Open Archive TOULOUSE Archive Ouverte (OATAO)
OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 12944

To link to this article : DOI :10.1007/978-3-319-04891-8_9
URL : http://dx.doi.org/10.1007/978-3-319-04891-8_9

To cite this version : Herbegue, Hajer and Filali, Mamoun and Cassé, Hugues *Formal Architecture Specification for Time Analysis*. (2014) In: International Conference on Architecture of Computing Systems - ARCS 2014, 25 February 2014 - 28 February 2014 (Lubeck, Germany).

Any correspondance concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

Formal Architecture Specification for Time Analysis

Hajer Herbegue, Mamoun Filali, and Hugues Cassé

CNRS-IRIT, Université de Toulouse
Toulouse, France
firstname.lastname@irit.fr

Abstract. WCET calculus is nowadays a must for safety critical systems. As a matter of fact, basic real-time properties rely on accurate timings. Although over the last years, substantial progress has been made in order to get a more precise WCET, we believe that the design of the underlying frameworks deserve more attention. In this paper, we are concerned mainly with two aspects which deal with the modularity of these frameworks. First, we enhance the existing language Sim-nML for describing processors at the instruction level in order to capture modern architecture aspects. Second, we propose a light DSL in order to describe, in a formal prose, architectural aspects related to both the structural aspects as well as to the behavioral aspects.

Keywords: Hardware, microarchitecture, pipeline, WCET, architecture language, formalization, constraints.

1 Introduction

System-on-chip and processor modeling methodologies are continuously improved to overcome the increasing complexities of critical embedded systems. Designers have to deal with complex features of new architectures and develop application/domain specific processors. It is highly desirable, as intending to reduce the costs and time-to-market, that the software design tool can be synthesized automatically from high level processor specifications. In this scope, there is a surge in architecture description languages. ADLs have been used in retargetable tools generation, design space exploration, hardware synthesis, verification and time analysis [14,16]. In order to have reliable and powerful design and analysis flows, ADLs have to convey the informal processor specification provided by vendors to the development tools, as closely as possible. Furthermore, the validation (and verification) is an important task in the system-on-ship design process, that ensures the correction of the system with respect to the correctness requirements and real-time constraints. Such a task is arduous because of the architecture complexity and lack of clear and explicit syntax and semantics in currently used architecture languages. Indeed, to ensure the completeness of the architecture requirements at the design stage, it is essential to have a precise and formal processor specification. ADL-driven flows for worst case execution time (WCET)

analysis, like the OTAWA framework [4], need a clear and explicit syntax and semantics for the architecture description to provide the required accuracy.

At the present time, OTAWA allows the time analysis using a constraint-based approach, in addition to the validation and the animation of time results [12] and the generation of fine-grained simulators at pipeline level. In this paper, we enhance the OTAWA work flow with a logic-based description that formalize the architecture properties. This description is used, with the architecture model described in the Sim-nML language, to generate a constraint-based description for the WCET computation. We first present how advanced architecture features, specially instruction with complex behaviors, can be handled by the OTAWA ADL description. Second, we give, in a formal prose, the operational properties of the hardware components and the instruction set. These properties describe the instructions behavior regarding to resources allocation, dependencies, parallel execution, etc. This description also provides a good basis for formal verification of time analysis methods.

The paper is organized as follows. Section 2 gives an overview of the ADL-based approach and our contribution. In Section 3, we present the Sim-nML language and its extension to describe advanced features of real-life architectures. In Section 4, we present the logic-based description, illustrated through a processor use case. In Section 5, we present an overview of related works and draw a comparison between their respective description languages. Section 6 concludes the paper.

2 ADL-Based Approach for Time Computation

OTAWA [4] is a framework dedicated to WCET computation of a program executed on a given processor. The time analysis is based on an abstraction of the target architecture and the binary. The WCET of a program corresponds to the execution time of the longest execution path, which is identified on the control flow graph (CFG) of the program. An execution path is a sequence of code snippets, called *basic blocks*. The WCET is a function of the time cost of the basic blocks and their execution counts [18]. In this paper, we focus on the computation of the basic block execution time. The pipeline analysis consists of modeling the instruction behavior of the pipeline and evaluating the impact of the hardware features on the instruction execution times [17]. The framework OTAWA was enhanced with an ADL-based approach [11] that aims at computing the time cost of a basic block considering the pipeline features. The carried analysis considers as input (1) the program binary, (2) the basic block as an instruction sequence and (3) the architecture description in the Sim-nML language [10] (see figure 1). Sim-nML was extended to support, in addition to the ISA description, the micro-architecture description of the target processor. The architecture description includes the resources accessed by the instructions such as pipeline stages, buffers, etc. and the execution model of instructions. The execution model describes the instruction behavior in terms of resource allocation. From the Sim-nML language and the binary, we generate an internal

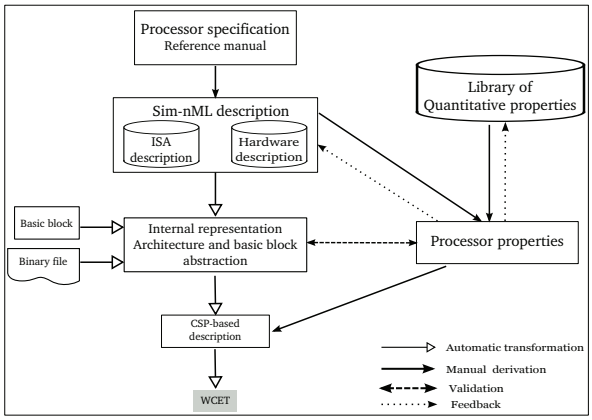

Fig. 1. ADL-based flow for time analysis

representation of the architecture and of the basic block. Then, a constraint-based description is automatically generated from the execution models of the instructions and the pipeline description. These constraints are combined to formulate a constraint satisfaction problem (CSP) [8], whose resolution provides the time cost of the basic block.

In this paper, we intend to provide a formal description of the architecture constraints, in which we can express the architecture elements and properties, regardless of the resolution method and the language used later for time computation. We propose a domain specific language that allows a logic description of the architecture properties. The idea is to provide a library in which the architect-user can find a set of reusable quantitative properties that assist him in (1) the definition of architecture high-level constraints that would be used for time analysis and (2) the validation of the correctness and the consistency of the architecture model with respect to the initial specification. Indeed, according to the initial Sim-nML description and what is provided in the properties library, the user defines a set of properties that will be used further to compute WCETs.

3 The Sim-nML Description Language Extension

Sim-nML [10] is a hierarchical and a highly structured language that describes the processor at instruction level, using an attributed grammar. The instructions and the addressing modes are described using pre-defined attributes. The *syntax* attribute defines the assembly representation of the instruction. The attribute *image* gives the binary representation and *action* defines the semantics of the instruction (register transfer). Our extension to the Sim-nML language [11] allows the definition of the processor resources and the execution model of the instruction set, giving how and when the resources are accessed by each instruction. The properties of the hardware components are specified as attributes. So, we can declare stages, buffers, registers and memories. Concerning stages, we can specify out-of-order execution, superscalarity, cache characteristics, if relevant for time analysis, etc. The instructions definition is extended with an attribute

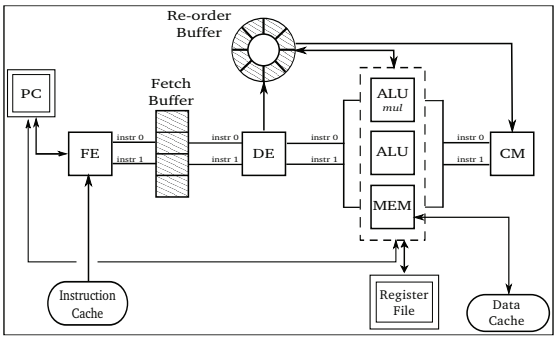

Fig. 2. An out-of-order superscalar processor

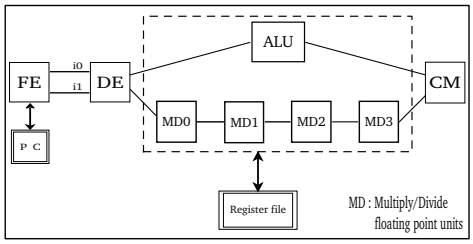

Fig. 3. Floating point pipeline

uses that describes the execution model. In fact, to start execution on a stage, an instruction has to wait for its resources to be available, including the operands, the executing stage, the memory, etc. Therefore, the execution time of an instruction is impacted by the resources state. The *uses* attribute defines, in a timed sequence called *clause*, the resources required by an instruction in each step of its execution. A sequence is defined using commas. Every clause in a sequence represents a step of the instruction execution. In every step, one or more resources are required, and access can be in read or write mode. Parallel access is expressed by the operator &. Access to some resources takes a fixed duration t, that is specified as $\#\{t\}$. An example of a 2-scalar out-of-order processor is illustrated in figure 2 and described in Sim-nML in lines 1-12 of listing 1.1. The language extension was amenable to describe some complications of instruction set architectures. In next paragraphs, we show how we extend the Sim-nML in order to handle pipelines with complex and long-running instructions.

Not-fully symmetric ALU. Specialized functional units are designed for specific operation patterns to achieve shorter delays. In the processor of figure 2, we assume that the first ALU occurrence implements a multiplier component executing multiply operations and ordinary data processing operations. Usually, processors include only one specialized ALU because it is expensive and there are more additions than multiply operations. Thus, arithmetic instructions are executed by any of the ALU. So no occurrence is specified in the ADD execution model (lines 18-19 of listing 1.1). While, in the execution model of MUL, we specify $ALU[0]$ as the required stage occurrence (lines 23-24 of listing 1.1). This will be used when scheduling instructions to be issued to the ALU units. Different latencies has been associated to the execution of the ADD and MUL instructions on the ALU unit.

Listing 1.1. Sim-nML processor description

```
1  stage FE , DE , ALU[2] , MEM , CM
2  extend FE , DE , CM
3      capacity = 2          // super-scalarity degree
4      inorder = true        // in-order stages
5  extend ALU , MEM
```

```
6       inorder = false // out-of-order stages
7
8   buffer FBuf [4] , RoB [8] // Fetch Buffer and Re-order Buffer
9
10  reg PC [1,card(32)] // 32-bit PC register
11  reg R [16,card(32)] // 16 registers of 32 bits
12  mem M [32,card(8)]// a memory of 2^32 8-bit words
13
14  op ADD (rd:card (4),rs:card (4),rn:card (4))
15   syntax = format ("add r%d r%d r%d",rd,rs,rn)
16   image = format ("00%2b%2b%2b",rd,rs,rn)
17   action = {R[rd] = R[rs] + R[rn] ;}
18   uses= FE & FBuf & PC.read, DE, ALU & RoB & R[rs].read &R[rn].read
19          & R[rd].write#{1}, CM
20
21  // Multiply instruction executed on the specilized ALU
22  op MUL
23   uses= FE & FBuf & PC.read, DE, ALU[0] & R[rd].write & R[rm].read
24         & R[rs].read & RoB#{5},CM
25
26  // load multiple instruction
27  op load_multiple (rlist: card(16) , rn: card(4) )
28   uses = FE & PC.read & FBuf, DE ,
29  if rl<0..0>==1 then MEM & M.read & R[rn].read & R[0].write & RoB
       endif,
30  if rl<1..1>==1 then MEM & M.read & R[rn].read & R[1].write & RoB
       endif, ..,CM
31
32  // branch instruction dumped after decode
33   var taken[1,u1]
34  extend B_Cond
35   uses = FE & PC.read & FBuf,DE & (if (taken==1) then PC.write endif)
```

Multi-cycle instructions. Some complicated arithmetic operations, such as multiply, divide and floating point operations, can require complex hardware with significantly longer delays than a single *ALU*. One solution is to have parallel pipelines for different multi-stage instructions. For example, division is frequently implemented using this scheme even in high performance superscalar processors. In addition, such an instruction stays many cycles on the same stage, mostly the first. In the pipeline of figure 3, we have a pipelined multiply/divide functional unit *MD*. The *ALU* unit executes simple operations. Instructions are issued in the *MD* floating point pipeline out-of-order. The listing 1.2 presents the execution model of the divide and multiply instructions. The *DIV* and *MUL* instructions have different latencies on the first stage of the pipelined *MD* unit. In fact, stages with different latencies is also a relevant pipeline property for hazards detection and the instructions scheduling, which is critical in an out-of-order issue processor. These latencies will be considered when generating the timing constraints to compute the execution time of instructions.

Listing 1.2. Floating point pipeline

```
1  stage FE, DE, ALU, MD0 , MD1 , MD2 , MD3, CM
2  extend FE , DE , CM
3          capacity = 2     // 2-superscalar stages
4  extend ALU , MD0
5          inorder = false // out-of-order stages
6  extend MUL
7   uses= FE#{1}, DE#{1}, MD0 & R[rd].write & R[rm].read & R[rs].read
       #{1},
```

```
8            MD1#{1}, MD2#{1}, MD3#{1}, CM#{1}
9   extend DIV
10   uses= FE#{1}, DE#{1}, MD0 & R[rd].write & R[rn].read & R[rm].read
          #{21},
11           MD1#{1}, MD2#{1}, MD3#{1}, CM#{1}
```

Micro-coded Instructions. Multiple register transfer instructions provide an efficient way of moving the contents of several registers to and from memory. These instructions take one cycle to issue but then use multiple memory cycles to load/store all the registers. We consider the pipeline of the figure 2. The load multiple instruction is given in lines 27-30 of listing 1.1. The list of registers to load is given by the operand *rl* coded on 16 bits. Every bit refers to a register and is set to one if the register is to load. So, if the register is loaded, then we have a clause in which the MEM unit, the memory and the register with the appropriate access mode are required. Otherwise, we have an empty clause. In order to have a wellformed final clause, with a valid pipeline path, we defined a semantic rule that states that: *in a clause sequence, if a clause is empty, then it is removed from the sequence:* $cl\ ,\ \emptyset\ ,\ cl' \Rightarrow cl\ ,\ cl'$. For example, we have the following instantiated clause for the instruction `ldmia r13, {, r11, r13, r15}`:
FE & PC.read&FBuf, DE , MEM&M.read&R[13].read&R[11].write&RoB , MEM&M.read& R[13].read&R[13].write&RoB ,MEM&M.read&R[13].read&PC.write&RoB , CM.

Branch Instruction. Some processors resolve branch target at the decode stage. The branch instruction is no longer used on next stages. So a branch instruction is dropped after the decode stage (lines 33-35 of listing 1.1). This is useful in out-of-order pipelines, since it reduces the structural hazards on functional units.

4 Formal Architecture Description

We formalize an architecture description through the architecture denoted by $\mathcal{A}$ and the instruction clauses of the basic block $\mathcal{BB}$. We generate an equation system $\mathcal{E}q(\mathcal{BB}, \mathcal{A})$ representing the analyzed $\mathcal{BB}$, with respect to $\mathcal{A}$. In $\mathcal{E}q(\mathcal{BB}, \mathcal{A})$, the execution times are not computed. In order to do that, we formulate a set of structural and temporal high-level constraints $Constraints\_\mathcal{A}$. The resolution of $\mathcal{E}q(\mathcal{BB}, \mathcal{A})$ and $Constraints\_\mathcal{A}$ provides an equation system where the instructions execution times have been computed (figure 4).

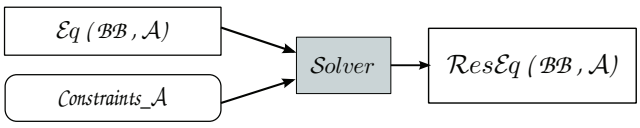

Fig. 4. Formal approach for time analysis

4.1 A Light DSL for Architecture Constraints

We introduce a light DSL (Domain Specific Language) for expressing the architecture and basic block properties. We derive from every instruction of the basic block a set of tasks that are divided into levels:

– *ISA level.* The task represents the lifetime of the instruction on the pipeline. It starts when the instruction enters the pipeline and finishes when it leaves. The task is given by the instruction index in the basic block.
– *Step level.* The task models the execution of an instruction on a stage or a functional unit, what we call a *step*. Hereinafter, we use processing units to refer to stages or functional units.
– *Resource level.* Basic tasks or leaves represent the resource allocation within an instruction step. This includes stages, functional units, buffers, registers and memory allocation.

The lifetime of every task is modeled using an interval. Table 1 summarizes the architecture DSL. We also consider the predefined functions $scal(st)$ and $nb(r)$ returning respectively a stage scalarity and a resource occurrences number. We consider the instruction sequence of figure 5 executed on the processor of figure 2 as a use case. From the Sim-nML description, the non terminal $\langle Stage \rangle$, $\langle Register \rangle$ and $\langle Memory \rangle$ are instantiated as in (1).

Table 1. Architecture DSL

Architecture domain	
$\langle Stage \rangle$, $\langle Register \rangle$, $\langle Memory \rangle$, $\langle Buffer \rangle$, $\langle Resource \rangle ::= \langle Register \mid Buffer \mid Memory \rangle$	
Basic block domain (of length n)	
$\langle instruction \rangle ::= nat$, $\langle step \rangle ::= nat$, $\langle interval \rangle ::= string$	
ISA level tasks:	**Step level tasks:**
$\langle \text{ISA} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle}$	$\langle \text{Step} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$
Resource level tasks (leaves):	
$_{\langle \text{occurrence}\mid ? \rangle}\langle Stage \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$ | $^{[r\mid w]}_{\langle \text{occurrence}\mid ? \rangle}\langle Resource \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$	

$$\begin{array}{l} \langle Stage \rangle ::= \{FE, DE, ALU, MEM, CM\} \ , \ \langle Register \rangle ::= \{R, PC\}, \\ \langle Memory \rangle ::= \{M\}, \langle Buffer \rangle ::= \{FBuf, RoB\} \end{array} \quad (1)$$

We assume the following semantic sets that we automatically generate from the architecture and the basic block:

– $\mathcal{I}$ denotes the set of tasks of the ISA level.
– $\mathcal{S}$ denotes the set of steps of all the instructions in the basic block,
– $\mathcal{L}$ denotes the set of synthesized leaves,
– $\mathcal{U}$ denotes the subset of leaves concerning stages or functional units,
– $\mathcal{B}, \mathcal{R}$ and $\mathcal{M}$ denote respectively the subset of leaves concerning buffers, registers and memories.

We consider the following dedicated quantifiers where $\forall_{\mathcal{I}}, \forall_{\mathcal{S}}, \forall_{\mathcal{L}}, \forall_{\mathcal{U}}, \forall_{\mathcal{B}}$ quantifies respectively over the basic block instructions, the steps, the leaves, the processing units and the buffers. We also use the predefined functions $Last_{\mathcal{S}}(i)$, $Buffer(i, s)$ and $Unit(i, s)$ that return respectively the last step task of i, the set of buffers and processing units of i at the step s.

$$\mathcal{I} = \{o_0^{t\_0}\ ,\ o_1^{t\_1}\ ,\ o_2^{t\_2}\}\ \ ,\ \ \mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2\ \ ,\ \ \mathcal{L}\ =\ \mathcal{L}_0 \cup \mathcal{L}_1 \cup \mathcal{L}_2$$

In the following, we detail the sets $\mathcal{S}_2, \mathcal{U}_2, \mathcal{R}_2, \mathcal{B}_2$ and $\mathcal{M}_2$ which are respectively the set of steps, stages, registers, buffers and memories of o_2. The instruction o_2 is a load multiple and loads **3 registers** from memory. So, we can observe that, based on the execution model in listing 1.1, **6** steps are generated, including **3** relative to the execution on the MEM unit. The instruction decomposition is illustrated in figure 6.

$$\mathcal{S}_2 = \{s_{2,0}^{t\_2\_0}\ ,\ s_{2,1}^{t\_2\_1}\ ,\ s_{2,2}^{t\_2\_2}\ ,\ s_{2,3}^{t\_2\_3}\ ,\ s_{2,4}^{t\_2\_4}\ ,\ s_{2,5}^{t\_2\_5}\}$$
$$\mathcal{L}_2 = \mathcal{U}_{i2} \cup \mathcal{R}_{i2} \cup \mathcal{B}_{i2} \cup \mathcal{M}_{i2}$$
$$\mathcal{U}_2 = \{{}_0FE_{2,0}^{tu\_2\_0}\ ,\ {}_0DE_{2,1}^{tu\_2\_1}\ ,\ {}_0MEM_{2,2}^{tu\_2\_2}\ ,\ {}_0MEM_{2,3}^{tu\_2\_3}\ ,\ {}_0MEM_{2,4}^{tu\_2\_4}\ ,\ {}_0CM_{2,5}^{tu\_2\_5}\}$$
$$\mathcal{R}_2 = \{{}_0^rPC_{2,0}^{tr\_2\_0}\ ,\ {}_{13}^rR_{2,2}^{tr\_2\_2}\ ,\ {}_{11}^wR_{2,2}^{tr\_2\_2}\ ,\ {}_{13}^rR_{2,3}^{tr\_2\_3}\ ,\ {}_{13}^wR_{2,3}^{tr\_2\_3}\ ,\ {}_{13}^rR_{2,4}^{tr\_2\_4}\ ,\ {}^wPC_{2,4}^{tr\_2\_4}\}$$
$$\mathcal{B}_2 = \{{}_?FBuf_{2,0}^{tb\_2\_0}\ ,\ {}_?RoB_{2,2}^{tb\_2\_2}\ ,\ {}_?RoB_{2,3}^{tb\_2\_3}\ ,\ {}_?RoB_{2,4}^{tb\_2\_4}\}$$
$$\mathcal{M}_2 = \{{}_0^rM_{2,2}^{tm\_2\_2}\ ,\ {}_0^rM_{2,3}^{tm\_2\_3}\ ,\ {}_0^rM_{2,4}^{tm\_2\_4}\ \}$$

	Assembly
o_0	`b 8410`
o_1	`sub sp, fp, #12`
o_2	`ldmia sp, {, fp, sp, pc}`

Fig. 5. Basic block

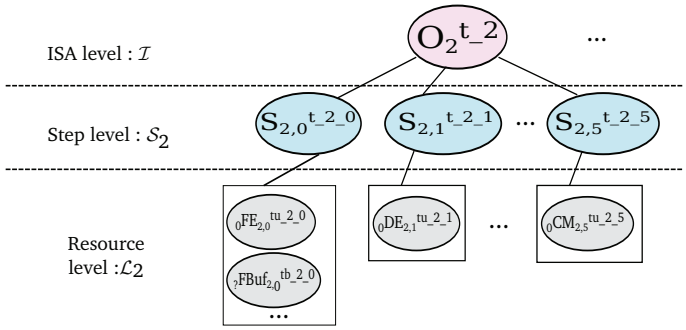

Fig. 6. Decomposition of instruction o_2

4.2 Architecture Properties

In this section we give a set of structural and dynamic properties that describes the architecture and instruction behavior. We use Allen intervals to express temporal relations between intervals. The properties are parametrized by the architecture and the basic block equations presented in the previous section.

Instruction Continuity. An instruction starts when its first step ($s = 0$) starts and terminates when its last step terminates. Considering two successive steps of an instruction, a current step finishes when the next step starts (2).

$$\begin{aligned} &\forall_{\mathcal{I}}\ o_i^t.\ \forall_{\mathcal{S}}\ s_{i,0}^{t'}.\ t\ \textbf{Starts}\ t' \\ &\forall_{\mathcal{S}}\ s_{i,s}^t.\ \forall_{\mathcal{S}}\ s'\ {}_{i,s+1}^{t'}.\ t\ \textbf{Meets}\ t' \\ &\forall_{\mathcal{I}}\ o_i^t.\ \forall_{\mathcal{S}}\ s_{i,Last_{\mathcal{S}}(i)}^{t'}.\ t\ \textbf{Finishes}\ t' \end{aligned} \tag{2}$$

Instruction Support. We assume that, within a step, an instruction requires one and only one stage and at the most one buffer resource (3a). This is a structural property used for architecture correctness validation. Two cases arise. First, if a buffer is required within a step, then this buffer is unique and is the

support of the instruction during that step. Indeed, the instruction is contained in the buffer slot throughout the step. The buffer is allocated since the instruction starts execution within the step, and remains so until the resources on the next step become available. The buffer is released when the instruction starts the next step (3b). Second, if no buffer is used, thus, the stage is the instruction support. It is blocked until the instruction starts the next step, i.e. next step resources are available (3c).

$$\forall i.\ \forall s.\ \mathbf{card}\ (Buffer(i,s)) \ \leq\ 1 \wedge \mathbf{card}\ (Unit(i,s)) \ =\ 1 \tag{3a}$$

$$\forall_{\mathcal{S}}\ s^t_{i,s}.\ \mathbf{card}\ (Buffer(i,s)) \ =\ 1 \ \Rightarrow\ \exists!_{?} b^{t'}_{i,s} \in Buffer(i,s).\ t = t' \tag{3b}$$

$$\forall_{\mathcal{S}}\ s^t_{i,s}.\ \mathbf{card}\ (Buffer(i,s)) \ =\ 0 \ \Rightarrow \exists!_{o} st^{t'}_{i,s} \in Unit(i,s).\ t = t' \tag{3c}$$

Resources Allocation Policy. An instruction executes on a given stage, once it gets all its required resources, including the stage. Thus, all required resources are allocated at the beginning of the step (4a). After the execution latency elapses, all or some of the owned resources are released. Actually, we assume that an instruction keeps the resources that are going to be asked on further steps. This allocation policy is defined to avoid deadlocks. Such a situation occurs when a micro-coded instruction, as the multiple load presented in section 3, uses the same register on several successive steps. Such instruction must not be preempted during its execution on the processing unit. When an instruction uses a resource through two successive steps, we force the temporal continuity on the allocation intervals, as presented in the constraint (4b). Some resources, like stages and buffers, can be required in *non-deterministic* way: the instruction requests for any of the available occurrences, or in a *deterministic* way: the instruction requests for a specific occurrence. For a resource r, we have to insure that no more then $nb(r)$ occurrences are allocated at the same time. This contention problem occurs in case of non-deterministic resources with dynamic scheduling. Currently, the property (4c) concerns stages and buffers.

$$\forall_{\mathcal{S}}\ s^t_{i,s}.\ \forall_{\mathcal{L}}\ {}^{a}_{o}res^{t'}_{i,s}.\ t\ \mathbf{Starts}\ t' \wedge t'\ \mathbf{During}\ t \tag{4a}$$

$$\forall_{\mathcal{L}}\ {}^{a}_{o}r^t_{i,s}.\ \forall_{\mathcal{L}}\ {}^{a}_{o}r^{t'}_{i,s+1}.\ t\ \mathbf{Meets}\ t' \tag{4b}$$

$$\forall_{\mathcal{U}\cup\mathcal{B}}\ {}_{o}x^t_{i,s}.\ \mathbf{card}\ (\{{}_{o}x^{t'}_{i',s'} \in \mathcal{U}\cup\mathcal{B} \mid t\ \mathbf{Overlaps}\ t'\}) \ \leq\ nb(x) \tag{4c}$$

Stage Specific Constraints. These constraints depend on the stages execution features. For a simple-scalar stage with in-order execution, only one instruction is issued to the stage per cycle and is issued before its successor in the program (5a).

$$\forall_{\mathcal{U}}\ {}_{o}st^t_{i,s}.\ \forall_{\mathcal{U}}\ {}_{o}st^{t'}_{i',s'}.\ \ i < i' \Rightarrow t\ \mathbf{Before}\ t' \tag{5a}$$

$$\forall_{\mathcal{U}}\ {}_{o}st^t_{i,s}.\ \mathbf{card}\ (\{{}_{o}st^{t'}_{i',s'} \in \mathcal{U} \mid t\ \mathbf{Overlaps}\ t'\}) \ \leq\ scal(st) \tag{5b}$$

$$\forall_{\mathcal{U}}\ {}_{o}st^t_{i,s}.\ \forall_{\mathcal{U}}\ {}_{o}st^{t'}_{i',s'}.\ i' < i + scal(st) \Rightarrow t\ \mathbf{StartsBeforeBegin}\ t' \tag{5c}$$

$$\forall_{\mathcal{U}}\ {}_{o}st^t_{i,s}.\ \forall_{\mathcal{U}}\ {}_{o}st^{t'}_{i+scal(st),s'}.\ t\ \mathbf{Before}\ t' \tag{5d}$$

In case we have a super-scalar stage st, at most $scal(st)$ successive instructions can be issued in parallel (5b). However the overall program order must be maintained. Thus, an instruction of index i can be executed in parallel with an instruction of index j such that $j < i + scalst$. We introduce a new time relation $StartsBeforeBegin$ that define a priority relation between two intervals (5c). The scalarity limit of the stage is expressed by a forced precedence between instruction i and instruction $i + scal(st)$ (5d). For out-of-order stages, no precedence is defined except those implied by the data dependencies.

Data Dependencies. Memories and registers can be owned in a read or a write mode. For example, when a register is accessed by two successive instructions such that the first request is a read access and the second is a write access, then, the read access must occur before the write access. Read After Write (RAW) hazards are explicited when instructions access registers and memories. Write After Write (WAW) hazards must be explicited for memory accesses.

$$\begin{aligned} &\forall_{\mathcal{R}}\ {}^{r}_{o}reg^{el}_{i,s}.\ \forall_{\mathcal{R}}\ {}^{w}_{o}reg^{el'}_{i',s'}.\ i \leq i' \Rightarrow el\ \textbf{Before}\ el' \\ &\forall_{\mathcal{R}}\ {}^{a}_{o}m^{el}_{i,s}.\ \forall_{\mathcal{R}}\ {}^{w}_{o}m^{el'}_{i',s'}.\ i \leq i' \Rightarrow el\ \textbf{Before}\ el' \end{aligned} \tag{6}$$

To elaborate the constraints $Constraints\_\mathcal{A}$, we instantiate all (or some of) the constraints presented in equations from 2 to 6, with respect to the architecture $\mathcal{A}$ and the basic block sets. We detail here the constraints of the instruction o_0 which is a branch (the corresponding execution model is in listing 1.1):

(2) $\rightarrow$ t_2 **Starts** t_2_0 $\wedge\, t\_2\_0$ **Meets** $t\_2\_1 \wedge$ ... t_2_1 **Finishes** t_2
(3) $\rightarrow$ $tb\_2\_0 = t\_2\_0$ $\wedge$ $tu\_2\_1 = t\_2\_1$ $\wedge$ $tb\_2\_2 = t\_2\_2$ $\wedge$ $tb\_2\_3 = t\_2\_3$ $\wedge$ ...
(4) $\rightarrow$ tu_2_2 **Meets** $tu\_2\_3 \wedge tu\_2\_3$ **Meets** tu_2_4 $\wedge$ tb_2_2 **Meets** $tb\_2\_3 \wedge \ldots$
(6) $\rightarrow$ t_0_1 **Before** $t\_2\_0 \wedge t\_1\_2$ **Before** t_2_2 $\wedge$ ...

These high-level constraints are used to generate a constraint description to be processed by a given solver. In [11], we show how to solve these constraints with the CSP/CHOCO [1]. The resolution of the constraints for the basic block of the figure 5 executed on the processor of figure 2 gives the following values, among others :$t\_0 = [0, 2]$; $t\_1 = [2, 6]$ *and* $t\_2 = [2, 9]$. In [12], we also show how to animate such properties through the timed automata provided by the UPPAAL tool [7].

5 Related Work

There have been a lot of research efforts in architecture description languages that aim to formalize the processor specification provided in user manuals. Most of these efforts was based on ADLs like ArchC [15], LISA [13], HARMLESS [5] and Sim-nML [11], that was used to generate retargetable tools. Since these ADLs are actually mature regarding the ISA level description, modern ISAs are currently supported and accurate simulators can be generated. Nevertheless, these approaches assume simple pipelines like DLX, since the micro-architecture

is not handled or limited to its structure. Our general approach based on Sim-nML supports the retargetable tool generation, in addition to complex instruction execution models. Retargetable WCET analysis tools based on processor description are presented, among others, in [14] [19] [6] [9]. Timed automata was observed in [6] [9] as a formal processor description. This model is difficult to elaborate and is hand-made, which limits experimentations to simple five stage pipelines with in-order execution. Our approach offers a high-level processor description that can be used to generate timed automata. The work of [19] is close to ours. The authors use the EXPRESSION language to describe the ISA and the micro-architecture and generates execution graphs for the analysis. However, the ADL description only includes the pipeline structure. The hardware components behavior, such as out-of-order execution and superscalarity, are specified in C++ external libraries. The use of many formalisms to processor description makes it not suitable for validation. Other methods verify program behavior and the memory design [3] on PowerPC and ARM architecture using a formalization. This approach uses the L3 language [2] for the ISA description and the generation of a HOL specification used for verification. The formal description is limited to the ISA level and cannot be used for time analysis. The table 2 summarizes the architecture languages capabilities.

Table 2. Architecture languages summary table

	ArchC	Chronos	LISA	Harmless	Expression	L3	Otawa
Code optimization							
Simulation	•	•	•	•			•
Retargetable tool generation	•		•	•			•
Design space exploration	•		•		•		
Time analysis		•			•		•
Formalization	•			•	•	•	•
Verification/Validation					•	•	•
Out-of-order pipeline		•				•	•
Superscalar pipeline	•	•	•			•	•
Complex instruction set	•		•			•	•

6 Conclusion

In this paper, we have showed how advanced architecture features can be described through an extension of the Sim-nML architecture description language. We have presented the syntax for describing advanced pipelines structure and instructions with complex execution models. We have also presented a specification language that aims to formalize the architecture features. We have elaborated a set of generic properties that can be used for the generation of architecture constraints for time analysis and the validation. This formalization can be a basis for formal verification, required to obtain certified WCET computation methods. The proposed approach for time analysis is modular since it is based

on the hardware description and the basic block, each one independent from the other. Modularity also comes from the fact that the Sim-nML description of the ISA and the micro-architecture are separated in implementation terms, so that the same ISA description can be reused for different pipelines. High-level constraint description present a significant advantage since the user is free to choose the resolution method for WCET computation. We have drawn a comparison between currently used description languages and formalism. In this paper we have used a processor model that possess real word processor features to test our approach. We are currently validating our formalization on real-life processors that are rolling out on the market, among which the Cortex-A8.

References

1. CHOCO: An Open Source Java Constraint Programming Library, `http://choco.mines-nantes.fr`
2. L3: An ISA Specification Languag, `http://www.cl.cam.ac.uk/~acjf3/l3/`
3. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and arm multiprocessor machine code. In: Workshop on Declarative Aspects of Multicore Programming (DAMP), pp. 13–24 (2008)
4. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: Min, S.L., Pettit, R., Puschner, P., Ungerer, T. (eds.) SEUS 2010. LNCS, vol. 6399, pp. 35–46. Springer, Heidelberg (2010)
5. Béchennec, J.L., Briday, M., Alibert, V.: Extending harmless architecture description language for embedded real-time systems validation. In: International Symposium on Industrial Embedded Systems, pp. 223–231 (2011)
6. Béchennec, J.L., Cassez, F.: Computation of wcet using program slicing and real-time model-checking. CoRR (2011)
7. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126 (2006)
8. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints 12, 21–62 (2007)
9. Dalsgaard, A., Olesen, M., Toft, M., Hansen, R., Larsen, K.: METAMOC: Modular execution time analysis using model checking. In: Workshop on Worst-Case Execution Time Analysis (WCET), vol. 15, pp. 113–123 (2010)
10. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using nML. In: European Design and Test Conference (EDTC), pp. 503–507 (1995)
11. Herbegue, H., Cassé, H., Filali, M., Rochange, C.: Hardware architecture specification and constraint based wcet computation. In: International Symposium on Industrial Embedded Systems (SIES), pp. 259–268 (2013)
12. Herbegue, H., Filali, M., Cassé, H.: A constraint-based wcet computation framework. In: Junior Researcher Workshop on Real-Time Computing (JRWRTC), pp. 33–36 (2013)
13. Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., Someren, H.V.: A methodology and tool suite for c compiler generation from adl processor models. In: Design, Automation and Test in Europe Conference and Exhibition, vol. 2, pp. 276–1281 (2004)
14. Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for WCET analysis. Real-Time Systems 34, 195–227 (2006)

15. Miele, A., Pilato, C., Sciuto, D.: An automated framework for the simulation of mapping solutions on heterogeneous mpsocs. In: International Symposium on System on Chip (SoC), pp. 1–6 (2012)
16. Mishra, P., Dutt, N.: Modeling and validation of pipeline specifications. ACM Trans. Embed. Comput. Syst., 114–139 (2004)
17. Rochange, C., Sainrat, P.: A context-parameterized model for static analysis of execution times. High-Performance Embedded Architectures and Compilers II (2009)
18. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-Case Execution-Time problem–overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS), 36:1–36:53 (2008)
19. Xianfeng, L., Abhik, R., Tulika, M., Prabhat, M., Xu, C.: A retargetable software timing analyzer using architecture description language (2007)