Electronic Communications of the EASST
Volume 44 (2011)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2011)

An Architecture Description Language for Embedded Hardware Platforms

Guillaume Savaton, Jean-Luc Béchennec, Mikaël Briday and Rola Kassem

16 pages

Guest Editors: Jordi Cabot, Robert Clariso, Martin Gogolla, Burkhart Wolff
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Architecture Description Language for Embedded Hardware Platforms

Guillaume Savaton[1], Jean-Luc Béchennec[2], Mikaël Briday[3] and Rola Kassem[4]

[1]guillaume.savaton@eseo.fr, http://trame.eseo.fr
TRAME, Transformation de Modèles pour l'Embarqué
ESEO, Angers, France

[2]jean-luc.bechennec@irccyn.ec-nantes.fr, [3]mikael.briday@irccyn.ec-nantes.fr,
[4]r.kassem@bau.edu.lb, http://www.irccyn.ec-nantes.fr/
Systèmes Temps Réel
IRCCyN, UMR CNRS 6597, Nantes, France

Abstract: Embedded software development relies on various tools – compilers, simulators, execution time estimators – that encapsulate a more-or-less detailed knowledge of the target hardware platform. These tools can be costly to develop and maintain: significant benefits could be expected if they were automatically generated from models expressed in a dedicated modeling language.

In contrast with Hardware Description Languages (HDLs), that focus on the internal structure and behavior of an electronic board of chip, Hardware *Architecture* Description Languages consider hardware as a platform for software execution. Such a platform will be described in terms of low-level programming interface (processor instruction set), resources (processing elements, memory and peripheral devices) and elementary services (arithmetic and logic operations, bus transactions).

This paper gives an overview of HARMLESS (*Hardware ARchitecture Modeling Language for Embedded Software Simulation*), a new domain-specific language for modeling embedded hardware platforms. HARMLESS and its associated tools follow the Model-Driven Engineering philosophy: metamodeling and model transformations have been successfully applied to the automatic generation of processor simulators.

Keywords: Model-Driven Engineering, Architecture Description Language, Computer Architecture, Simulation

1 Introduction

Hardware designers have a long tradition of using models to represent the structure and behavior of electronic circuits. These models are based on a wide range of paradigms and abstraction levels: for instance, digital logic is usually described either structurally – as an interconnection of gates, flip-flops, or more complex components – or behaviorally through equations, truth tables, state machines. Since the 1980s, most of the digital hardware design is done using hardware description languages (HDLs) such as VHDL[IEE00] and Verilog. Inspired by programming

languages, HDLs allow to model hardware using similar concepts as those used in software: algorithmic representation of sequential behaviors, process-based concurrency, etc.

However, today, most digital hardware architectures include one or more programmable processors: an electronic chip or board is no longer a self-contained system, but is now considered as a *platform* for running software. While HDLs are still useful for digital hardware synthesis, they suffer from heavy limitations when used for system analysis and simulation, as soon as software is involved. A partial solution to this problem is provided by System-Level Design Languages (SLDLs) such as SystemC, that allows to mix hardware as well as software components in a single model, providing abstractions for modeling concurrency and communications.

While an HDL-based model focuses on capturing the internal microarchitecture of a processor – merely considering software as a set of binary data stored in a memory device –, an SLDL-based model will capture the behavior of software components, regardless of the processors that will execute them. As a consequence, the use of SLDLs still shows limitations in the context of software execution analysis. In fact, early embedded software verification and test would benefit from tools that would provide information such as: execution time, stack usage, exceptions such as privilege violations, etc. When it comes to low-level software execution analysis, SLDL models will not provide the needed information, while HDL models will provide excessively detailed information with no straightforward relationship with the concerned software entities.

There is a need for a third kind of languages: hardware architecture description languages, or rather hardware *platform* description languages, that would explicitly capture the software execution mechanisms, in terms of low-level programming interface (processor instruction set), resources (processing elements, memory and peripheral devices) and elementary services (arithmetic and logic operations, bus transactions). As a possible solution, this paper presents a new domain-specific language called HARMLESS (*Hardware ARchitecture Modeling Language for Embedded Software Simulation*) that addresses the problem of modeling embedded computers at an adequate abstraction level for software execution analysis. As the name implies, HARMLESS models are primarily considered as a source for generating simulators, but other uses can emerge as the project evolves.

This paper will focus on the most stable part of HARMLESS, that allows to model processor cores. Modeling microcontrollers and microcontroller-based systems – including processor cores, on-chip and off-chip memory and peripheral devices –, and generating the corresponding simulators, is work in progress. The pipeline definition constructs provided by HARMLESS are presented in [KBB+08] and will not be described in detail in this paper.

2 Related Works

The idea of hardware architecture description languages and, more specifically, processor description languages is not new. Several research teams have proposed alternatives to the existing HDLs, generally with the following concerns: automatic hardware synthesis from a higher abstraction level; automatic toolkit generation (assemblers, compilers); automatic simulator generation. As far as simulation is concerned, a distinction must be made between Instruction-Set Simulators (ISS), that simulate only the functional behavior of the processor, and Cycle-Accurate Simulators (CAS) that can also measure execution time. In order to generate a cycle-accurate

simulator, the needed timing information will either be explicitly captured in the source model, or inferred from a description of the internal concurrency and synchronization mechanisms used by the processor. There are basically three kinds of hardware architecture description langugages:

- Structure-oriented languages such as MIMOLA[BBH+94] will model a hardware architecture as a set of interconnected components. This kind of languages is particularly suited to automatic hardware synthesis, where the main goal of the modeler is to design and implement a new computer. They often tend to resemble glorified HDLs that will suffer from similar limitations.

- Instruction-set-oriented languages such as nML[Fre93] and ISDL[HHD97] will focus on modeling the hardware/software interface (instruction set, memory and peripheral address mapping), usually based on behavioral abstractions. Models produced with these languages can have various uses: automatic generation of software development tools [Bha01], and, in a limited way, automatic hardware synthesis[Bas01]. But the primary goal of such models is to help software analysis.

- Mixed languages such as LISA[PHZM99] and EXPRESSION[HG99] will provide both kinds of constructs. In fact, LISA is usually presented as a language for custom processor design, so that multiple concerns must be addressed: hardware synthesis as well as compiler and simulator generation.

As far as software execution analysis is concerned, structure-oriented languages tend to require too much information about the internal microarchitecture of the processor. While such structural details can be useful to infer how much time will be spent along the datapath for each kind of instruction, we believe that the same information can be captured in a more concise way.

The aforementioned instruction-set-oriented languages are well-suited to ISS generation. The additional information required to infer timing and other properties must often be expressed separately, e.g. in the form of hand-written C libraries. Therefore these language tend to be tedious to use for modeling processors with complex pipelines – e.g. multiple-issue pipelines with out-of-order execution.

3 Overview of HARMLESS

3.1 Information Needed for Software Execution

HARMLESS is based on the trivial observation that an instruction set is a language. As such, modeling a processor can be closer from designing the grammar of a language than describing interconnected components. To further illustrate this claim, we can observe that the databook of a processor generally provides four kinds of information about the instruction set. For each instruction or instruction class, we can get: its syntax in assembly language; its binary format; its execution semantics; its timing information.

At this level of abstraction, it is important to notice that the assembly language syntax and the binary format of instructions are just two different representations, human-readable and machine-readable, of the very same information. From this observation, we can infer the existence of

a common "abstract syntax" for processor instructions, from which the assembly language and machine language are just two possible concrete forms. The abstract syntax will capture the static properties that define an instruction (which operation to perform, which registers or addressing modes to use), regardless of any concrete (textual) syntax or binary encoding.

To expose the execution semantics of instructions, processor databooks generally provide minimal information about the processor's hardware. This is generally known as the "programmer's view", or "programmer's model" of the processor. This programmer's model usually provides the list of registers and the memory organization as seen by the processor (address mapping, endianness, access sizes, alignment constraints, etc), without detailing the busses and control signals. Therefore the execution semantics of an instruction can be expressed in terms of elementary arithmetic and logic operations that modify the state of registers and memory cells.

Finally, depending on the processor family, the timing information is provided either as a number of clock cycles needed to perform each basic operation, or as a description of the pipeline organization.

3.2 A Grammar-Based Modeling Language

Like *nML* [Fre93], HARMLESS is based on the assumption that an instruction set is better described as a grammar than a plain list of instructions. A grammar will provide a hierarchical decomposition of the instruction set into "production rules" (also called "partial instructions"), allowing to factorize properties and behaviors common to several instructions.

Unlike *nML* and *Sim-nML*, HARMLESS enforces a clean separation between different views of a processor model: abstract syntax and behavior of instructions, assembly language syntax, binary format, pipeline description. The main reasons for this separation are:

- Separation of concerns: the designer can focus on one view at a time while keeping his model readable.
- Processors with multiple instruction sets: some processor families (ARM, MIPS) support a default 32-bit instruction set and a "compact" 16-bit instruction set ("MIPS16", "ARM Thumb").
- Processors with complicated instruction sets: the mapping between abstract syntax and binary format is not always straightforward (e.g. the same addressing mode can be encoded differently depending on the instruction). This happens frequently in commercial processor families that have evolved through many versions of their architectures and instruction sets while attempting to preserve backward compatibility.

3.3 Modeling and Code Generation Strategy

Like many domain-specific modeling languages, HARMLESS combines constructs specifically related to the domain of computer modeling (e.g. instruction encoding, pipeline definitions) as well as constructs from traditional programming languages (e.g. imperative constructs for modeling instruction behavior). Depending on their technical background, beginners in HARMLESS can feel disoriented; confusions may arise, that will lead to incorrect or inefficient models. A successful use of HARMLESS will rely on the following steps (see Figure 1):

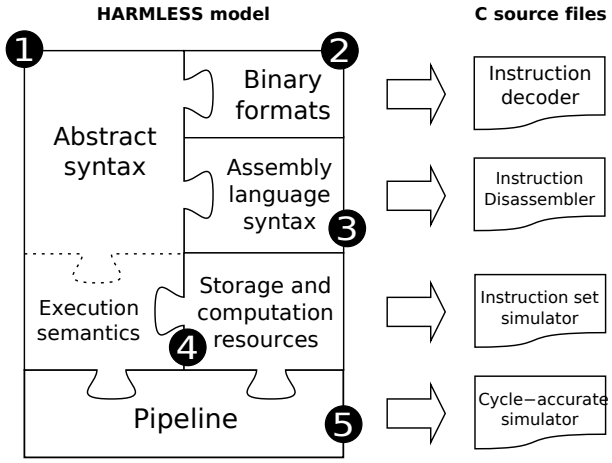

Figure 1: Processor views and simulator generation using HARMLESS

1. Preliminary analysis of the instruction set: an "abstract syntax" model is written in the form of a grammar that captures the hierarchy of partial instructions and their static properties.

2. Description of the binary format(s) attached to each partial instruction.

3. Description of the assembly language syntax attached to each partial instruction.

4. Definition of the programmer's model (registers, memory, behavior of the functional units) and description of the behavior attached to each partial instruction.

5. Modeling of the pipeline.

Since a computer model can be complex, designers will wish to test their model incrementally. In the aforementioned sequence of steps, each intermediate model can contribute to a part of the generated simulator that can be tested separately. As shown in Figure 1, after step 2, an instruction decoder can be generated and tested; test vectors can also be automatically generated. Step 3 allows to generate a full disassembler, that can be tested against the same test vectors. After step 4, the model contains all information required to generate an instruction set simulator (ISS). After step 5, a full processor simulator is obtained, with clock-cycle-accurate timing information.

4 Processor Modeling with HARMLESS

This section exposes an extract of the HARMLESS metamodel, with examples written using the concrete syntax of HARMLESS. The organization of this section follows the logical order proposed in Subsection 3.3.

4.1 Analysis of the Instruction Set

The instruction set of the processor is modeled in the form of an abstract grammar composed of a hierarchy of rules called *partial instructions*. Figure 2 shows a simplified view of the classes involved in describing the abstract syntax of an instruction set. Partial instructions can fall into two categories:

- An *alternative partial instruction* is an OR-type rule that describes a selection between several choices.
- An *aggregate partial instruction* is an AND-type rule that describes a composition of several other partial instruction instances.

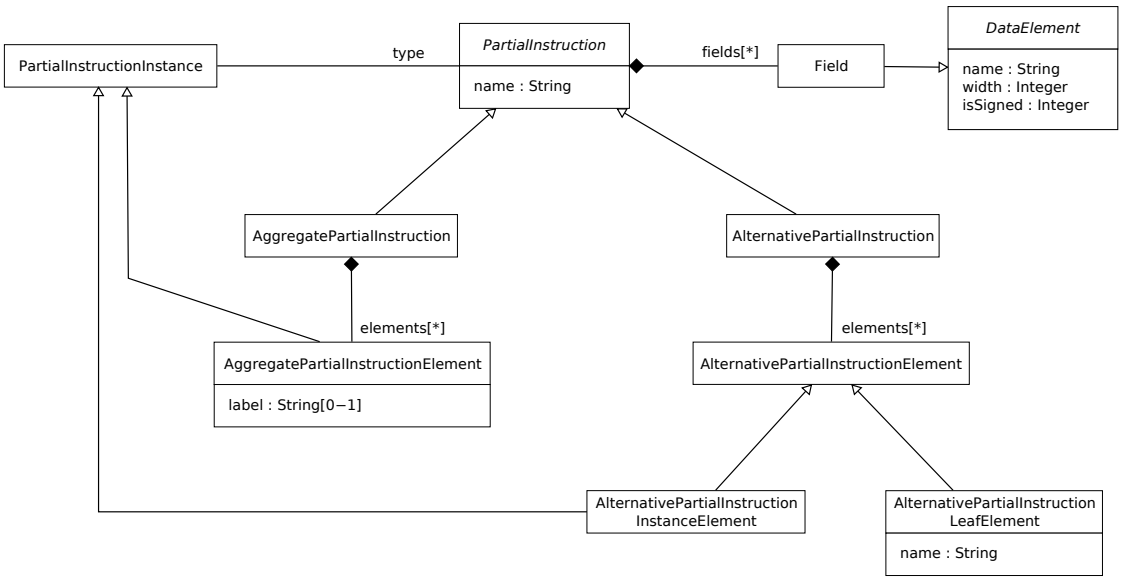

Figure 2: Extract of the HARMLESS metamodel: instruction abstract syntax modeling

Both kinds of partial instructions can have *fields* that will hold the static data needed by the instruction. A field is always of integer type, with a given width (number of bits) and signedness.

Figure 3 shows an extract of a simple, hypothetical, processor description. The first *alternative* states that an *instruction* can be either a *data processing instruction*, a *memory access instruction* or a *branch instruction*. The following aggregate defines a *data processing instruction* as a composition of a *source operand* and a *data processing operation*. This aggregate has one field that provides the index of the register that will receive the result of the operation. The last shown alternative defines the set of available *data processing operations*: each operation is defined by a terminal (leaf) symbol, syntactically identified by the "#" character.

```
alternative Instruction {
  DataProcessingInstruction
  MemoryAccessInstruction
  BranchInstruction
}

aggregate DataProcessingInstruction {
  field u'3 targetReg -- Target register number
  SourceOperand
  DataProcessingOperation
}

alternative SourceOperand {
  Register
  Immediate
}

aggregate Register {
  field u'3 reg -- Register number
}

aggregate Immediate {
  field s'16 imm -- Immediate value
}

alternative DataProcessingOperation {
  #MOV
  #NOT
  #ADD
  #SUB
  #AND
  #OR
}
...
```

Figure 3: HARMLESS example: abstract instruction set definition

4.2 Binary Formats

The binary encoding of instructions is defined as a set of *formats* attached to the *partial instructions*. First of all, a *format* defines how each *field* of a partial instruction is obtained from a set of selected bits of the binary instruction word. For most *aggregate partial instructions*, no additional information needs to be provided, unless we are dealing with a variable-length instruction set and we want to enforce a precise ordering of the contained *elements* as they are expected to be fetched and decoded. For an *alternative partial instruction*, the associated formats will expose the mapping between each possible choice and the corresponding values of a selection of bits in the instruction words. Figure 4 illustrates how these concepts are captured in the HARMLESS metamodel.

The listing shown on Figure 5 provides the set of formats associated with *data processing instructions* in our example processor. It is important to notice that it is not mandatory to provide a *format* for each *partial instruction*: contrarily, the designer will focus on selecting a minimal set of formats that unambiguously specify the encoding of all available instructions.

4.3 Assembly Language Syntax

The syntax of instructions is defined in a similar way as the binary formats (Figure 6). A *syntax* element describes a mapping between a *partial instruction* and a corresponding textual representation. For an *aggregate partial instruction*, the textual representation is provided as an *expression* that specifies how the text should be built using the *partial instruction*'s contents. For an *alternative partial instruction*, each choice is mapped onto a similar *expression*; optionally, a *prefix* and a *suffix expression* can be provided, common to all choices. In both kinds of *partial instructions*, an *expression* must be provided for each *field*.

The metamodel classes for *expressions* used in syntax definitions are not shown on Figure 6. They are better illustrated on the example given on Figure 7. Five kinds of *expressions* are shown in this example: literal character strings (`"R"`, `"MOV"`); field values, in decimal (`\d`), hexadecimal (`\x`) or binary (`\b`); insertion of the text for a given field (`reg`, `imm`); insertion of the text for a given partial instruction (`DataProcessingOperation()`, `SourceOperand()`); concatenation of two or more expressions, listed between delimiters << and >>; the + operator indicates that no space should be inserted between two consecutive expressions.

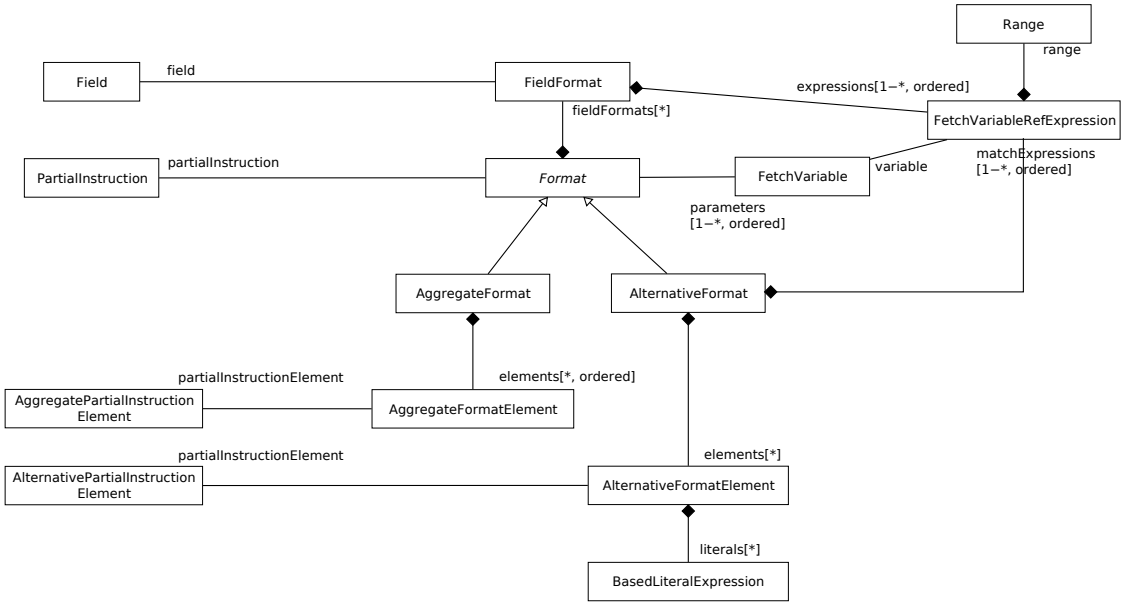

Figure 4: Extract of the HARMLESS metamodel: instruction binary format modeling

```
fetch var u'16 instr

aggregate format DataProcessingInstruction(instr) {
  targetReg=instr{10..8}
}

alternative format DataProcessingOperation(instr) match instr{14..12} {
  #MOV is \b"000"
  #NOT is \b"001"
  #ADD is \b"010"
  #SUB is \b"011"
  #AND is \b"100"
  #OR  is \b"101"
}

alternative format SourceOperand(instr) match instr{11} {
  Register  is \b"0"
  Immediate is \b"1"
}

aggregate format Register(instr) {
  reg=instr{2..0}
}

aggregate format Immediate(instr) {
  imm=instr{7..0}
}
...
```

Figure 5: HARMLESS example: binary format definition

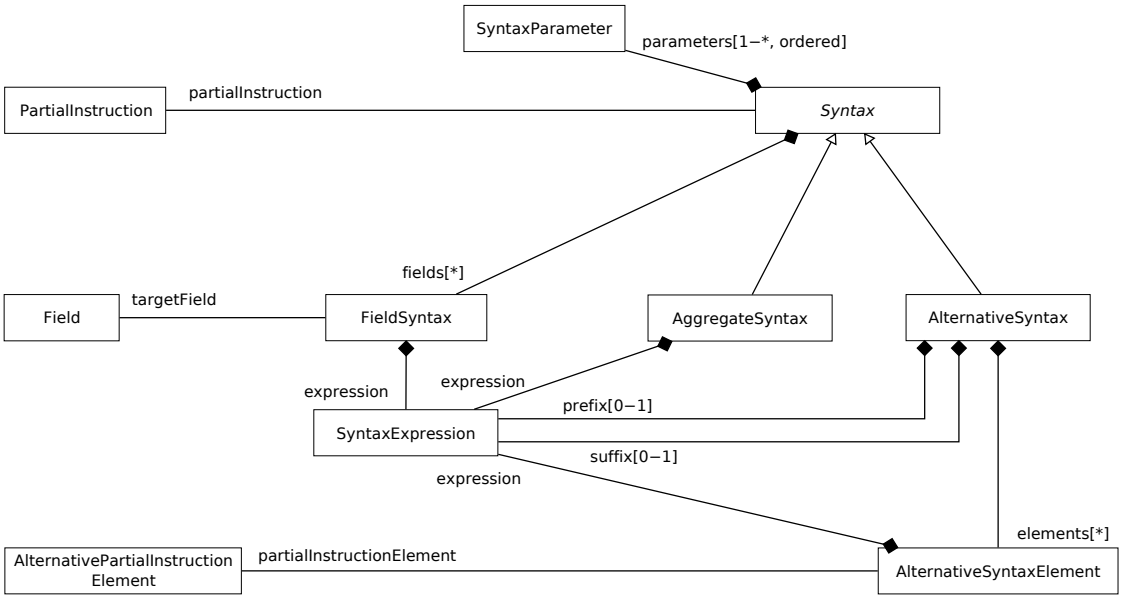

Figure 6: Extract of the HARMLESS metamodel: assembly language syntax

```
aggregate syntax DataProcessingInstruction {
  targetReg = << "R" + \d >>
  emit << DataProcessingOperation() targetReg
       + "," SourceOperand() >>
}

aggregate syntax Register {
  reg = << "R" + \d >>
  emit reg
}

aggregate syntax Immediate {
  imm = << "#" + \x >>
  emit imm
}

alternative syntax DataProcessingOperation {
  #MOV is "MOV"
  #NOT is "NOT"
  #ADD is "ADD"
  #SUB is "SUB"
  #AND is "AND"
  #OR  is "OR"
}
...
```

Figure 7: HARMLESS example: assembly language syntax definition

4.4 Execution Semantics

The execution semantics of an instruction set relies on a programmer's model of the processor, where the storage elements (registers, memory) and the computation units are defined. Storage elements are modeled as *global variables* of the model while computation units are modeled as *components* that provide *operations* (see the corresponding metamodel classes on Figure 8). The detailed behavior attached to each *operation* or *partial instruction* is defined in the form of sequences of *statements* similar to the imperative statements found in most programming languages: *variable assignments*, *operation calls*, *conditional* and *loop* constructs. In *partial instructions*, *local variables* and *parameters* represent the dynamic data that are needed or computed when their behavior is executed.

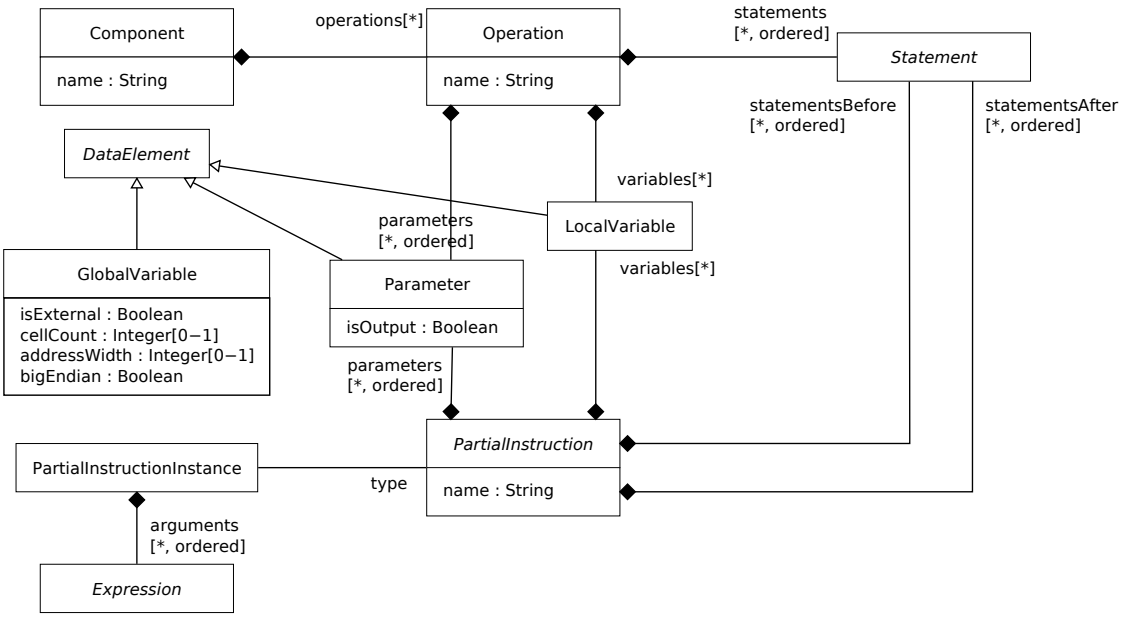

Figure 8: Extract of the HARMLESS metamodel: execution semantics of partial instructions

Figure 9 shows an example where a few global variables are defined, representing the internal register bank of the processor, its status bits and its memory interface. An extract of an Arithmetic and Logic Unit (ALU) *component* is provided, with a sample operation *add* that computes the sum of its arguments and updates the status bits accordingly.

Figure 10 is a completed version of Figure 3, with behavior information added to the *partial instructions*. In *aggregate DataProcessingInstruction*, it is noticeable that the ordering of the instanciated *partial instructions DataProcessingOperation* and *SourceOperand* does not affect the actual execution ordering, which is automatically inferred from the data dependencies.

```
var u'16 R[8] -- The register bank
var u'1 N -- "Negative" status bit
var u'1 Z -- "Zero" status bit

-- External memory, with 8-bit data and 16-bit addresses
external var u'8 Memory<16> big endian

component ALU {
  s'16 add(s'16 a, s'16 b) {
    var s'16 result
    result := a + b
    N := result < 0
    Z := result = 0
    return result
  }
  ...
}
...
```

Figure 9: HARMLESS example: storage and computation resource definition

```
aggregate DataProcessingInstruction {
  field u'3 targetReg
  var s'16 sourceVal
  DataProcessingOperation(R[targetReg], R[targetReg], sourceVal)
  SourceOperand(sourceVal)
}

alternative SourceOperand(out s'16 data) {
  Register(data)
  Immediate(data)
}

aggregate Register(out s'16 regValue) {
  field u'3 reg
  do { regValue := R[reg] }
}

aggregate Immediate(out s'16 immValue) {
  field s'16 imm
  do { immValue := imm }
}

alternative DataProcessingOperation(out s'16 result, s'16 a, s'16 b) {
  #MOV { result := ALU.mov(b) }
  #NOT { result := ALU.not(b) }
  #ADD { result := ALU.add(a, b) }
  #SUB { result := ALU.sub(a, b) }
  #AND { result := ALU.and(a, b) }
  #OR  { result := ALU.or(a, b) }
}
...
```

Figure 10: HARMLESS example: instruction behavior definition

5 Model-Driven Simulator Generation

The HARMLESS modeling environment is based on Eclipse and the AMMA platform, including the following technologies: EMF (Eclipse Modeling Framework) for model data management, TCS (Textual Concrete Syntax) for text-to-model and model-to-text transformation [JBK06] and ATL (ATLAS Transformation Language) for model-to-model transformation [JK05]. Simulator generation combines the following transformation steps (see Figure 11 and Figure 12):

1. Injection (text to model): the source processor description is parsed into an EMF model. The user is informed of possible syntax errors in the source description.

2. Semantic analysis: this step is basically an ATL transformation that checks the source HARMLESS model against a set of well-formedness rules [BJ05]. Conditions that source elements must respect are expressed in OCL, and a "problem" element is generated (i.e. a message with severity and location information) each time a condition fails. The AMMA platform provides the "Problem" metamodel and an API to feed the "Problems" view of the Eclipse workbench.

3. Preprocessing: this transformation puts the source HARMLESS model into a "canonical form" in order to prepare the following transformation.

4. Simulator model generation: the *Generic Procedural Programming Language* metamodel is a language-agnostic abstraction of programming languages such as C, Ada or Pascal. While our concern is C code generation, this metamodel allows our transformation to focus on the generic concepts required to model a simulator (data types, subprograms, statements, etc), rather than C-specific concepts and syntax.

5. Extraction (model to text): C source code is generated from the intermediate generic procedural model. The resulting files are compiled using GCC.

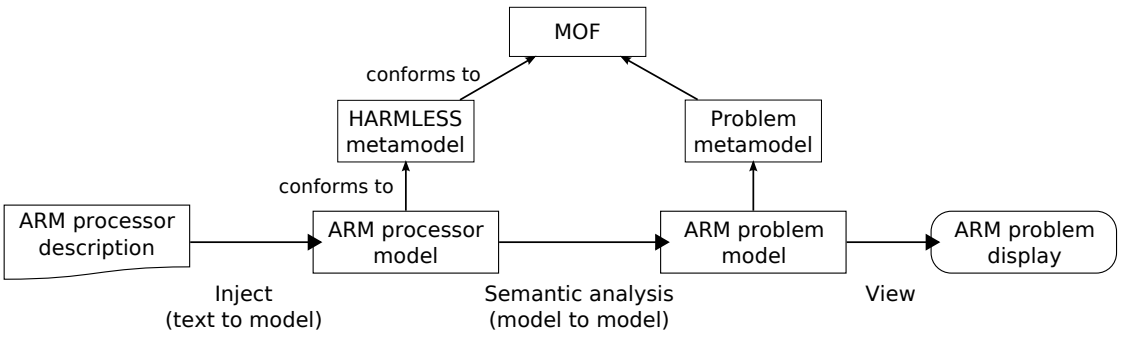

Figure 11: Semantic analysis of HARMLESS models and error reporting

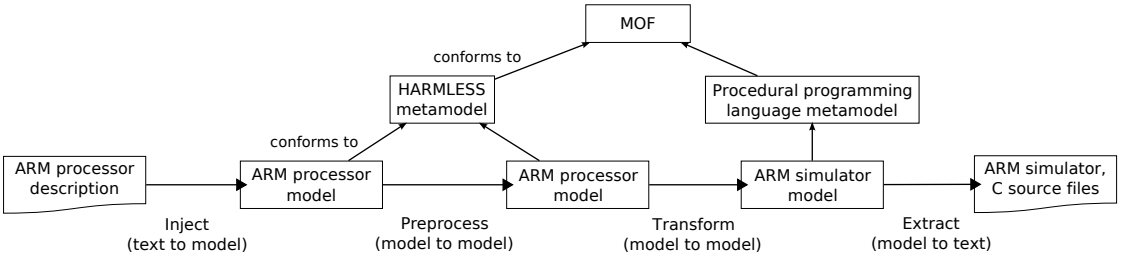

Figure 12: Simulator generation from HARMLESS models

6 Results

Table 1 shows results for a few processor models. For each processor family, the following metrics are given: the number of *partial instructions* (PI) in the source (textual) processor description; the number of *format* definitions in the source description; the number of *syntax* definitions in the source description; the number of lines of code (LoC) in the source description file; the number of lines of C code in the generated simulator. In the case of the Freescale HCS12, the source model does not contain behavior information: the generated C files do not constitute a full simulator, but only a disassembler (instruction decoding and assembly code extraction).

Table 1: Examples of HARMLESS processor descriptions

Processor family	PI	Formats	Syntaxes	Src. LoC	C LoC
ARM	25	24	22	980	8100
Renesas SH2	30	41	25	1724	11590
Freescale HCS12	54	74	59	1340	13320

The Renesas SH2 model was created collaboration with B2i Automotive Engineering[1]. The work was carried out by a student in master's degree of computer science and electronic engineering, during a six-month internship. The student had received basic education in computer architecture and had been trained in low-level hardware modeling with languages such as VHDL, Verilog and SystemC. He had no prior knowledge of the HARMLESS project, and no previous experience of the SH2 processor family. Table 2 gives the approximate time spent in each modeling phase.

During this work, a significant part of the time was spent testing and debugging the processor model, and writing documentation. Currently, the on-going work consists in validating the timing measurements given by the cycle-accurate simulator against a real SH2-based embedded computer.

[1] http://www.b2i-automotive.com

Table 2: Renesas SH2 processor modeling tasks

Task	Duration
Self-introduction to HARMLESS and the SH2 architecture	1 week
Instruction set modeling (partial instructions, formats, syntax) Disassembler generation and test	1 month
Behavior modeling Instruction-set simulator generation and test	2 months
Pipeline modeling Cycle-accurate simulator generation	2 weeks

7 Conclusion and Future Work

This paper has given an overview of the main features provided by HARMLESS for processor instruction set modeling. HARMLESS has proved to have a fast learning curve, even for users non-expert in computer architecture: this is a strong advantage for a new language based on concepts found neither in programming languages nor in hardware description languages. The results show the benefits of using HARMLESS as a front-end for simulator generation.

HARMLESS and the associated tools are still under development and will soon provide a full environment for modeling complex processors. Particularly, pipeline modeling and cycle-accurate simulator generation are already showing promising results [KBB+08]. The generated simulators will also benefit from the work presented in [BBT05], allowing to extract higher-level information – such as stack usage and task scheduling analysis – from a low-level software simulation. But obviously, modeling a processor core is not enough to capture all the useful information related to a hardware platform. A preliminary study and implementation is under way to complete HARMLESS models with abstractions of memory devices, peripherals and busses.

Finally, HARMLESS has proved to be an interesting case study in Model-Driven Engineering. Metamodeling and model transformation have had a strong positive impact on the development, evolution and maintenance of the HARMLESS language and the simulator generator.

References

[Bas01] S. Basu. High Level Synthesis from Sim-nML Processor Specifications. Master's thesis, Dept. of Computer Science and Engineering, Indian Institude of Technology, Kanpur, aug 2001.
citeseer.ist.psu.edu/basu99high.html

[BBH+94] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer. The MIMOLA Language - Version 4.1. Technical report, Computer Science Dpt., University of Dortmund, sep 1994.
citeseer.ist.psu.edu/bashford94mimola.html

[BBT05] M. Briday, J.-L. Béchennec, Y. Trinquet. Task Scheduling Observation and Stack Safety Analysis in Real Time Distributed Systems Using a Simulation Tool. In *10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05)*. september 2005.

[Bha01] S. Bhattacharya. Generation of GCC Backend from Sim-nML Processor Description. Master's thesis, Dept. of Computer Science and Engineering, Indian Institude of Technology, Kanpur, jul 2001.
citeseer.ist.psu.edu/bhattacharya01generation.html

[BJ05] J. Bézivin, F. Jouault. Using ATL for Checking Models. In *Proc.International Workshop on Graph and Model Transformation (GraMoT)*. 2005.

[Fre93] M. Freericks. The nML machine description formalism. Technical report TR SM-IMP/DIST/08, TU Berlin CS Dept, 1993.

[HG99] A. Halambi, P. Grun. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. European Conference on Design, Automation and Test (DATE)*. mar 1999.
citeseer.ist.psu.edu/halambi99expression.html

[HHD97] G. Hadjiyiannis, S. Hanono, S. Devadas. ISDL: an instruction set description language for retargetability. In *DAC'97: Proceedings of the 34th annual conference on Design automation*. Pp. 299–302. ACM Press, New York, NY, USA, 1997.
doi:http://doi.acm.org/10.1145/266021.266108

[IEE00] IEEE/DASC/VASG. Draft IEEE Standard VHDL Language Reference Manual. Technical report, 2000. IEEE P1076.2000/D3.

[JBK06] F. Jouault, J. Bézivin, I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc. fifth international conference on Generative programming and Component Engineering (GPCE)*. Pp. 249–254. 2006.

[JK05] F. Jouault, I. Kurtev. Transforming Models with ATL. In *Proc. Model Transformations in Practice Workshop at MoDELS*. 2005.

[KBB+08] R. Kassem, M. Briday, J.-L. Béchennec, Y. Trinquet, G. Savaton. Simulator Generation Using an Automaton Based Pipeline Model for Timing Analysis (submitted). In *Proc. International Workshop on Real Time Software (RTS)*. oct 2008.

[PHZM99] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr. LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Design Automation Conference*. Pp. 933–938. 1999.
citeseer.ist.psu.edu/pees99lisa.html