Invited Paper

The Extendable Translating Instruction Set Simulator (ETISS) interlinked with an MDA Framework for fast RISC Prototyping

Daniel Mueller-Gritschneder
Technical University of Munich
Germany
daniel.mueller@tum.de

Martin Dittrich
Technical University of Munich
Germany
martin.dittrich@tum.de

Marc Greim[*]
Technical University of Munich
Germany
marc.greim@mytum.de

Keerthikumara Devarajegowda
Infineon Technologies AG
Germany
keerthikumara.devarajegowda@infineon.com

Wolfgang Ecker
Infineon Technologies AG
Germany
wolfgang.ecker@infineon.com

Ulf Schlichtmann
Technical University of Munich
Germany
ulf.schlichtmann@tum.de

ABSTRACT

This paper describes the Extendable Translating Instruction Set Simulator (ETISS). In addition to binary translation, ETISS features a plugin mechanism that allows to quickly include new functionality into the translation stage, the simulation loop, during accesses to the memory or whenever an interrupt is received. ETISS targets to become an advanced industrial-strength ISS with special focus on virtual prototypes (VPs) written in SystemC/TLM. In this paper, we will show examples of ETISS Plugins, which include tracing tools, SystemC interfaces, closey-coupled peripherals or triggers for fault injection. A major drawback of developing a new binary translator such as ETISS is its lack of support for a variety of instruction set architectures (ISAs). At the moment ETISS supports the open-source OpenRISC or1k and partly RISC-V ISAs. Yet, in order to overcome this problem, we developed a toolchain to generate the binary translation stage for different ISAs following the MDA concept based on meta-modeling and code generation. It is planned to make ETISS available as an open-source tool to the research community.

CCS CONCEPTS

•**Computer systems organization →Embedded systems;**

KEYWORDS

RISC-V, Instruction Set Simulator, Code Generation, ISA

1 INTRODUCTION

In modern System-on-chip (SoC) design, the development of the software is carried out in parallel with the development of the hardware. For this, software developers emulate the software on a computer model of the hardware, referred to as virtual platform or virtual prototype (VP). Next to being platforms for SW development, VPs also can be used for architectural design space exploration and early performance evaluation. Even when a physical hardware of the SoC becomes available, SW development on top of a VP has the advantage of better debuggability due to higher observability of internal states and signals.

VPs must be sufficiently detailed to be able to execute the target software for the SoC. For this, the software-accessible parts of the SoC are modeled. This includes buses, interrupt lines, memories and memory-mapped registers in peripherals, so-called special-function registers (SFRs). Here, SystemC/TLM has emerged as a quasi-standard modeling language and simulator. Additionally the target processor is modeled sufficiently accurately to execute target instructions. The processor is usually the bottleneck in the VP limiting the simulation speed. Therefore fast instruction set simulators (ISSs) are used. Yet in terms of ISSs used in VPs, there is no common standard. There exist ISS-based processor models that are provided as part of commercial SystemC development environments such as Synopsys Virtualizer [16]. On the other hand, there exist open software cross-platform emulators such as Qemu [13] or semi-open ones such as OVP [12]. A major advantage of these emulators is that they already come with support for various processors' instruction set architectures (ISAs). Some are specific to a certain instruction set architecture while others support many different ISAs.

The disadvantage of these solutions is that they are either not targeting VP simulation or are not completely open. To implement and validate advanced analysis methods, one often wants to introduce new functionality into the processor model. This requires often to "hack" the simulation loop of the used ISS. For closed-source ISSs this is not possible. Even for the open emulators this can be very hard, as the code is often quite complex and highly optimized for simulation speed. In order to be able to add features to an emulation platform, we developed the Extendable Translating Instruction Set Simulator (ETISS). Its main features are (1) that it can be easily integrated in SystemC/TLM-based VPs, (2) that it uses state-of-the-art fast dynamic binary translation based on just-in-time (JIT) compilation, but most importantly, (3) that it provides a mechanism to quickly introduce new functionality into the simulation loop using so-called Plugins. The novel Plugin mechanism provides several

[*]Marc Greim is now with Intel, Munich, Germany

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RSP'17, Seoul, Republic of Korea
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5418-9/17/10…$15.00
DOI: 10.1145/3130265.3138858

hooks during the translation and simulation stage. These hooks allow the introduction of new functionality during the translation stage, before execution of a block of instructions, at load and store accesses and when an interrupt is received. Additionally, the ISA of the processor is also included as an architectural Plugin, allowing ETISS to model different ISAs and micro-architectures.

ETISS for now only fully supports the OpenRISC1000 architecture running the or1k instruction set [6]. In order to make ETISS widely usable in research but also in industry, we additionally developed a code generation tool to quickly model new ISAs and generate the required architectural Plugin for ETISS from a compact ISA specification. This includes the decoder, instruction behavior, architectural registers, exception behavior and closely-coupled peripherals. The code generation is based on a model-driven architecture (MDA) principle, such that the same source can be used to generate the ETISS Plugin, a documentation of the ISA or even an RTL implementation of the processor for fast RISC prototyping.

The remainder of the paper is structured as follows. First we discuss related work. Then we introduce ETISS internals and the new Plugin mechanism. Then, we will show the MDA-based code generation approach. Finally, we give some experimental results for benchmark programs.

2 RELATED WORK

Just-in-time (JIT) compilation is an established technique [2]. For executing cross-compiled code, the straight-forward approach is to fetch, decode and execute each instruction individually during software execution. This approach is implemented in so-called interpreting ISSs. To speed up the process, already decoded instructions can be cached using the pre-decode principle. Instead of decoding the instruction again, the cached instruction is executed directly. In dynamic binary translation this principle is extended to caching and executing a full block of instructions in a single call [7, 15]. Usually, then this is referred to as a translating ISS.

There is a wide range of tools available for virtual prototyping and for simulation of embedded processors. Dynamic binary translation for fast instruction set simulation is most notably implemented in the open Qemu emulator [13]. Qemu is written in C and provides its own class-like C constructs to describe the peripherals and memories. A SystemC/TLM Wrapper is available called Qbox [11]. In contrast, the Gem5 simulator [10] focuses on micro-architecture simulation, e.g., involving details on the pipeline. Commercially available tools include processor models as part of OVP [12], Synopsys Virtualizer [16] or Windriver Simics [1]. There exists a huge range of other processor simulators, yet, mostly there is, to the best of our knowledge, at the moment no ISS readily and openly available that offers the same easily accessible flexibility as provided by ETISS.

A completely different approach is taken in so-called host-compiled simulation. Here, the code is not cross-compiled. This improves performance but advanced methods are required to maintain accuracy [3, 9].

3 ETISS DESCRIPTION

ETISS is an instruction set simulator based on dynamic binary translation. C is used as the intermediate format between the target processor instructions and the simulation host instructions. It's main novelty is a sophisticated Plugin mechanism to quickly introduce new functionality into the ETISS simulation loop. In the following we will first describe the main simulation loop and then give details on the Plugin features.

3.1 ETISS Binary Translator

Fig. 1 shows the simplified program flow of ETISS's simulation loop. Clearly, it follows the standard flow for dynamic binary translation:

First a block of instructions to execute is identified using the current program counter (PC) value and it is checked whether a translation block for this PC is cached.

If no translation block is cached, then a new translation block is loaded by fetching instructions starting from the current PC value. A translation block ends either at a control flow instruction (branch, jump, arithmetic operation with PC as target, etc.) or when a maximal configurable block size is reached. Please note, a dynamic binary translator usually works at the granularity of instruction blocks, hence, any system will first observe all instruction fetches for the translation block and subsequently the data accesses when the block is executed. After the translation block is fetched, it is translated. ETISS uses C as its intermediate format. First the instructions inside the block are decoded. For each instruction, a C code section is generated that describes the behavior of the instruction. The C code sections are united in a single function that implements the behavior for this translation block. The translation block does not correspond directly to the basic blocks of the control-flow graph of a program because during translation it is not clear whether there exists another block that branches into the middle of the translation block. In order to account for this, a switch statement to jump to any PC in the block is included that allows to start execution in the middle of the translation block.

Finally, the C-Code is compiled into a shared library object or in-memory using the JIT compilation paradigm. The library is then dynamically loaded into ETISS such that the function for the block can be called. Then, the block is registered with its PC range in the translation cache. In case there already exists a translation block in the cache, the translation process is omitted. Hence, simulation is much faster for code that repeatedly calls the same code sections as translation is done only once.

After the translation stage, the code gets executed by calling the generated C function from the compiled shared library or the in-memory compiled code of the translation block. The C statements change the architectural state of the processor according to the instruction behavior. Whenever there is a load or store access, the block issues a read or write request to a simple standardized system interface. Finally, it is checked whether the system generated an exception (e.g. bus errors) or whether there is an interrupt pending. Whenever there is an exception generated during the execution of instructions, the exception behavior of the processor is called. Finally, the next translation block is determined based on the PC value. This stage is again accelerated by block chaining, which allows to omit the cache look-up.

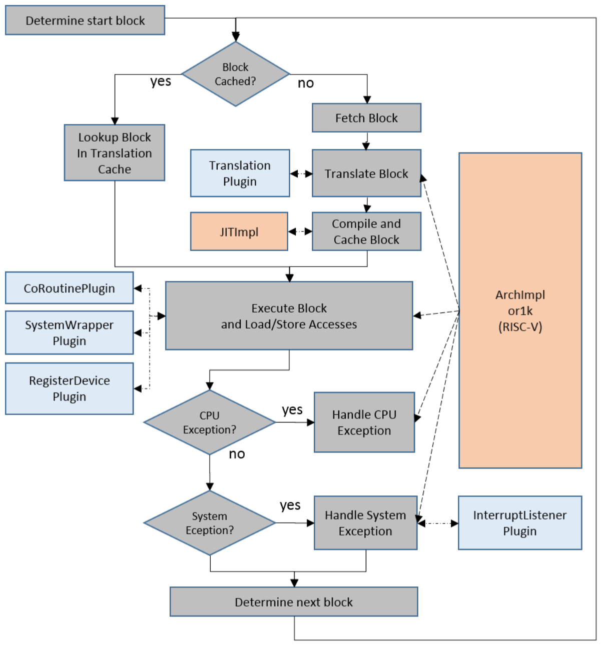

Figure 1: The simplified ETISS Binary Translation Loop

3.2 ETISS Plugin Mechanisms

ETISS offers a range of different Plugin types, which we will shortly describe in the following.

TranslationPlugin: An arbitrary number of *TranslationPlugins* can be registered in ETISS. Before translating a block, ETISS will check for and execute any registered *TranslationPlugins*. A *TranslationPlugin* can be used to add code to the C function of an translation block or to the C Code section of any instruction inside the block. The *TranslationPlugin* itself can have a state using a struct as member variable. This, e.g, allows to implement an instruction counter or other profilers, whose states are updated with extra code added in each C code section of the block.

CoRoutinePlugin: An arbitrary number of *CoRoutinePlugins* can be registered in ETISS. Before executing a block, ETISS will check for and execute any registered *CoRoutinePlugin*. *CoRoutinePlugins* can have arbitrary code and member variables. This allows to model complex behavior. *CoRoutinePlugins* can be used for example to model closely-coupled peripherals such as interrupt controllers or timers. At the start of a basic block the execution of the *CoRoutinePlugin* can update the timer, check for a timer overflow or for external interrupts.

SystemWrapperPlugin: An arbitrary number of *SystemWrapperPlugins* can be registered in ETISS. Before calling the external read or write functions of the system interface, ETISS will first check for any registered *SystemWrapperPlugins* and call the read or write function of the *SystemWrapperPlugin*. This allows to handle store or load operations inside ETISS. This can be used e.g., to model memory-mapped closely-coupled peripherals in ETISS.

InterruptListenerPlugin: The *InterruptListenerPlugin* is notified whenever an interrupt from the system is changing its status. It can be used to override the standard interrupt handling of the target processor.

RegisterDevicePlugin: An arbitrary number of *RegisterDevicePlugins* can be registered in ETISS. The *RegisterDevicePlugin* is registered for a subset of the architectural registers of the processor. Whenever the values of the registers are changed, the *RegisterDevicePlugin* is notified. The *RegisterDevicePlugin* can be used to model closely-coupled peripherals that are configured by the processor via special instructions accessing directly special-purpose registers (SPRs).

ISA Plugin *ArchImpl*: One ISA Plugin *ArchImpl* is registered in ETISS per core. The ISA Plugin is the description of the processor's ISA. This includes all general-purpose registers (GPRs) and special-purpose registers (SPRs) of the processor. Additionally it includes a description of the instruction binary encoding and the instruction behavior. Finally, it includes a description of the processor's exception handling behavior and reset behavior. The ISA Plugin is used throughout the binary translation loop. In the translation stage it is used to decode and translate the target instructions. In the execution stage, it keeps the processor state and is called whenever there is an exception. Finally, it is called whenever there is an exception from the system or from *CoRoutinePlugins*, e.g., an interrupt or bus access error. The ISA Plugin needs to be written for each ISA. This requires high effort, even for simple ISAs. Therefore we will introduce in Section 4 a MDA-based tool to generate the ETISS ISA Plugin. At the moment, or1k is fully supported and RISC-V support is under development.

JIT Plugin *JitImpl*: One Just-in-time compilation (JIT) Plugin *JitImpl* is selected for each ETISS instance. It compiles the C functions of the translation blocks (generated from the ISA meta-model) into shared libraries. At the moment ETISS supports JIT based on gcc, CLANG and tcc. The gcc compiler requires to write the C-code into files for compilation while CLANG and tcc support in-memory compilation. The tcc compiler translates fastest but the compiled code is least optimized such that the execution of an translation block is slower.

3.3 Examples of ETISS Plugins

Our chosen strategy was to model within ETISS all parts of the processor specified in the ISA or respective architectural manual. This does not only include the instructions but often also closely-coupled peripherals. These closely-coupled peripherals are not involved in the instruction execution but are part of the processor core. Typical examples in embedded processors are programmable timers, programmable interrupt controllers or processor configuration units. The Plugin mechanism of ETISS allows for easy integration of these peripherals. In the following, we will shortly outline some examples for using the Plugin mechanism for peripherals and other functionality.

Instruction Tracing: Instruction Tracing can be implemented as a *TranslationPlugin*. It adds code to the C-Code section of the instruction to print a selected subset of the processor state to an instruction trace file. For example one can print the PC trace together with a subset of GPR values for each instruction.

Logger The Logger is a simple example of a *SystemWrapperPlugin*. Whenever there is an access to a certain memory address, the Logger outputs the data to the terminal instead of forwarding the access to the system. This allows for simple software debugging and testing.

or1k Timer and Interrupt Controller The OpenRISC architecture specifies a timer and programmable interrupt controller (PIC). Both peripherals are configured and read-out via SPRs by move-to-spr or move-from-spr instructions. Hence, both peripherals need to implement the *RegisterDevicePlugin*. This gives them the capability to react to configurations on the SPRs as they get notified on register updates. Additionally, both peripherals need to implement the *CoRoutinePlugin*. This allows them to raise an exception before the next block is executed such that the processor can trigger the exception handler.

or1k GDB Server With the combination of *TranslationPlugin*, *CoRoutinePlugin* and *SystemWrapperPlugin* a GDB Server is implemented. For being able to step through all instructions of the program, the *TranslationPlugin* inserts return statements after instructions. The return statements are executed when a breakpoint is reached. After this return of the execution the *CoRoutinePlugin* gets active for the communication with the gdb client. The *SystemWrapperPlugin* is used to catch memory breakpoints.

Memory-mapped peripherals Some architectures define memory-mapped timers and interrupt controllers as closely-coupled peripherals. Therefore they need to implement the *SystemWrapperPlugin* such that accesses to these peripherals are handled internally by ETISS and not forwarded to the system. As for the or1k case, they also need to implement the *CoRoutinePlugin* such that the processor checks for exceptions before executing a translation block.

Multi-level Simulation For a research project on fault injection we implemented a rather complex Plugin. It allows to switch between the instruction set simulation of the OpenRISC processor and an RTL simulation of the processor, a so-called multi-level simulation. This required to switch off the ISS and connect the system to the RTL model. Here we made use of the *SystemWrapperPlugin*, *CoRoutinePlugin* and *InterruptListenerPlugin* mechanism. We were able to implement the complete functionality as ETISS Plugin such that we did not need to "hack" the original ETISS code.

3.4 ETISS Implementation and Usage

ETISS Installation: ETISS is implemented in C++11. It is compiled as a shared library that can be linked to any C++ virtual prototype. In general, all Plugins are implemented as classes. Simply by deriving a child class of the parent Plugin class, one can implement a customized Plugin behavior. Plugins can either be compiled within the ETISS library or as part of the virtual prototype that links the ETISS library. This gives flexibility in sharing the Plugins. A VP-specific Plugin may remain with the VP, a general Plugin used in several VPs within the ETISS library. A Plugin compiled with ETISS does not cause performance overheads, as it needs to be registered in the VP to become active. No ETISS nor VP re-compilation is required to de/activate Plugins.

SystemC/TLM Interface: ETISS can run as a standalone C++ program. Yet, we specifically targeted to use ETISS as processor model in Virtual Platforms based on SystemC/TLM. Therefore, there already exists a SystemC wrapper for ETISS. ETISS can have several instances in a SystemC environment such that also multi-processor systems can be simulated. Synchronization points can be freely chosen and ETISS has a variable to keep track of simulation time implemented for this purpose. Our SystemC interface synchronizes at block boundaries and data accesses. It also supports the SystemC Quantum Keeper for temporal-decoupled simulation. ETISS is thread-safe such that it would allow to advance simulation of all ETISS cores and SystemC in parallel as different simulation-host-OS threads. This should offer an advantage in simulation performance for multi-processor systems but we have not yet implemented it.

4 MDA-BASED CODE GENERATION

4.1 Meta-model, Model, MDA and Code-Generation

Meta-modeling facilitates an efficient way of code generation and has been widely used in Infineon to raise design productivity [5]. An inhouse generation framework based on meta-modeling has been developed that heavily utilizes Python and Mako templates. This automation framework is effecively used for more than 100 applications ranging from digital hardware — with or without power reduction techniques — over analog and firmware to test and safety. Over 100 different automation solutions achieve a high productivity benefit in various designs.

The existing automation framework is currently enhanced following the idea of OMG's model-driven architecture (MDA) vision [4, 14]. As shown in Figure 2 on the left hand side, three different models, which in this context are meta-model instances, are involved:

(1) The model-of-things (MoT) is a specification level model that captures abstract requirements. For RISC processors, it mainly defines the major objects and the instructions. The MoT corresponds to computational-independent model (CIM) in the original MDA definition [8].
(2) The model-of-design (MoD) is a micro-architecture level model. It is defined by a so called template-of-design (ToD), which defines the general architecture template and the adaptations needed to enable the architecture to support the intended behavior. Therefore, the ToD extracts information from the MoT in order to build the MoD. The ToD is Python code using the generated model APIs to read and write models. The ToD shows some similarities with hardware description languages but allows to build the MoD in a much more flexible way due to the underlying Python language. The MoD corresponds to platform-independent model (PIM) in the original MDA definition [8].
(3) The model-of-view (MoV) is essentially an abstract syntax tree. A writer finally generates HDL RTL code out of it by considering layout and formatting rules. In the original MDA [8] the platform-specific model (PSM) is introduced, which relates to this MoV. The translation of MoD to MoV is a one-time approach, i.e. the designer can focus on architecture design when generating the design.

In order to verify the generated code as well as the generation process, the framework is further enhanced to apply OMG's model-driven architecture vision to property generation. Similar to the automated design generation, property generation starts with the specification level model MoT. Therefore, the generated properties verify the generation process as such. As shown in Figure 2 in the middle, a property language-independent model-of-property (MoP)

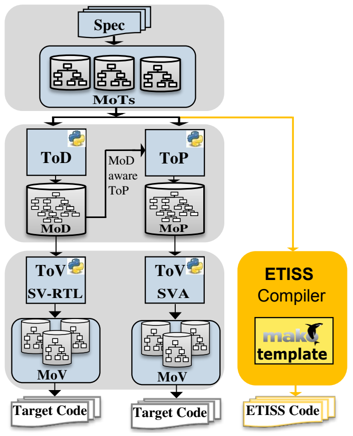

Figure 2: Generating ETISS files from a generic MDA based code generation infrastructure.

is introduced as well as a model-of-view, which relates in this case to a property language such as SVA or PSL.

The described process has been used to generate VHDL RTL code of a five-stage pipelined implementation of a CPU supporting RISC-V RV32I instruction set. Here, the MoT is a formalized version of the RV32I ISA and the ToD mainly creates a 5-stage architecture template. Adaptations are done only for the instruction decoder and the operations supported in the EX-stage.

4.2 The MetaRisc Meta-Model

The MoT of a RISC-Processor mainly models the ISA. Architecture considerations are not considered and the instruction fetch-decode-execute cycle is implicitly assumed. Figure 3 shows the UML class diagram (also called Meta-Model of MoT) describing the MetaRISC Metamodel.

The model definition consists of three parts, the instruction encoding (shown on the left, describes how individual instructions are encoded), the instruction behaviour and the architectural state (ObjectProperties: contains all stateful elements of the CPU). This is extended by instruction parameter definitions and details of the behavior description. A dataflow semantic is used for the latter ones.

4.3 ETISS ISA Plugin

Having a generation framework that supports systematic generation of RISC-CPUs and having the ISA defined in a formal way, it is obvious to approach generation of the ETISS ISA Plugin starting from the same model as well (MoT). As shown in Figure 2 on the right, the ETISS ISA Plugin generation follows a different flow than the generation of the design and the properties. At the moment, the ETISS code is directly generated using Mako templates since it does not need any kind of architecture definition.

In the long run, ETISS ISA description shall be generated as an intermediate. The ETISS Compiler itself has models involved and the input format, the ETISS ISA description, is semantically closely related to the MoT.

5 EXPERIMENTAL RESULTS

5.1 Benchmark programs

Table 1 shows the simulation performance for two selected benchmark programs. The simulation performance is usually given as the number of million instructions of the modeled target processor simulated within one second execution time of the simulation host [MIPS]. The host is a Intel Core i7 (3.4 GHz) with 8GB RAM. As can be seen, the performance for the *edge* benchmark is quite low. This is due to the fact that the program is very short. The initialization overhead and program translation are dominant. TCC-based JIT offers the best performance as it offers the fastest translation due to little code optimization. The *isort* benchmark executes far more instructions. This improves the simulation performance significantly. Overall, the performance improves with the length of the SW scenario and with the percentage of repeated execution of cached translation blocks vs. newly translated blocks.

5.2 SystemC/TLM VP of MCU

ETISS's primary target is its use inside a full-system VP. For demonstration, ETISS is integrated in a VP of a simple Microcontroller Unit (MCU) executing an embedded control application. For this, the target processor executes a PI control algorithm in software. Within a control period, the algorithm reads two sensor values, computes the actuator values based on the PI control law and writes the control value to the acutator. Buses, memories, sensors, actuator and plant are modeled in the SystemC/TLM loosely-timed modeling style. The system model is inspired by an automotive adaptive cruise control (ACC) system. Table 1 shows the simulation performance. Obviously it performs better than the benchmarks. This is due to the fact that the model executes code for a very long software scenario (20s). The software only needs to be translated once. During the sleep phase between computing two values the CPU is running an idle loop with no memory accesses. This allows for fast simulation compensating the overhead caused by the synchronization and accesses to SystemC/TLM modules as well as the Plugins for or1k peripherals. The SystemC interface is also very lightweight such that overheads are minimized. CLANG is the fastest JIT due to its sophisticated code optimization that in this case dominates the translation overhead.

6 CONCLUSIONS

In this paper ETISS and MDA-based code generation was introduced. We plan to use both approaches to quickly evolve ETISS into an industrial-strength processor simulator. Next to the instruction part of the ISA, it is also planned to generate ETISS peripheral models with the MDA-based approach. Finally, we plan to provide ETISS as open-source tool, once the basic ISAs are well tested and documented.

ACKNOWLEDGMENTS

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) in the project ConFIRM under grant 16ES0566.

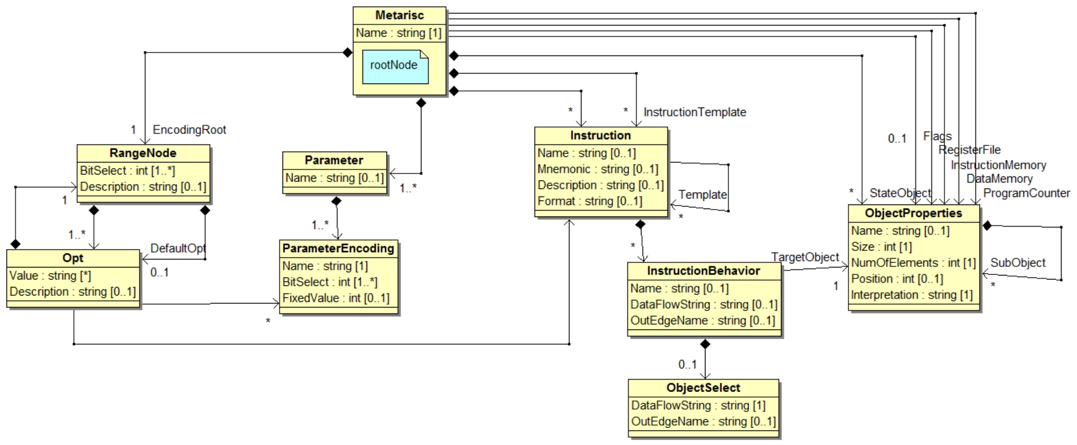

Figure 3: The MetaRisc Meta-Model

Table 1: Simulation Speed of ETISS

SW			JIT			ARCH	Other Plugins				System		Simulation Performance		
edge	isort	PI controller	CLANG	GCC	TCC	OR1K	Logger	OR1KTimer	PIC	ResetHandler	Standalone	SystemC	[MIPS]	#Instr	Sim Time [s]
✓					✓	✓					✓		1.42	305E3	0.21
✓					✓	✓	✓				✓		1.39	305E3	0.22
✓				✓		✓	✓				✓		1.19	305E3	0.25
✓			✓			✓	✓				✓		0.27	305E3	1.1
	✓				✓	✓					✓		13.9	2.8E6	0.20
	✓				✓	✓	✓				✓		13.6	2.8E6	0.22
	✓			✓		✓	✓				✓		11.9	2.8E6	0.24
	✓		✓			✓	✓				✓		3.45	2.8E6	0.83
		✓			✓	✓		✓	✓	✓		✓	49.5	2.0E9	40.4
		✓		✓		✓		✓	✓	✓		✓	54.7	2.0E9	36.5
		✓	✓			✓		✓	✓	✓		✓	59.9	2.0E9	33.3

REFERENCES

[1] Daniel Aarno and Jakob Engblom. 2014. *Software and System Development using Virtual Platforms: Full-System Simulation with Wind River Simics.* Morgan Kaufmann.
[2] John Aycock. 2003. A Brief History of Just-in-time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077
[3] Oliver Bringmann, Wolfgang Ecker, Andreas Gerstlauer, Ajay Goyal, Daniel Mueller-Gritschneder, Prasanth Sasidharan, and Simranjit Singh. 2015. The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition.* EDA Consortium, 1698–1707.
[4] W. Ecker and J. Schreiner. 2016. Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC).* 1–6. https://doi.org/10.1109/VLSI-SoC.2016.7753576
[5] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. 2014. The meta-modeling approach to system level synthesis.. In *Design, Automation & Test in Europe Conference (DATE).*
[6] Damjan Lampret et al. 2014. OPENCORES.ORG, OpenRISC 1000 Architecture Manual, Architecture Version 1.1. (2014). http://opencores.org/or1k/
[7] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. 2009. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '09).* ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/1629435.1629446
[8] Liangora Research Lab. [n. d.]. What is MDA? Why considering BNPM. ([n. d.]). https://research.linagora.com/pages/viewpage.action?pageId=3639295
[9] Daniel Mueller-Gritschneder and Andreas Gerstlauer. 2017. Host-Compiled Simulation. *Handbook of Hardware/Software Codesign* (2017), 1–27.
[10] N.N. [n. d.]. The Gem5 Simulator. ([n. d.]). http://gem5.org/Main_Page
[11] N.N. [n. d.]. GreenSoCs QBOX. ([n. d.]). https://www.greensocs.com/get-started#qbox
[12] N.N. [n. d.]. Imperas OVP. ([n. d.]). http://www.OVPworld.org
[13] N.N. [n. d.]. Open Source Processor Simulator QEMU. ([n. d.]). http://wiki.qemu.org/
[14] "OMG". 2016. MDA - The Architecture of Choice for a Changing World. (2016). http://www.omg.org/mda/
[15] Frédéric Pétrot, Luc Michel, and Clément Deschamps. 2017. *Multi-Processor System-on-Chip Prototyping Using Dynamic Binary Translation.* Springer Netherlands, Dordrecht, 1–27. https://doi.org/10.1007/978-94-017-7358-4_20-1
[16] Synopsys. [n. d.]. Virtualizer Tool Website. ([n. d.]). https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html