Design and Analysis of Multi-Core Architecture for Cyber-Physical Systems

Julien Delange, Peter Feiler

Design and Analysis of Multi-Core Architecture for Cyber-Physical Systems

Julien Delange and Peter Feiler

Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA15213-2612 USA
jdelange@sei.cmu.edu, phf@sei.cmu.edu

Abstract. Cyber-Physical Systems are becoming software intensive, collocating many functions on a single processor and requiring a significant processing capacity which increased over the years. In recent years, improving processing performance has been achieved by adding more processing cores on the same chip rather than increasing its frequency. This new design also introduces issues: interaction among cores may impact software performance and might also arm software isolation layers, such as the one defined in ARINC653. For that reason, software using multi-core architecture must be carefully designed and specified with hardware and software aspects. This would help to analyze the system and detect potential design issue. This paper proposes an approach to represent multi-core architectures and their association with software artifacts, such as the ones used for cyber-physical systems (e.g., the ARINC653 platform). For that purpose, we use the AADL language and define specific modeling patterns with new properties.

Keywords: multi-core, ARINC653, model-based engineering, AADL

1 Introduction

1.1 Context

Cyber-physical systems (CPS), as used in military or avionics domains, become increasingly software-reliant and operate many functions. Software defects may have minor (temporary disturbance) or major impact (mission failure), depending on execution environment, system criticality, and other criteria. Enforcement of continuity of operation is difficult because software components interact and may impact each other. Actual military aircraft systems contain more than 1.7M lines of code [1], and new civil aircraft feature more than 4M lines of code [2]. Thus, design and validate software by using traditional design techniques is an overwhelming task.

One solution consists of isolating software functions according different criteria (criticality level, component provider, etc.) so that one cannot interfere with another. Thus, each function can be analyzed separately, easing the overall validation process. For that purpose, the ARINC653 [3] standard (for avionics systems) defines a specific runtime that isolates system functions from each other.

However, as such systems continue to grow and they now require a significant processing power. This trend was traditionally addressed by upgrading hardware and increasing the processor speed, but is now achieved by using multi-core architectures with

processors containing several processing cores on the same chip. As these cores share resources (memory, caches, buses), they may interact each other so that one function executing on one core may affect others executing on remaining cores. This can impact the underlying software architecture by breaking software isolation as defined in ARINC653 [3].

1.2 Problem and Approach

Software executing on one core may generate disturbance and impact functions executing on other cores, even when using specific runtime with isolation services such as ARINC653 [3]. This may affect different business drivers: from a reliability perspective, a function issuing several requests on a shared bus may postpone the execution of a function that is using the same resource so that it overruns its deadline. From a security perspective, a function classified at a low security level may also read data from a software component classified at a higher security level if their processing cores use shared memories. These issues are dependent on both software and hardware architectures as well as the system business goals. Thus, there is no standard and general solution, and these issues must be evaluated on a per-project basis.

For that reason, modeling multi-core processors and their association to the software and architecture (such as ARINC653 [3]) would help system designers to review, analyze, and detect potential issues when using multi-core processors. This would also help represent different system criteria, such as

- use of shared resources (caches, buses, devices, etc.)
- allocation strategies of software on multiple cores
- impact on software scheduling
- allocation of mixed-criticality levels on several cores

We propose an approach to model such architectures with appropriate patterns using the Architecture Analysis and Design Language (AADL) [4], a language that already supports analysis tools for CPS design and analysis. It is compatible with the core language and its extensions (such as the ARINC653 annex[5]) so that use of multi-core architectures does not require a new model and can be added as a component refinement. The paper is organized as follows:

1. Overview of the related work: CPS software architectures (such as the ARINC653 [3] partitioned architecture), multi-core processors, and AADL
2. Modeling partitioned multi-core systems with AADL
3. Application with a case study
4. Conclusion and perspectives

2 Related Work

2.1 Multi-Core Architectures

Until recently, increasing processing capability of processors consisted of increasing their frequencies or adding more transistors on the same chip. This trend has been used

for years, but the process has reached its limits; the number of transistors being collocated in the same area have some physical limitations. One solution consists of reusing the symmetric multiprocessing (SMP) architecture that executes software functions in parallel. Collocated processing cores also share resources (buses, memory, caches, etc.), which may create disturbance between concurrent cores.

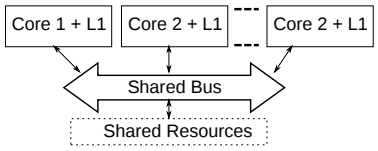

Fig. 1. Example of a multi-core architecture

Most multi-core processors have their own cache (L1) and share resources (such as the second cache (L2)) through one or several dedicated buses, as shown in Figure 1.

Unlike single-core architecture, applications performance with multi-core processors mostly depends on software: a specific scheduler for multi-cores, data dependencies between tasks, use of lock mechanisms between concurrent threads, etc. Sharing resources may create problems [6–8], especially if a core issues many memory access requests that increase bus latency.

On the other hand, having several cores may be of special interest, especially when isolating applications (such as in ARINC653 [3]). Each software partition may be allocated to a separate partition, ensuring time isolation. However, this may create disturbance, especially because some resources are still shared (such as the L2 cache) and may break the partitioning policy. Several research efforts have already been performed in that context.

2.2 ARINC653 and Partitioned Runtime

ARINC653 [3] is an industrial avionics standard published by Aeronautical Radio. It defines a set of services and a standardized interface to isolate software applications into partitions. Each partition is independent with its own resources and appears to be running as if it were allocated to a single processor. The isolation policy is enforced by an underlying kernel (as shown in Figure 2) in terms of space and time:

- **Space**: Each partition is associated to a unique address space (memory segment) to store code and data. One partition cannot read/write other segments.
- **Time**: Time frames are allocated to each partition to execute their threads and cannot be overrun. In addition, each partition uses its own scheduling policy because the overall system has two scheduling layers: kernel level (scheduling of partitions) and partition level (scheduling of partitions tasks).

Most of the time, the kernel-level scheduler uses a static scheduling protocol that executes partitions at a fixed rate called the *major time frame*. An example of such isolation is shown in Figure 3 with two partitions being executed consecutively according to a fixed and pre-configured timeline.

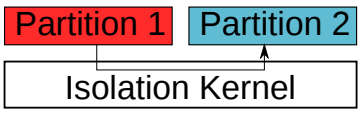

Fig. 2. Architecture of an ARINC653 system

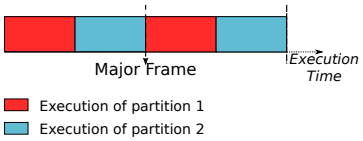

Fig. 3. Scheduling of ARINC653 partitions

Because partitions are isolated into their memory space and cannot share memory, communications are implemented with a dedicated inter-partition communication service. Cross-partition communication channels are pre-configured at design time so that no partition can create a communication path that has not been explicitly declared.

Finally, the standard defines a service to detect, recover, and isolate fault occurrence and propagation. A fault can occur at different level: kernel, partition, or task. Upon fault occurrence, a dedicated function is executed to fix the error.

ARINC653 defines a set of common faults/exceptions (that can be extended by each specific ARINC653-compliant OS) that may arise in the system. They range from software fault (e.g., division by zero) to hardware fault (e.g., loss of power). The system provides the flexibility to detect and recover fault at different levels so that the system designer can easily implement a system health-monitoring policy.

As new systems are currently switching from single- to multi-core processors, one major concern is to ensure isolation enforcement with new hardware components. To do so, users need methods and tools to model an ARINC653 software architecture (with partitions, tasks, communication channels, etc.) and its association on the hardware platform using either using single- or multi- core processors. Thus, this representation could then be supported by analysis tools to check scheduling feasibility and any potential issue that may break ARINC653 isolation principles.

2.3 AADL

AADL [4] is a modeling language standardized by the Society of Automotive Engineers (SAE). It defines a notation for describing architecture hardware and software concerns within a single and consistent model. The language uses a component-based approach, with each system artifact being defined as a component. The core language specifies several components (processor, process, bus, thread, etc.). Each one defines its interfaces and characteristics. Components are aggregated (with one component containing subcomponents; for example, a task includes several subprograms or a process contains several tasks) and connected (for example, a processor accesses a bus or a memory, or two tasks are connected to exchange values/data). Component-specific characteristics are parameterized using properties (for example, period of a task, size of a memory, speed of a processor). The components composition constitutes the system architecture.

The language is extensible; users may adapt it to their needs using two mechanisms:

1. **User-defined properties**. New properties can be defined to extend the components' characteristics. This is a convenient way to add specific architecture criteria into the model (for example, criticality of a subprogram or task)

2. **Annex languages**. Third-party languages [9, 10] can be attached to AADL components to augment their description and specify additional characteristics and requirements (for example, specifying the component behavior [10] by attaching a state-machine). They are referred to as annex languages, constituting additional pieces of information related to the component.

AADL provides two views to represent models:

1. The **graphical view** outlines components' hierarchy and dependencies (bindings, connection, bus access, etc.). While not providing all architecture details, it is very useful for documenting the architecture.
2. The **textual view** shows the complete model description, with component interfaces, properties, and languages annexes. It is appropriate for users to capture system-internal details and for tools to process and analyze the system architecture from models.

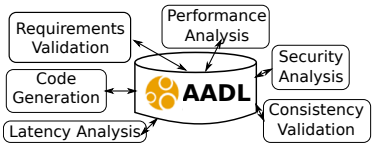

Fig. 4. AADL ecosystem

The language is supported by several tools for analyzing system requirements or producing the implementation, as shown in Figure 4. AADL has already been successfully used to validate several quality attributes such as security [11], performance, and latency [12]. Supporting analysis functions have been implemented in the Open Source AADL Tool Environment (OSATE) [13], an Eclipse-based tool that supports AADL.

A standard annex document (the ARINC653 annex [5]) provides guidance to model avionics architecture, as defined in the ARINC653 [3]. However, modeling of multi-core applications has not been yet explored and no modeling pattern has been proposed. This makes the design of such systems more difficult because each model would represent the architecture in a different manner that is not suitable for all analysis tools.

Our contribution consists of proposing AADL modeling patterns for multi-core processors that are consistent with the existing standards documents, including the ARINC653 annex [5]. Keeping compatibility with existing modeling patterns would also help the transition of models using single-core to multi-core processors and keep the potential changes to the minimum.

3 Model of Partitioned Multi-Core Architectures

3.1 Multi-Core Processors

Processor cores are hardware parts similar to physical processors: they access a bus to read/write into shared resources, have their own private resources, etc. Thus, a core

is associated with an AADL `processor` component and a multi-core processor with an AADL `system` component containing multiple AADL `processor` subcomponents, each one representing a separate core. This modeling approach also provides flexibility: an AADL `system` can contain other components to represent heterogeneous architectures, such as system on chip (SoC), that include a network controller, USB bus, and other shared resources.

In order to specify processor requirements, we introduce a new dedicated AADL property to describe the following processor characteristics:

- **Speed**: using various metrics such as core frequency or instructions per second
- **Power Dissipation**: using a range value for each core
- **Power Consumption**: using a range value for each core, reflecting the power used at full speed or when using energy-saving mode

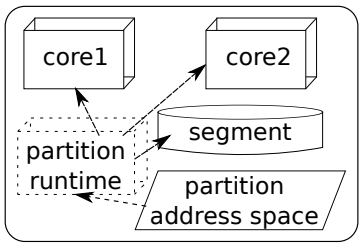

Fig. 5. First modeling pattern for partitioned multi-core architectures

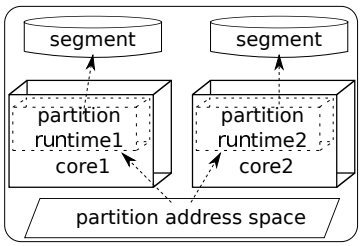

Fig. 6. Second modeling pattern for partitioned multi-core architectures

3.2 Binding with Isolation Runtime

The ARINC653 annex [5] for AADL mandates to model partitions at runtime using a `virtual processor` within a `processor` component. Then, the partition address space (AADL `process` that abstracts partition code and data) is associated with this runtime using the regular AADL property (`Actual_Processor_Binding`). Representing partitioned architectures on multi-core processors uses the same approach: the partition's runtime (`virtual processor`) is added to each core (`processor`) and its address space (`process`) bound to it.

However, in the context of multi-core, a partition can be associated to several cores according to a particular policy (a partition being relocated from one core to another upon failure, redundancy, etc.). To specify this particular requirement, the following approaches are proposed:

1. Define the partition runtime (`virtual processor`) separately and bound it to the partitions' supporting cores (`processor`) with the `Actual_Processor_Binding` or `Allowed_Processor_Binding` properties. Then, we also associate it with the partition address space (`process`). This approach assumes that the underlying execution platform (Operating System) can relocate the partition execution context

from one core to another. For one partition executed on two cores, this modeling pattern would use one `virtual processor` (associated with the partition address space – an AADL `process` component) bound to two `processor`s, as shown on Figure 5. We also associate the memory segment to the `virtual processor`, not the `process`, which means that when relocating the partition, the same memory segment would be used.
2. Define as many runtimes (`virtual processor`) as required on each core (`processor`) and associate the partition address space (`process`) with each of them. This approach assumes that the underlying OS also isolates the different execution runtimes in separate memory segments (each one being associated with a separate `virtual processor`) and would relocate the context between partitions' runtimes when relocating the partition from one core to another. For one partition executed on two cores, this modeling pattern would use two `virtual processor`s, each one contained within a core (`processor`). Then, the partition address space (`process`) is associated with both `virtual processor` components, as shown in Figure 6.

Users may distinguish the cores that can execute the partition from the one that is actually executing it. For that, the language defines two properties:

1. `Allowed_Processor_Binding` provides the list of cores that may execute the partition. These cores may be selected when relocating the partition.
2. `Actual_Processor_Binding` defines the list of cores that execute the partition. As properties may be mode-specific, we can define bindings for each component mode, showing the different deployment strategies according to the system state.

4 Case Study

We established a model of a multi-core partitioned architecture. The graphical model is shown in Figure 7. The textual model is not included in this paper but is available online at the official OSATE examples repository [14] and documented on the official AADL wiki [15]. This architecture defines an architecture that is representative of an acquire-and-control system with three partitions:

1. **Acquisition** that retrieves raw values from sensors
2. **Processing** that receives the data and does some computation (filtering bad data, checking values according to predefined bounds, etc.)
3. **Control** that activates motors and/or thrusters according to the received command

These partitions are then associated with a memory segment (space isolation of an ARINC653 architecture) and a dedicated runtime (dashed boxes in the middle-right). Then, these components are associated with the processor, which is represented using a single `system` component:

– The hardware memory component that hosts partitions memory segments is then connected to the processor using a `bus access`, which is shared among all cores.

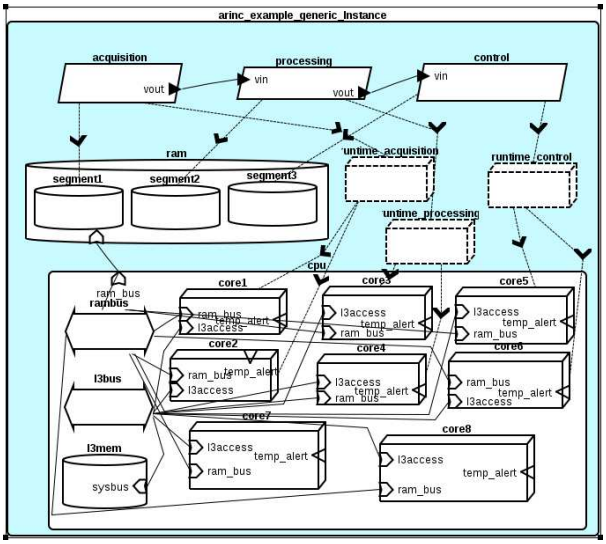

Fig. 7. AADL model of our case-study

– Each partition is replicated on two cores. To capture this requirement in the model, we associate each partition runtime `virtual processor` with two cores (`processor`). When a failure is detected, the underlying operating system can be relocated from one core to another. In this architecture, six cores are used (two per partition) while two are not associated with any partition.

```
processor e500mc
features
    l3access     : requires bus access l3cachebus;
    ram_bus   : requires bus access ram_bus;
    temp_alert   : in event port;
modes
    Nominal   : initial mode;
    Degraded     : mode;
properties
    Processor_Properties::Speed =>
      [Freq => 0.0 Mhz .. 750.0 MHz; Kind => (FREQ);] in modes (Degraded);
    Processor_Properties::Speed =>
      [Freq => 0.0 Mhz .. 1.5 GHz; Kind => (FREQ);] in modes (Nominal);
end e500mc;

processor implementation e500mc.generic
subcomponents
    dcache   : memory Multi_Core_Cache::cache {Byte_Count => 32_000;};
    icache   : memory Multi_Core_Cache::cache {Byte_Count => 32_000;};
    l2cache  : memory Multi_Core_Cache::cache {Byte_Count => 128_000;};
end e500mc.generic;
```

Listing 1.1. Model or a core of the P4080 processor

The processor represented is a P4080, an SoC that integrates eight cores on a single chip and provides dedicated functionalities with respect to software partitioning [16]. The textual AADL model representing a core is shown in Listing 1.1. The main benefits of using the `processor` AADL component to represent the core are shown here by defining bus accesses (use of shared resources) and additional properties (as the speed that may vary according to internal modes).

5 Conclusion

Cyber-Physical Systems (CPS) become software intensive and requires more processing capacity. For that reason, use of multi-core processors will be mandatory. As this type of architecture may impact software execution, new technologies and tools are required to identify and avoid potential defects.

This paper presents an approach for designing multi-core architectures using AADL. We propose modeling patterns for describing multi-core processors with respect to their attributes. This component aggregation fits with methods and approaches specific to CPSs (such as ARINC653, an avionics standard for building partitioned architecture).

The resulting models would be used as inputs for analysis tools to simulate system execution and detect issues related to CPSs requirements, such as scheduling or security. For example, one application would consist of exporting the AADL notation to a scheduling analyzer to check the enforcement of timing constraints on each core, ensuring that task deadlines can be met according to a specific software allocation on each processor core.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

1. Wikipedia: Lockheed Martin F-22 Raptor. http://en.wikipedia.org/wiki/Lockheed_Martin_F-22_Raptor
2. Military Aerospace: DO-178C nears finish line with credit for modern tools and technologies. http://www.militaryaerospace.com/articles/print/volume-21/issue-11/technology-focus/do-178c-nears-finish-line-with-credit-for-modern-tools-and-technologies.html (May 2010)
3. Airlines Electronic Engineering: Avionics application software standard interface. Technical report, Aeronautical Radio, Inc (1997)
4. SAE International: AS5506 - Architecture Analysis and Design Language (AADL). (2012)
5. SAE International: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2. (2011)
6. Paolieri, M., Quiñones, E., Cazorla, F.J., Bernat, G., Valero, M.: Hardware support for WCET analysis of hard real-time multicore systems. SIGARCH Comput. Archit. (June 2009) 57–68
7. Schliecker, S., Rox, J., Negrean, M., Richter, K., Jersak, M., Ernst, R.: System level performance analysis for real-time automotive multicore and network architectures. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **28**(7) (2009)
8. Anderson, J.H., Calandrino, J.M., Devi, U.C.: Real-time scheduling on multicore platforms. In: Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE, IEEE (2006) 179–190
9. Rugina, A.E., Kanoun, K., Kaaniche, M.: An architecture-based dependability modeling framework using AADL. In: 10th IASTED International Conference on Software Engineering and Applications (SEA'2006). (2006)
10. Frana, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex – experiments and roadmap. Engineering Complex Computer Systems (July 2007) 377–382
11. Hansson, J., Greenhouse, A.: Modeling and validating security and confidentiality in system architectures. Technical report, Carnegie Mellon Software Engineering Institute (2008)
12. Feiler, P., Hansson, J.: Flow latency analysis with the Architecture Analysis and Design Language (AADL) - tn cmu/sei-2007-tn-010. Technical report, Carnegie Mellon Software Engineering Institute (December 2007)
13. Carnegie Mellon Software Engineering Institute: Open Source AADL Tool Environment. Technical report (2006)
14. Carnegie Mellon Software Engineering Institute: AADL examples repository. https://github.com/osate/examples/
15. Carnegie Mellon Software Engineering Institute: AADL official wiki. https://wiki.sei.cmu.edu/aadl/index.php/Main_Page
16. Freescale: P4080 QorIQ integrated processor hardware specifications (2013)