A Model-based Transformation Approach to Reuse and Retarget CASM Specifications

Philipp Paulweber and Uwe Zdun

University of Vienna
Faculty of Computer Science
Währingerstraße 29, 1090 Vienna, Austria
{philipp.paulweber,uwe.zdun}@univie.ac.at

Abstract. The Abstract State Machine (ASM) theory is a way to specify algorithms, applications and systems in a formal model. Recent ASM languages and tools address either the translation of ASM specifications to a specific target programming language or aim at the execution in a specific environment. In this work-in-progress paper we outline a model-based transformation approach supporting (1) the specification of applications or systems using the Corinthian Abstract State Machine (CASM) modeling language and (2) retargeting those applications to different programming language and hardware target domains. An intermediate model is introduced, which not only captures software-based implementations, but also the generation of hardware-related code in the same model. This approach offers a new formal modeling perspective onto modular, reusable and retargetable software and hardware designs for the development of embedded systems. We provide a short overview of our CASM compiler design as well as the retargetable model-based approach to generate code for different target domains.

1 Introduction

Since 1995 where Gurevich has described the Abstract State Machine (ASM) theory [1], many approaches have been proposed to interpret, execute, translate, verify and validate ASM specifications (summarized by Börger [2]). Generally speaking all available (public) tools either aim to integrate an ASM language into a specific (software) platform system/framework or focus on a domain specific purpose. We want to enlarge the scope of ASM language tools and provide a general purpose modeling system for the Corinthian Abstract State Machine (CASM) modeling language (introduced by Lezuo et al. [3]). Such a system will enable us to specify arbitrary applications/systems in this language and translate them into one or multiple programming language and hardware target domains. To the best of our knowledge, such a generic translation does not yet exist.

Furthermore, not only is the focus of our investigation not limited to translations to several software environments, it also includes the idea to translate CASM specifications to different Hardware Description Language (HDL) contexts. This will enable us to even describe electronic circuit designs with CASM specifications and will result in a broad range of applications from specifying small embedded applications to describing Reduced Instruction Set Computing (RISC) microprocessor or even complete System-on-Chip (SoC) designs in a formal way.

1.1 Modeling Language and Compiler

The CASM modeling language was designed and used by Lezuo et al. [3] to describe the semantics of machine languages. Moreover, they performed compiler correctness proofs through the usage of the ASM machine models and compiled specifications written in this language into efficient C/C++ applications [4]. Unlike other ASM specification languages such as AsmL [5] or CoreASM [6], CASM currently consists of a small grammar and a static, strong type-system, and it only supports a static subset of the CoreASM modeling language. The static, strong type system allows to optimize such specifications. Initially, the syntax of CASM followed CoreASM, but over time it diverged significantly (differences to other ASM modeling languages are described by Lezuo et al. [3]).

Due to the (currently) small grammar, the optimization possibilities and simplicity, the CASM modeling language suits well in our investigation to retarget ASM specification. Before we go into details, let us review the design of the compiler infrastructure proposed by Lezuo et al. [4]. Figure 1 depicts the translation process.

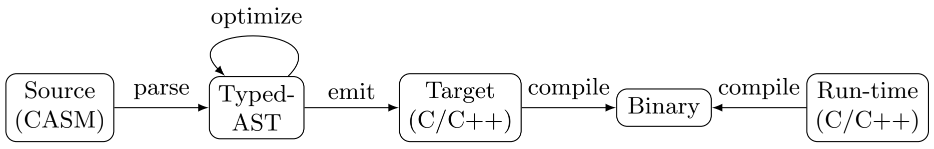

Fig. 1. CASM Compiler with C/C++-Backend

The parsed CASM specifications are transformed into an Abstract Syntax Tree (AST), and after that type checks and type annotations are performed. Several static optimizations are performed to eliminate run-time overheads. All transformations which need run-time specific calculations and knowledge are redundantly implemented in the AST-based optimizations. The compiler directly emits C/C++ code in the next step, which then gets compiled and linked against the statically implemented run-time.

1.2 Motivation and Goal

The design in Figure 1 is not a retargetable infrastructure. That is, in this design, the existing code emitter and run-time implementation need to have to be checked for correctness and tested that the execution and calculation of the generated C program equals the specified CASM input specification. If we would retarget this design to different software or hardware environments, we would have to check again for each new environment the code emitter and run-time implementation that the calculation behavior of the generated target equals the specified CASM input specification.

The emitting stage depicted in Figure 1 is the main focus of our approach. Our solution to this *retargetable CASM specification problem* is to abstract the run-time and the emitted code in a specific calculation model. This will allow us to check the transformation from the CASM model to this specific computational model once. And for every new target environment (software or hardware) we add to the compiler, only the transformation has to be checked from the specific calculation model to the new target environment. Therefore, we can develop several different code emitter implementations hand-in-hand with one(!) run-time implementation and one(!) CASM transformation implementation.

This approach enables us to create and generate reusable and retargetable software or hardware artifacts. Those artifacts are self-contained because in our approach we even include the full CASM run-time in the generated artifacts. Hence, the generated artifacts of CASM input specifications can be deployed without further libraries or dependencies. The latter is very important when it comes to hardware-related generated code, because it will not only ease the integration in other hardware designs, but will also allow HDL compilers to fully optimize the generated HDL code on module level.

2 Retargetable Approach and Models

The design of our CASM implementation follows a strict model-based transformation approach to overcome the *retargetable CASM specification problem*. Figure 2 depicts our model-based transformation approach where we introduce two models – the Intermediate Representation (IR) and the Emitting Language (EL) model.

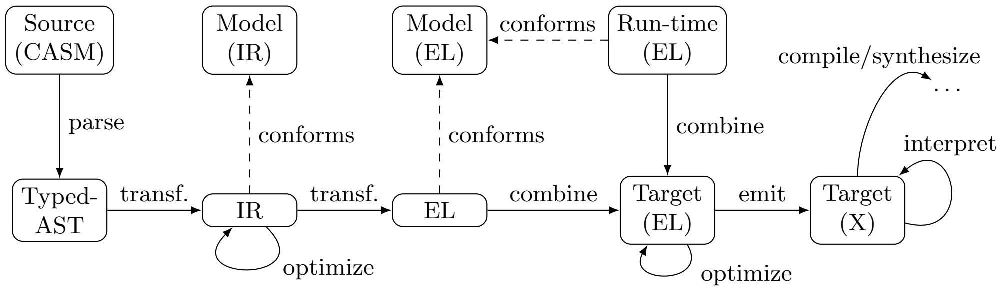

Fig. 2. CASM Compiler with Model-based Transformation

2.1 Intermediate Representation Model

The IR is a full CASM semantics aware model which will be used to analyze and optimize the input specification. An instance of this model is created during the AST to IR transformation (first transformation step in Figure 2). The IR model consists of two important characteristics – parallel/sequential Control Flow Graph (CFG) (introduced by Lezuo et al. [4]) and explicitly modeled operations which are not covered in the AST representation from Lezuo et al. [4] e.g. the location of a ASM state function. The proposed ASM specific *lookup and update elimination* optimizations by Lezuo et al. [4] are planned to be implemented at this level. Software back-ends will profit from those optimizations to be able to execute the specifications much faster (as shown in [4]). Furthermore, we strongly believe the hardware back-ends will benefit from the proposed optimizations too. Because the generated HDL code will result in a less complex digital design by reducing the number of performed calculations just like it applies to the generated software code.

2.2 Emitting Language Model

The EL model is the main contribution in this paper. An instance of this model is created during the IR to EL transformation of the IR instance (depicted in Figure 2). It allows us to express the CASM run-time and the CASM input specification in a CASM semantics unaware fashion. Thereby we are forced to find generic abstract language constructs for the EL model which allow us to express calculations, procedures and sequential and parallel execution behavior.

The EL model is designed to make the mapping to different software/hardware targets easier, but this generic abstraction does not come without limitations. For example the only data type allowed in the EL model is a bit-precise integer value to enable a clean translation to HDL data types. To represent complex or compound data a structure concept is available in the EL model as well to create records of several bit-precise integer values.

The overall model construct is a *Module* which can contain besides *Constants*, *Variables*, *CallableUnits* also explicitly defined *Memory* blocks. The *Memory* blocks are used to properly allocate the appropriate amount of wiring and memory storage in the generated HDL designs. The difference between a *Memory* and *Variable* storage is that *Variables* are translated to HDL designs as plain registers and only permit a single write access. Whereas *Memory* blocks permit multiple write access and we assume in the EL model that each write access is mutually exclusive. Latter is important, because the model allows the construction of mixed parallel and sequential *Statement* blocks.

CallableUnits are divided into to procedural constructs – *Functions* and *Intrinsics*. Software back-end languages like C, Python etc. use this differentiation to emit efficient target language code, which can be used by the target compiler/interpreter to optimize the execution of the program. Hardware back-end languages can derive a differentiation between behavioral descriptions and computational logic blocks. At this point, another important EL model characteristic is that a *CallableUnit* does not have a "return" value. All incoming and outgoing data of a *CallableUnit* has to be explicitly defined through "in" and "out" parameters. Hence, software back-ends will use this to generate "call-by-reference" constructs and hardware back-ends generate direct component wiring.

All *CallableUnits* can contain mixed parallel and sequential *Scopes* to define a concurrent and sequential calculation hierarchy. Every *Scope* in the EL model can contain several *Statements*. A *Statement* can either be a "trivial", "branch" or "loop" behavioral container. Every *Statement* consists of a list of *Instructions*, which form the leaf nodes in the EL model and perform the actual operations.

Furthermore, due to the flexibility of the EL model and the possibility of unbounded in time of rule evaluations in the sense of CASM, we decided to translate EL instances in the HDL back-ends to asynchronous digital designs. Hence, every *Function*, *Intrinsic*, *Statement* etc. from the EL model follows a request-acknowledge handshake protocol. Currently we only focus, besides the software C back-end, for the hardware back-ends on the generation of Very High Speed Integrated Circuit Hardware Description Language (VHDL) code with an assumed annotated timing information. The generated designs are validated in a HDL simulator environment. But in the future the generated code shall be synthesizeable to Field Programmable Gate Array (FPGA) boards as well.

2.3 Compiler Design

From the software design point of view of the compiler, both presented models (IR and EL) follow a Single Static Assignment (SSA) based internal representation. They use a similar class design and analyze/transformation pass design proposed by the Low Level Virtual Machine (LLVM) compiler infrastructure by Lattner and Adve [7]. Latter, was used in early experiments to translate the CASM IR model to the LLVM IR model, but due to the retargetable focus for assembly code it turned out that the LLVM IR model was to low-level to realize our retargetable approach. Therefore, we started the design of the EL model.

3 Preliminary Results, Outlook and Conclusion

The current development status of the compiler and the models are in an early state. Major infrastructure and transformation passes are implemented. The optimizations for the IR and EL model are planned. We were able to retarget a small CASM filter application to a valid C program and VHDL digital design. The example application consists of three functions, one rule and two parallel update terms.

The overall goal we want to achieve in our future work is to create at least for four language domains a translation back-end implementation. CASM specifications shall be translated to C11 (native), Python (script), JavaScript (web) and VHDL (hardware). A possible field of application would then be the construction of a new RISC microprocessor design in CASM. The proposed retargetable approach of our modeling system generates then directly an Instruction Set Simulator (ISS) for software debugging, an ISS for integration in a website (e.g. for presentation and testing purposes), and a valid synthesizeable hardware implementation.

We have outlined our CASM based retargetable compiler infrastructure and the model-based transformation approach which will enable the reuse, integration and execution of a single CASM specification in different software and hardware environments through the usage of the EL model.

References

[1] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," *Specification and Validation Methods*, pp. 9–36, 1995.

[2] E. Börger, "The Abstract State Machines Method for High-Level System Design and Analysis," in *Formal Methods: State of the Art and New Directions*, pp. 79–116, Springer, 2010.

[3] R. Lezuo, G. Barany, and A. Krall, "CASM: Implementing an Abstract State Machine based Programming Language.," in *Software Engineering (Workshops)*, pp. 75–90, 2013.

[4] R. Lezuo, P. Paulweber, and A. Krall, "CASM - Optimized Compilation of Abstract State Machines," in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pp. 13–22, ACM, 2014.

[5] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic Essence of AsmL," in *Formal Methods for Components and Objects*, pp. 240–259, Springer, 2004.

[6] R. Farahbod, V. Gervasi, and U. Glässer, "CoreASM: An Extensible ASM Execution Engine," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 71–104, 2007.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization*, pp. 75–86, IEEE, 2004.