache memories may also be found, but the non-deterministic access time to memory words that this structure involves, can be a major drawback for real-time system implementations. A precise knowledge of the memory layout is needed to build an application in that context. Characteristics like the read access time, the number of supported write cycles, the permanence of the storage,... must be considered in the design and building process. For example, often used program variables can be put on a scratch-pad memory to increase the performance of the system [2].

Operating systems organize the usage of the memory space by splitting it into segments assigned to the storage of different kinds of information. Processor hardware requirements, such as the interrupt vector location, the address bus width, or the growing direction of the execution stack must be obviously considered in this organization. Some segments are reserved for the storage of the operating system internal data or code. Others will be initialized when an application is loaded. The *text*, *BSS* and *data* segments are usually used to store respectively the application code, uninitialized global variables and initialized global variables. The location of these segments within the address space are defined by the link editor and the loader, according to a configuration file called a linker script.

General purpose operating systems assign a private address space to each process, cut off from the other ones. An address in this space is said virtual or logical, and requires a translation before to be send into the physical address space. Hardware devices (i.e. Memory Management Unit) and operating system services support address translations.

However, embedded operating systems may implement another approach where all the processes share a unified address space. Hardware virtual memory support is optional in that context, even if the associated protection and security features remain significant. For example, Windows CE 5.0 splits the memory space into 32 MB slots, where each slot is bound to one process, and large memory blocks are located to a shared segment called *Large memory Area*. *VxWorks* operating system considers tasks as object modules

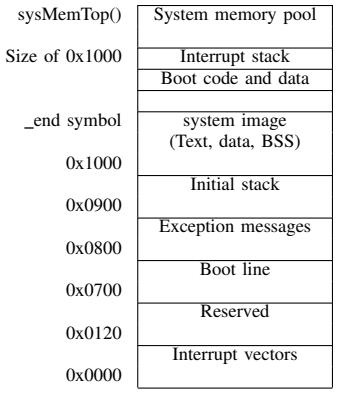

Figure 1. Memory layout of VxWorks for the MC68040 processor

dynamically linked with the operating system kernel and manages only a unique address space. So, each target system has its own memory layout.

Figure 1 lists the memory layout of the VxWorks kernel for a MC68040 processor. This layout defines the location of user code in the *System memory pool*, and several memory areas reserved by the kernel for both the kernel image and interrupt vectors.

The previous discussion shows that, especially for embedded systems, the memory layout may change from one target to another and may be complex. In a context of limited resources, the model of a system must encompass a description of the memory layout, which can be used to check compliance with the application needs or to guide the building process.

III. Modeling memory architectures with AADL

In this section, we present AADL. Then we show how to model with AADL hardware and software memory architectures and especially how software memory architectures are mapped on hardware memory architectures. We also introduce AADL extensions that were required for such a purpose.

A. AADL

AADL is a textual and graphical language for model-based engineering of embedded real-time systems. It has been published as an SAE Standard AS-5506A [1]. AADL is used to design and analyze software and hardware architectures of embedded real-time systems. Many tools provide support for AADL: Ocarina implements Ada and C code generators for distributed systems [3], TOPCASED [4], OSATE [6] and Stood [5] provide AADL modeling features, the Fremont toolset [7] and *Cheddar* implement AADL performance analysis methods [8]. An updated list of supporting tools can be found on the official AADL web site http://www.aadl.info.

An AADL model describes both the hardware part and the software part of an embedded real-time system. Basically,

an AADL model is composed of components types and implementations of different categories: data, threads or processes (components modeling the software side of a system), processors, memories, devices and buses (components modeling the hardware side of a system). A data component represents a data structure in the program source text. It may contain sub-programs that act as accessors. A thread is a sequential flow of control that executes a program and can be implemented by an Ada task or a POSIX thread. AADL threads can be dispatched according to several policies: a thread may be periodic, sporadic, etc. An AADL process models an address space. In the most simple case, a process contains threads and data. Finally, processors, memories, buses and devices represent hardware components running one or several applications. Relevant to our study, memory components define storage area for code and data.

B. Modeling hardware memory architectures with AADL

```
memory address_space end address_space;
memory implementation address_space.board
subcomponents
  ram : memory memory_segment.impl {
    Base_Address => 016#00000000#;
    Byte_Count   => 016#00400000#; };
  devices_registers : memory memory_segment.impl {
    Base_Address => 016#FFFF0000#;
   --     ...   };
properties
  Base_Address => 016#00000000#;
  Byte_Count => 002#1#32;  -- 32 bits address bus
end address_space.board;
```

Figure 2. Memory architecture for a flat address space

Memory components can form a hierarchy, to reflect complex memory layouts and address space. In the figure 2, we define a flat address space of $4GB$ (32 bits) where the memory addresses ranging from `0` to `0x400000` are RAM memory, and addresses starting at `0xFFFF0000` represent I/O registers. This layout is used by embedded processors that map I/O to memory like PowerPC or LEON3 processors.

From this example, we may derive general principles to model memory architectures: the basic entity we use to model a memory layout is the segment. A segment is a memory component associated to a range of memory address, which is localized in the memory layout by the properties `Base_Address` and `Byte_Count`.

As we presented in section II, different kinds of memory exist. Thus, we introduce a new property set, called `Memory_Segment_Properties`. Property examples of this property set are:

- `segment_kind`, precises the segment type: address space or memory segment.
- An address space represents only a range of memory addresses. The property `address_kind` models the actual implementation of the address:
  - `physical`: the address selects directly a word stored in a memory component (in fact a semiconductor memory);
  - `logical`: the address selects a word stored in a memory component, optionally after an address translation. Some word may be bound to one or several logical addresses, depending of the address redirection;
  - `virtual`: the address selects a word stored in the main memory, or in a slower secondary memory device, such as hard drive for example;
  - `io_register`: words mapped at such address space are produced by an input/output instruction, and are expected to address registers to control or communicate with input/output devices.
- A memory segment represents a set of memory words accessible within a range of addresses. Memory words completely cover this address range i.e. one and only one word is mapped to each address.

In addition to these properties, we have to define additional legality rules to enforce that these memory layout models are correct. Hence, the following requirements shall be respected in all AADL models:

- Address spaces may contain subcomponents of address space or memory segment type. If a real storage is associated, directly or not, to a sub-range of an address space, such a range must be covered by a memory segment.
- Memory segments can only contain subcomponents of memory segment type. If subcomponents exist, they must cover all the address range of their segment container.
- Memory segments have access capabilities for read, write or execute operations, due to their realization technologies, and access restrictions due to operating system protection. The property `Access_Type` gives a list of the allowed operations on a segment. If a segment contains sub-segments, its access rights must be included in the access rights of all its sub-segments.

C. Modeling software view of memory with AADL

In addition to the modeling of hardware memory, we note we have also to model the software view of the memory. Any operating system uses the existing hardware memory to map its own internal structures (task control block, interrupts vector table, exception message, etc) into the memory, and then will allocate several memory areas to user code.

Compared to hardware memories, software-view of the memory can be made more specific: each segment can define its access mechanism as execute, read; or its kind (text for code, data for global variables) following linker scripts conventions.

The AADLv2 language and the ARINC653 annex document define properties for modeling both. Yet, we note the

following limitations:

- The defined memory kinds do not cover all possible combinations as mandated by real-time operating system linker scripts options: they only list read or write memories as available. We cannot tag a memory as heap, data, etc;
- The access mechanism lacks the execute flag for indicating a memory segment can hold code that can be executed. This feature is required to model precisely security operations;
- Finally, it lacks support for indicating different modes of operation of a memory: for example, the write accesses into a FLASH are limited, and their speed may change with respect to the memory state.

Thus, we propose to modify existing property sets to make the enumerator `Supported_Memory_Kind` and `Supported_Access_Type` part of the `AADL_Project` property set. This would allow the designer to enrich it depending on actual project needs.

We also propose to extend the property `ARINC653::Access_Type` to denote a list of `Supported_Access_Type` instead of a single one.

This allows for the modeling of complex memory layout as seen in real-time operating system kernel configuration, as in figure 3.

```
memory implementation memory_segment.system_image
subcomponents
 seg_text : memory memory_segment.impl {
    Base_Address => 016#001000#;
    arinc653::Memory_Kind => text;
    arinc653::Access_Type => (execute, read);
    Byte_Count => 3000; };
 seg_bss : memory memory_segment.impl {
    Base_Address => 016#003000#;
    arinc653::Memory_Kind => bss; };
 seg_data : memory memory_segment.impl {
    Base_Address => 016#002000#;
    arinc653::Memory_Kind => data_seg; };
 properties
   Base_Address => 016#001000#;
   arinc653::Memory_Kind => image;
end memory_segment.system_image;
```

Figure 3. Memory architecture for a VxWorks kernel image. Some properties, like *Byte_Count* in this example, may be omited if they are unknown in the early stages of the design process.

Like hardware memory layout, we have to define additional legality rules. We have three concerns to address:

1) Memory segments are consistent: no overlap of segments, contiguous segments, availability of at least one executable segment, etc;
2) The software view of the memory matches the hardware view: 1) the address of each segment matches existing hardware segment; 2) access mechanism (read-/write) are compatible;
3) The user code fits in the dedicated memory area.

D. Binding AADL hardware memory architectures to software views of memory

Once software and hardware views of the memory are defined, we have to bind them. Figure 4 shows how to take advantage of the `Actual_Memory_Binding` property to attach the software view of the memory to the process, and the hardware view of the memory to the processor.

```
system implementation vxworks.impl
subcomponents
  process1 :    process node_a.impl;
  logical_as  : memory address_space.vxworks;
  physical_as : memory address_space.mv162;
  processor1  : processor MC68040.impl;
properties
  Actual_Memory_Binding
     => (reference(logical_as))  applies to process1;
  -- Binding ''software-view'' memory to the process
  Actual_Memory_Binding
     => (reference(physical_as)) applies to  processor1;
  -- Binding ''hardware-view'' memory to the processor
  Actual_Processor_Binding
     => (reference(processor1)) applies to process1;
end vxworks.impl;
```

Figure 4. Mapping memories architecture

This model being complete, we can now check its consistency using the REAL checker. We discuss how to implement these additional legality rules in the next section.

IV. Defining and enforcing memory architectural constraints

A. The REAL annex language

REAL (Requirement Enforcement Analysis Language) is a domain-specific language, implemented as an AADL language annex. It aims at checking constraints enforcement on architectural descriptions at the specification step, saving significant time over verification at execution time. REAL pursues multiple design goals:

- Enabling easy navigation through AADL model elements. To do so, we defined REAL as a DSL (Domain specific Language) based on AADL meta-model concepts to ease writing of constraints, and on set theory to ease definition of constraints;
- Allowing for modularity through definition of separate constraints that can be later combined.
- Being integrated to the AADL as an annex language, so that constraints are coupled to models in the model repository.

REAL is based on set theory and associated mathematical notations. The basic unit of REAL is a theorem. A theorem verifies an expression over all the elements of a set that is called the range set.

Range set are defined using universal quantifiers ($\forall$, $\exists$) and AADL-specific keywords (like `property_exists` or `is_bound_to`) are used to fetch all entities matching a particular predicate. This allows one to build sets whose

elements are AADL entities (connections, components or subprogram calls). Verification can then be performed on either a set or its elements by stating Boolean expressions.

In order to write complex expressions, one can use predefined sets, which contain the instances of the AADL model of a given type, or build intermediary sets, using relations between elements of sets (e.g. returns the elements of the set A which are subcomponents of elements of set B).

Subtheorems calls can be used to extract values computed from range sets - thus allowing constructs like *get all the instances of threads which periodicity is equal to the minimum periodicity in the system*. These can also be used to define pre-required constraints on the model.

In [9], we have discussed the integration of REAL as an annex language in Ocarina. We demonstrated it is relevant to check specific constraints related to annex documents like the "Data Modeling Annex" or the "ARINC653 Annex" annex documents. These annexes were using simple predicates to check the validity of some combinations of properties.

In the following, we emphasize on REAL capabilities to check more complex predicates, based on subcomponents hierarchy (memory layout), and arithmetic on properties.

B. Constraints definitions

In the following, we review the set of constraints we defined to model software and hardware views of memory. We note there are four classes of constraints to satisfy: general considerations about the memory layout, constraints on alignment to match processor requirements, general software constraints and finally operating system specific constraints. We review each of them:

1) General memory layout constraints: These theorems check whether the memory layout as described by the AADL model is consistent with its definition. For example:

- A memory segment contains only memory segments (address space segments are forbidden). Figure 5 shows the implementation of the REAL theorem checking this property.
- Within a memory segment, the sum of the sub-memory segment size must be less than or equal to the memory segment container size. This requirement is restricted to a simple equality if the size of the segment and its sub-segments are known.
- Within an address space segment, the sum of size of all sub-segments (memory or address space) must be less or equal to the container size (see figure 6).
- Within a memory segment, sub-segments must cover all the address range. This requirement can only be checked if the segment and its sub-segments are located within the address range, i.e. their `base_address` and `Byte_Count` properties are known.
- Sub-segments do not overlap, and their address ranges are included in the address range of the memory segment or address space container.

```
theorem check_memory_segment_structure
 foreach seg in Memory_Set do
 sub_segments := {x in Memory_Set |
  property_exists(seg,
    "Memory_Segment_Properties::Segment_Kind") and
  property(seg, "Memory_Segment_Properties::Segment_Kind")
    = "memory"
  and Is_Subcomponent_Of (x, seg) };
 sub_memories := {x in sub_segments |
  property_exists(x,
    "Memory_Segment_Properties::Segment_Kind") and
  property(x, "Memory_Segment_Properties::Segment_Kind")
    = "memory" };
 check (sub_segments = sub_memories);
end check_memory_segment_structure;
```

Figure 5. First REAL theorem example. Within a memory segment that describes a range of memory words, sub-segments cannot be "address space" segment. The *sub_segments* set contains all the sub-segments of a given segment, including unexpected "address space" segments. The *sub_memories* set contains only its memory sub-segments. These two sets must be equal.

```
theorem check_address_space_size
 foreach seg in Memory_Set do
 sub_segments := {x in Memory_Set |
                    Is_Subcomponent_Of (x,seg) };
 var seg_size :=
      if (property_exists (seg, "Byte_Count"))
         then property (seg, "Byte_Count") else 0.0;
 check (cardinal (sub_segments) = 0
  or not property_exists (seg, "Byte_Count")
  or sum(property (sub_segments, "Byte_Count"))<=seg_size);
end check_address_space_size;
```

Figure 6. Second REAL theorem example. The sum of the size of sub-segments must be less or equal of the container segment size. The *sub_segment* set contains the list of sub-segments. The *seg_size* variable stores the size of the container segment. The reduction function *sum* computes the sum of the "Byte_Count" property values for all elements of the set sub-segments.

2) Software binding constraints: The theorems check that software components memory requirements match the resources provided by the hardware memory.

- Process, i.e. the software components that are included in the system, accesses the memory through its private logical address space. Then, to select a real memory word, a processor, maybe through a MMU, produces addresses within a physical address space; also, a processor must be bound to one physical address space description (see III-D). Thus, each process must be bound to one logical address space description and to one processor, and a physical address space bound to a processor. This ensures there exists a physical support for the logical memory space.
- All the access types supported by a segment must be also supported by all its sub-segments.
- In memory systems that do not involve memory address translation, all the memory words must have the same location in the logical and in the physical address space. Each memory segment localized in the logical space is bound to a memory segment described in the physical

```
theorem check_allowed_access
 foreach m in Memory_Set do
  good_segments:={x in Memory_Set |
   Is_Subcomponent_Of (x, m) and (
   not property_exists(x,"arinc653::Access_Type") or
   Is_In(property(m,"arinc653::Access_Type"),
         property(x,"arinc653::Access_Type"))) };
  segments:={x in Memory_Set | Is_Subcomponent_Of (x, m)};
 check ( cardinal(good_segments) = cardinal(segments) );
end check_allowed_access;
```

Figure 7. Third REAL theorem example. The REAL operator *Is_In* is used to check whether all the access rights of a segment are included in the rights of its sub-segments.

space, at an including address range.

- For a given process, the sum of the sizes of the text segments within its logical address space is greater than the sum of its thread code sizes.
- Text segments must be at least executable and readable.

3) Additional VxWorks constraints: Finally, these theorems model requirements are related to the VxWorks kernel:

- Within the system image segment, the sub-segments are ordered in memory as follow: text, data, and BSS.
- At the application start time, the sum of the thread stack size are less or equal than the system memory pool size.

C. Evaluation of the theorems

We note REAL provides appropriate support to model and then check all these constraints. We defined 21 theorems to be validated, forming 300 SLOCs (Source Line of Code)[1] We note these constraints can be easily implemented using simple queries while exercising navigation capabilities of REAL through the AADL model.

We evaluated all theorems using a model of the Motorola $MV162$ board, a board compatible with VxWorks $5.x$. We defined its hardware memory components and the software view of the memory. We also defined a toy software example that defines some memory requirements. We could check all theorems on this model.

Having a DSL greatly helps in structuring each theorem: accessors to be build queries on the model are direct basic constructs of the language. This reduces the learning curve, but also eases the review of each theorem to ensure it matches requirements informally defined.

Since REAL is implemented as part of Ocarina, the validation process is also fully automated, and quite efficient. On the model we tested, there is no noticeable performance hit when adding the validation of all 21 theorems.

V. Conclusion

Real-time embedded systems must demonstrate all resources are correctly dimensioned. This is quite challenging since embedded systems use particular memory types (volatile or permanent) and mappings.

[1]The AADL models and the REAL theorems are available at the following URL : http://pagesperso.univ-brest.fr/~rubini/Research/Memory_model

In this paper, we showed how the combined use of new AADL properties and REAL constraints can support the correct modeling and evaluation of memory layouts. We illustrated our approach using the VxWorks real-time operating system and a complete case study.

Evaluating memory requires a priori modeling of the target environment, and a posteriori evaluation of the memory consumption of each block (real-time operating system, subprograms, thread stack, etc). We use Ocarina to generate all glue code, and thus evaluate the cost of each software elements. By injecting these values in the model, and evaluating REAL constraints on the model, we close the loop and complete the memory consumption analysis.

Future work will consider more complex bindings for multi-core multi-memory systems. The use of such complex layout has a deep impact on performances. Modeling such architecture is required to extend the range of analysis supported by AADL.

References

[1] SAE, "Architecture Analysis and Design Language (AADL) AS-5506A," The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, Tech. Rep., January 2009.

[2] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 6–26, November 2002.

[3] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina." In 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Porto Allegre, Brazil, Jun. 2007.

[4] P. Farail, P. Gaufillet, A. Canals, C. L. Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel, "TOPCASED : An Open Source Development Environment for Embedded Systems," *Chapter 11, From MDD Concepts to Experiments and Illustrations, ISTE Editor*, pp. 195–207, September 2006.

[5] P. Dissaux, "Using the AADL for mission critical software development," *2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE Toulouse*, January 2004.

[6] SEI, "OSATE : An extensible Source AADL Tool Environment," *SEI AADL Team technical Report*, December 2004.

[7] O. Sokolsky, I. Lee, and D. Clark, "Schedulability Analysis of AADL models ." International Parallel and Distributed Processing Symposium, IPDPS 2006,, Apr. 2006.

[8] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand, "Investigating the usability of real-time scheduling theory with the Cheddar project." *Journal of Real-Time Systems, Springer Verlag*, vol. 43, no. 3, pp. 259–295, November 2009.

[9] O. Gilles and J. Hugues, "Expressing and enforcing user-defined constraints of AADL models," in *Proceedings of the 5th UML& AADL Workshop (UML&AADL 2010)*, University of Oxford, UK, Mar. 2010, pp. 337–342.