Modelling and Verification of Memory Architecture with AADL and REAL

AADL Standards Meeting, Paris

Stéphane Rubini and Frank Singhoff - University of Brest, UEB, France
{stephane.rubini,frank.singhoff}@univ-brest.fr
Jérôme Hugues - University of Toulouse, ISAE, France
jerome.hugues@isae.fr

June 8, 2011

Outline

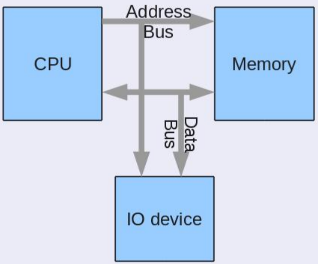
1. Why to model memory systems?
   - Hardware concerns: Memory components are not universals
   - The application's point of view
   - The OS point of view: memory layouts
2. Modelling of the memory
   - Modelling Guideline and Goals
   - AADL properties
   - Examples
3. Operations on the memory model
   - Ocarina, REAL annex
   - What can we check? REAL theorem examples
   - Configuring the deployment process
4. Conclusion and further works

Outline

1. Why to model memory systems?
   - Hardware concerns: Memory components are not universals
   - The application's point of view
   - The OS point of view: memory layouts
2. Modelling of the memory
3. Operations on the memory model
4. Conclusion and further works

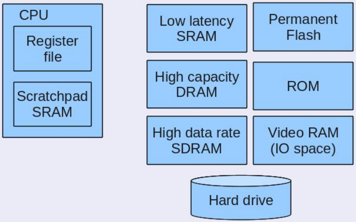
Why to model memory systems?

Architecture Von Neumann



But, today's memory systems are not so simple !

Few types of memories



The universal memory technology, suitable to all needs, does not exist.

Why to model the memory systems ? (2)

Design space of a memory system

- Performances (capacity, latency, data rate)
- Capabilities (persistence, supported operations, access type and ports)
- Other criteria (reliability, consumption, cost, robustness, . . .)

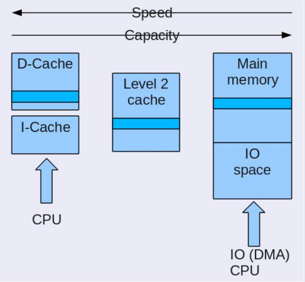
In a computer system (embedded), the memory is a resource which must be characterised.

- Design phase : complexity, cost
- Verification, validation and documentation : software usage (amount of resources, access time variability, technological limits, . . .)
- Support of the deployment phase

The memory hierarchy

- Memory wall, consumption ⇒ memory hierarchy, cache memories (1 or more levels)
- 1 variable = 1 or more physical storage locations
  ⇒ Coherency (read sensitive address, Direct Memory Access)
  ⇒ Predictability (how to evaluate the WCET ?)
  ⇒ Shared memory in multi-processor context

Memory levels



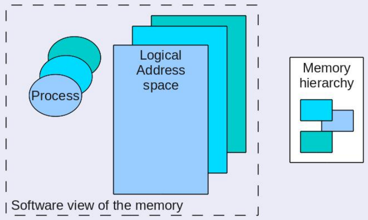
Needs to control how the data are managed within the memory hierarchy.

The application's point of view: logical and virtual address

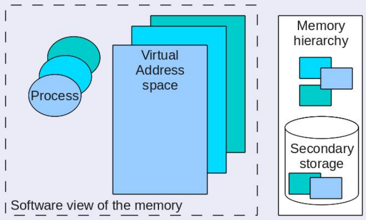
Memory as a private and homogeneous object...

- One logical address space per process (or task)
- Address translation: logical address $\rightarrow$ physical address in the real memory
- Process (or task) isolation

Logical address spaces



Virtual address spaces



OS point of view: memory layout examples

VxWorks (MC68040 board) Physical address space

sysMemTop()	System memory pool
Size of 0x1000	Interrupt stack
	Boot code and data
_end symbol	system image (Text, data, BSS)
0x1000	
	Initial stack
0x0900	
	Exception messages
0x0800	
	Boot line
0x0700	
	Reserved
0x0120	
	Interrupt vectors
0x0000	

Windows Embedded CE6 Logical address space

FFFF FFFF	
	Static Mapped Uncached
A000 0000	
	Static Mapped Cached
8000 0000	
	Shared System Heap
7000 0000	
	Memory Mapped Objects
6000 0000	
	Shared User Mode DLL
4000 0000	
	Per process virtual space
	code
0000 0000	reserved

Outline

Modelling Guideline

Hardware modelling

- Describe the physical features of memory components.
- Describe their mapping within the physical address space of a processor.

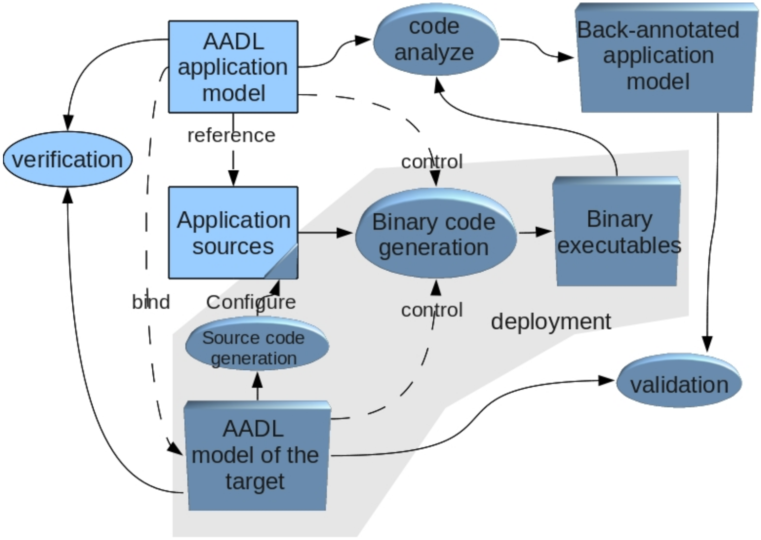
OS/software view modelling

- Describe the memory segments as defined by the OS and the applications. A memory segment is a range of address dedicated to a given usage, which must support a same set of operations.
- Describe their mapping within the address space (process memory layout) ; this layout may be different than the physical one.

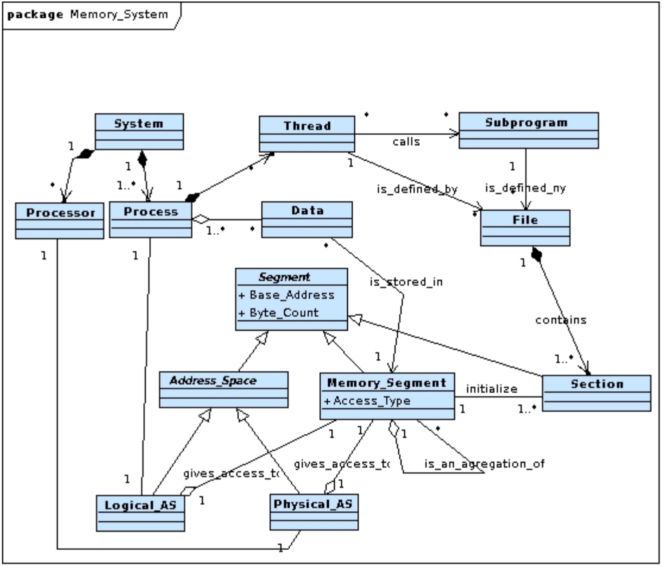
Modelling goal and Operation

1. Define the matching (static) between the software view and the hardware implementation of the target memory system.
2. Validate the software requirements and the physical features of the target memory features match.
3. Control the development chain for targeting a given memory system.
4. Estimate, from the model, the quantitative and functional memory requirements for a software component.
5. Back-annotate (update) the model with the memory usage information arises from the code generation.

Model-Driven Deployment



Modelling a memory system



Core AADL Properties useful for memory models

- `Base_Address : aadlinteger 0 .. Max_Base_Address` address of the first word in the memory
- `Source_Stack_Size: Size`, `Source_Heap_Size: Size`, `Source_Data_Size: Size`, `Source_Code_Size: Size` maximum size of respectively the stack, heap, data and code
- `Word_Space: aadlinteger 1 .. Max_Word_Space => 1` word alignment constraints
- `Word_Size: Size => 8 bits` smallest independently readable and writable unit of storage in the memory
- `Byte_Count: aadlinteger 0 .. Max_Word_Count` number of bytes in the memory

Properties from the ARINC653 annex

- `Supported_Access_Type : type enumeration (read, write, read_write)`
  `Access_Type : Supported_Access_Type`
- `Supported_Memory_Kind : type enumeration (memory_data, memory_code)`
  `Memory_Kind : Supported_Memory_Kind`

Limitations :

- The memory types do not encompass the diversity of the usage of the memory segments.
- The access types do not represent all the operational aspects bound to some memory technologies.

New properties

`segment_kind`, precises the segment type: (1) address space (2)memory segment or (3)file segment (section).

1. 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;
   - `virtual`: the address selects a word stored in the main memory, or in a slower secondary memory device;
   - `io_register`: address registers to control or communicate with input/output (read-sensitive locations).
2. A *memory segment* represents a set of memory words accessible within a range of addresses. One and only one word is mapped to each address.
3. A *file segment* represents a chunk of a binary file, used by program loaders to initialise a memory segment.

Additional properties

- `aadl_project.aadl`: access type and memory kind defined per project

```
Supported_Memory_Kind : type enumeration (text, stack, heap, bss, data_seg, memory_system_pool);
Supported_Access_Type : type enumeration (read, write, execute, slow_write, erase);
```

- `memory_segment_properties.aadl`: an additional property set

```
property set Memory_Segment_Properties is
  Supported_Address_Kind: type enumeration (physical_as, logical_as, virtual_as, io_register_as);
  Address_Kind : Supported_Address_Kind applies to (memory);
  Supported_Segment_Kind : type enumeration (memory_segment, address_space, file_segment);
  Segment_Kind : Supported_Segment_Kind applies to (memory);
  Page_Size : Size applies to (memory);
end Memory_Segment_Properties;
```

- `arinc653_properties.aadl`: an update proposition

```
-- Supported_Access_Type : type enumeration (read, write, read_write); into aadl_project
-- Supported_Memory_Kind : type enumeration (memory_data, memory_code);
  Access_Type : list of Supported_Access_Type applies to (memory);
```

Modelling of an address space

Layout of the physical address space of a board

```
memory address_space
properties
  Memory_Segment_Properties::Segment_Kind => 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
  Memory_Segment_Properties::Address_Kind => physical_as;
end address_space.board;
```

Layout of the memory

Layout of the *VxWorks* segment "system image"

```
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#004000#;
    arinc653::Memory_Kind => bss; };
 seg_data : memory memory_segment.impl {
    Base_Address => 016#006000#;
    arinc653::Memory_Kind => data_seg; };
 properties
   Base_Address => 016#001000#;
   arinc653::Memory_Kind => image;
end memory_segment.system_image;
```

Binding logical and physical view

```
system implementation vxworks.impl
subcomponents
  process1      : process node_a.impl;
  logical_as    : memory address_space.vxworks;
  physical_as   : memory address_space.mv162;
  cpu1          : processor MC68040.impl;
properties
  Actual_Memory_Binding => (reference(logical_as)) applies to process1;
  -- Binding ''OS-view'' memory to the process
  Actual_Memory_Binding => (reference(physical_as)) applies to cpu1;
  -- Binding ''hardware-view'' memory to the processor
  Actual_Processor_Binding => (reference(cpu1)) applies to process1;
end vxworks.impl;
```

Outline

REAL Language

- REAL (Requirement Enforcement Analysis Language), a Domain-Specific Language, implemented as an AADL language annex.
- Based on set theory and associated mathematical notations, a REAL theorem verifies an expression over all the elements of a set of AADL entities.
- Checking constraints enforcement on AADL architectural descriptions at the specification step:
  - Enabling easy navigation through AADL model elements;
  - Allowing for modularity through definition of separate constraints that can be later combined;
  - Being integrated as an AADL annex language, constraints are coupled to models.

What can we check on the memory model?

We defined 21 REAL theorems to be validated on a model.

1. General memory layout constraints check whether the memory layout as described by the AADL model is consistent with its definition:
   - usage of modelling patterns, size of segments, non-overlapping of segments.
2. Software binding constraints check that software components memory requirements match the resources provided by the hardware:
   - address space binding, access types, address translations, software segment sizes.
3. Alignment constraints ensure all memory boundaries are correctly aligned:
   - word size and page size alignments.
4. Specifics VxWorks constraints:
   - software segment sizes and order.

Check the memory model: example 1

Within a memory segment that describes a range of memory words, sub-segments cannot be "address space" segment.

Consistency check

```
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;
```

Check the memory model: example 2

All the access types supported by a segment must be also supported by all its sub-segments.

Consistency of the allowed access types

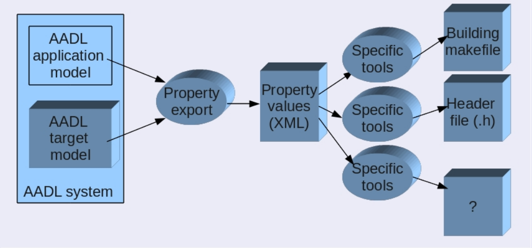
```
theorem check_allowed_access
 foreach m in Memory_Set do
  good_segments:={x in Memory_Set |
   Is_Subcomponent_Of (x, m) and (
   not property_exists(x,"arinc653::Access_Type") or
   Is_In(property(m,"arinc653::Access_Type"),
         property(x,"arinc653::Access_Type"))) };
  segments:={x in Memory_Set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

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.

Control of the binary code generation

1. Configuring the building tools
   - Makefile generator
   - Tuning of the source codes (for example C preprocessor symbols)

   ⇒ export of AADL properties into a XML file
   + simple specific tools (PERL scripts for example)



2. Control of the link editor (linker script)

Linker script: what could be done at that level (1)

- Collect all the sources files defined in the model

AADL application model

```
subprogram Prod_Spg
 properties
  source_text => ("prod.c");
end Prod_Spg;
subprogram Cons_Spg
 properties
  source_text => ("cons.c");
end Cons_Spg;
...
thread implementation Producer.impl
 calls
  Mycalls:{P_Spg:subprogram Prod_Spg;};
 properties
  source_text => ("prod.c", "util.c");
end Producer.impl;
thread implementation Consumer.impl
 calls
   Mycalls: {C_Spg : subprogram Cons_Spg;};
 properties
   source_text => ("cons.c", "util.c");
end Consumer.impl;
```

Linker script

```
INPUT prod.o
INPUT cons.o
INPUT util.o
SECTIONS {
 .text : {
   prod.o cons.o util.o { .text }
   * { .text }
 }
 .data  : {
   * { .data }
 }
 .bss : {
   * { .bss } * { COMMON }
 }
}
```

Linker script: what could be done at that level (2)

- Validate the binary codes with regards to the hardware memory usage

AADL target model

```
memory implementation address_space.mv162
 subcomponents
   ram : memory memory_segment.impl {
      Base_Address => 016#00000000#;
      Byte_Count => 016#00400000#;
      arinc653::Access_Type => (read, write, execute);
   };
   rom : memory memory_segment.impl {
      Base_Address => 016#FF800000#;
      Byte_Count => 016#00040000#;
      arinc653::Access_Type => (read, execute);
   };
 properties
   memory_segment_properties::Address_Kind => physical_as;
 end address_space.mv162;
```

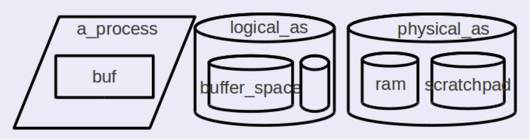
Linker script

```
MEMORY {
    ram(wx)  : ORIGIN=0x00000000 LENGTH=0x00400000
    rom(rx)  : ORIGIN=0xFF800000 LENGTH=0x00040000
}
```

Linker script: what could be done at that level (3)

- Control the binding of software structures to hardware memory resources

AADL model




```
system implementation sys.impl
subcomponents
 a_process    : process process.impl;
 logical_as   : memory address_space.os;
 physical_as  : memory address_space.a_target;
properties
 Actual_Memory_Binding => (reference(logical_as.buffer_space))
                          applies to a_process.buf;

 -- but I cannot do that with Actual_Memory_Binding ...
 Actual_Segment_Binding_ => (reference(physical_as.scratchpad))
                          applies to logical_as.buffer_space;
end sys.impl;
```

Linker script

```
MEMORY {
 ram (rwaix)      : ORIGIN = ...
 scratchpad (wx): ORIGIN = ...
}
SECTIONS {
  ...
  .buffer_space : {
    PROVIDE(_start_buffer_space = .);
    buffer.o (.data)
    # from a Byte_Count property
    . = 0x10000;
  } > scratchpad
}
```

Outline

Conclusion

- We have defined a set of AADL properties to describe more precisely a system memory layout;
- The REAL DSL has been shown to be useful in verifying the internal consistency of the memory model;
- A precise memory model $\Rightarrow$ Characterizing of Off The Shelf Components

Remains to be done

- Control the development chain from the memory model $\rightarrow$ generation of linker scripts and property exports (*Ocarina* backends)
- Back-annotate the model from the produced code
  - size and localisation of the memory segment
  - localisation of the system objects
  - estimation of the memory footprints of a system.
- And in medium terme: Cache Conscious Data Placement (CCDP)