CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation

Philipp Paulweber, Emmanuel Pescosta*, and Uwe Zdun

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

Abstract. The Abstract State Machine (ASM) theory is a well-known formal method, which can be used to specify arbitrary algorithms, applications or even whole systems. Over the past years, there have been many approaches to implement concrete ASM-based modeling and specification languages. All of those approaches define their type systems and operator semantics differently in their internal representation, which leads to undesired or unexpected behavior during the modeling, the execution, and code generation of such ASM specifications. In this paper, we present CASM-IR, an Intermediate Representation (IR), designed to aid ASM-based language engineering which is based on a well-formed ASM-based specification format. Moreover, CASM-IR is conceptualized from the ground up to ease the formalization of ASM-based analysis and transformation passes. The feasibility of CASM-IR solving the *uniform ASM representation problem* is depicted. Based on our CASM-IR implementation, we were able to integrate a front-end of our statically inferred Corinthian Abstract State Machine (CASM) modeling language.

Keywords: CASM, Type System, Instruction, Register Machine, Abstract State Machine, Intermediate Representation, Modeling and Specification Language

1 Introduction

In 1995 the Abstract State Machine (ASM) theory has been described by Gurevich [1] as a formal method based on transition rules, states and algebraic functions. ASMs are used to describe formally the evolving of function states in a step-by-step manner. This also explains why ASM theory was formerly called *Evolving Algebra* [2]. Based on the ASM programming language model from Gurevich, several tools with Domain-Specific Languages (DSLs) were created to solve application-specific problems, which were summarized by Börger [3].

* No affiliation. Member of CASM organization epescosta@casm-lang.org

The diversity of ASM-based applications[1] is widespread, ranging from formal specification semantics of programming languages, such as those for Java by Stark et al. [4] or VHDL by Sasaki [5], compiler back-end verification by Lezuo [6], software run-time verification by Barnett and Schulte [7], software and hardware architecture modeling e.g. of Universal Plug and Play (UPnP) by Glässer and Veanes [8], or even Reduced Instruction Set Computing (RISC) designs by Huggins and Campenhout [9].

Despite this diversity in applications, over the past years, different ASM-based language dialect were created to cover single or multiple application specific problem domains. This might not be perceived as a problem, as many *language users* [10] like to choose among multiple language dialects. The problem however is that the *language engineers* [10] craft and design those languages according to the needs of the *language user* and bind their implementations to a specific execution environment technology, instead of generalizing the mathematical foundation of the ASM-based languages in an independent model representation. This, in turn, means that those languages are difficult to integrate with each other [11], cannot easily be based on a common execution environment technology, and establishing a common set of language tools is difficult.

Moreover, the binding to various execution environment technologies introduces undesired and unexpected behaviors, e.g. if the same algorithm so to say is specified with different ASM modeling languages and the model execution leads to different floating point values or depending on the Integer representation to different overflow states. To overcome this *uniform ASM representation problem* a clear, precise, and formal intermediate model has to be introduced, which has the ability to represent various ASM language constructs of different contexts.

The major advantages of such an approach are the generalization of ASM-related analyzes, optimization, and transformation capabilities – first envisaged by Lezuo et al. [12] – in one single uniform model. Furthermore, another benefit for existing ASM languages is to directly reuse the numeric as well as the – proposed by Lezuo [6] – symbolic execution of specified ASM models. A huge disadvantage in the perspective of a *language engineer* is to port existing ASM language implementations to such a uniform ASM model.

This paper focuses on the design, implementation, and integration of an ASM-based Intermediate Representation (IR) model named CASM Intermediate Representation (CASM-IR) to address the *uniform ASM representation problem*. The main contribution of this paper is the definition of a well-formed ASM-based IR model which is independent of language front-ends and provides a well-defined type system, operator and built-in semantics.

This work is organized as follows: In Section 2 we describe our research context and the motivation of this paper. In Section 3 we describe our CASM-IR model. Section 4 presents details about the current implementation and integration of the CASM-IR. Section 5 gives an overview of the related work regarding IR's of other ASM languages and tools. Finally, in Section 6 we conclude the paper and outline the future work.

---

[1] for ASM applications of various domains, see: `http://web.eecs.umich.edu/gasm`

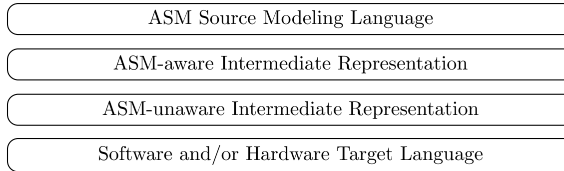

Fig. 1. CASM System Abstraction Layers

2 Motivation

The broader context of our research is the creation of a modern state-of-the-art ASM modeling language implementation named the Corinthian Abstract State Machine (CASM), as well as transformation and deployment of CASM specifications to executable artifacts[2]. The primary application context of this work is the specification of embedded systems in a formal way. However, in the context of CASM, we not merely focus on specific application contexts like embedded systems, but rather aim to describe and specify arbitrary software and/or hardware applications. This overall idea is not new, but our approach to achieve this goal of generic transformations is different from a language engineering perspective, because we set our ASM-based IR into the center of the front-end language development. Other ASM language approaches, which are described in Section 5, do not, because they implement a forward directed transformation from ASM to the desired target language like C or C++. The transformation of ASM source specifications to specific target languages is by no means trivial. It involves the mapping of a mathematical-based specification model to a real executable program, which for itself resides in a specific execution environment.

To overcome this complex transformation, we proposed and followed a model-based transformation approach in our earlier work [13], which defines four abstraction layers (illustrated in Figure 1). At the top resides the *ASM Source Modeling Language* layer that includes besides the language grammar definition the lexer, parser, type inference, type checker, and Abstract Syntax Tree (AST) representation. A parsed input specification gets translated to the next layer, the *ASM-aware Intermediate Representation* layer. At this abstraction layer the CASM-IR, proposed in this paper, is defined. It allows us to analyze, transform and optimize the input specification for ASM related properties.

The CASM-IR gets further transformed in the next layer called *ASM-unaware Intermediate Representation*. At this abstraction layer the transformed specification has no longer any knowledge about the semantics or behavior of ASMs. Therefore it can be analyzed, transformed and optimized for traditional properties like execution speed or program size. In the final layer of Figure 1, the *ASM-unaware Intermediate Representation* is mapped to various *Software and/or Hardware Target Languages*. Those CASM system abstraction layers describe a full transformation of an ASM specification to its desired target language. Due

[2] for CASM project website, see: `http://casm-lang.org`

Fig. 2. CASM System Design (High-Level Overview)

to the proposed layered structure, it is possible to only use a sub-set of the full functionality as well. For example, AST-based execution is used to recursively walk over the in-memory AST representation and interpret the input specification as it was parsed. In the context of AST-based execution, *language engineers* can (re)use the type system from the ASM-aware IR layer and can rely on its defined behavior and semantics. Therefore the CASM-IR can be used not only for transformation and code generation purposes, but also for AST-based interpreter applications. Furthermore, besides ASM-based languages, this proposed IR and its functionality could be used for other functional programming languages as well for their function definitions, type relations, numeric and symbolic computations. The approach to address the *uniform ASM representation problem* – introduced and described in Section 1 – with the CASM-IR raises several concerns regarding its existence and usefulness. First of all, the effort to investigate into such an IR design arises from the fact that accordingly to the state-of-the-art and to our knowledge no comparable IR for ASM languages with the focus on well-formed, reusable, retargetable, and optimizable ASM specifications exist yet. Second, as presented by Lezuo et al. [12], the optimization potential is huge of ASM languages regarding redundancy eliminations, but still not covered and addressed by any ASM language implementation in a unified manner.

3 CASM-IR

This section describes our ASM-based IR design that can be (re)used for designing and building ASM-based and other possible functional related specification languages. Before we go into the details of the model and the format of this IR, we first outline the composition of our CASM system [13]. Figure 2 depicts a more detailed overview of the sketched abstraction layers from Section 2 (see Figure 1). A parsed ASM source – in our case the CASM language – gets translated to an AST representation and necessary type information gets inferred. In order to do so, the CASM-IR – depicted as Model (IR) – needs to provide type information for all possible operators and their type relations, which a

language front-end can use, to implement a type inference pass. Furthermore, the CASM-IR model provides the ability to directly implement AST-based interpreter applications on top of it, because language front-ends can access the implemented run-time of the IR to evaluate expressions and terms.

If the execution shall be done directly using the IR model itself, a language front-end just has to perform a model-to-model transformation from its AST-based representation to an instance of this IR model. At this point the IR can optimize the specification for ASM-related properties fully decoupled from the original input specification in form of an AST representation. Some optimization properties were proposed by Lezuo et al. [12]. Furthermore, then the IR instance can be executed by the run-time implementation of the IR model.

For further processing (code generation) of the specification to a specific programming target language, the IR instance can be transformed into an Emitting Language (EL) model, as proposed in our earlier work [13]. Details about the EL model are out of the scope of this paper.

3.1 Motivating Example

To better understand the solving of the research question regarding the *uniform ASM representation* problem that CASM-IR deals with, we describe a small ASM specification and point out the issues, which are addressed by the CASM-IR design and implementation. Listing 1.1 on Page 6 depicts a valid (high-level) CASM specification of a modeled *swap* algorithm[3]. It defines a rule `swap` (Line 6) and two nullary functions `x` (Line 3) and `y` (Line 4) of result type integer. The `init` (Line 1) defines a single execution agent with starting top-level rule `swap`. Rule `swap` defines a parallel block rule (Line 7-11) and three update rules. The first two update rules (Line 8-9) are producing updates to swap the function values from `x` and `y`. In the last update rule (Line 10), the ASM *program* function gets updated with an undefined value, which results into a termination of the specification, because the single execution agent top-level rule gets set to an undefined value and therefore the ASM execution concludes the model execution.

To get a feel for the expressed *swap* algorithm ASM specification in other ASM languages, we depict three further examples of the same algorithm modeled in *AsmL* [14] (Listing 1.2), *CoreASM* [15] (Listing 1.3), and *Asmeta* [16] (Listing 1.4). Even in this small specification, several behaviors and definitions are implicit and slightly different in the various ASM languages. E.g. the used function `program` (Listing 1.1 at Line 10) is not an explicitly defined function in this valid CASM specification, because this function definition is hidden from the *language user* and it gets implicitly defined, because it depends on an agent type domain. The type relation of this function would be a projection of the current agent type domain to a stored top-level rule, which is similar in the *CoreASM* specification (Listing 1.3 at Line 12). Furthermore, the initialization of this `program` function to the rule `swap` is implicit as well. In CASM and *CoreASM* this is achieved by setting the underlying agent through the `init` definition (Listing 1.1 at Line

[3] for CASM concrete syntax description, see: `http://casm-lang.org/syntax`

```
CASM init swap

function x : -> Integer
function y : -> Integer

rule swap =
{
  x := y
  y := x
  program( self ) := undef
}
```

Listing 1.1. Swap Example (*CASM*)

```
var x as Integer
var y as Integer

swap()
  x := y
  y := x

Main()
    swap()
  step
    // terminates after this step
```

Listing 1.2. Swap Example (*AsmL*)

```
CoreASM swap
use StandardPlugins
init swap

function x : -> Integer
function y : -> Integer

rule swap =
  par
    x := y
    y := x
    program( self ) := undef
  endpar
```

Listing 1.3. Swap Example (*CoreASM*)

```
asm swap
import ../STDL/StandardLibrary

signature:
  dynamic controlled x : Integer
  dynamic controlled y : Integer

definitions:
  main rule swap =
    par
      x := y
      y := x
    endpar
```

Listing 1.4. Swap Example (*Asmeta*)

1, and Listing 1.3 at Line 3). Similar behavior can be achieved in *Asmeta* by setting a certain rule to a `main` rule (Listing 1.4 at Line 9) or in *AsmL* which forces the uses to define a `Main()` rule (Listing 1.2 at Line 8) which controls the computation directly. Moreover, it can be observed that the *swap* examples of CASM and *CoreASM* explicitly define the termination of the specification whereas the *swap* examples written in *AsmL* and *Asmeta* do not.

In order to implement e.g. an AST-based interpreter to execute this specifications a *language engineer* would have to implement a run-time kernel, which handles those implicit defined behaviors. Furthermore, if we think about optimizing such specifications, implicitly defined behaviors cannot be optimized and addressed by transformation passes in a generic way.

Generally speaking we have discovered two implicit behaviors – *initialization of functions* and *agent life cycle handling*. Latter is very important if we consider synchronous and asynchronous multi-agent ASM specifications. To express ASM specifications in a well-formed IR we present in the following sub-sections the definition of our CASM-IR model and its textual representation.

3.2 Types, Constants, and Functions

Due to the fact that every ASM-based language will eventually be executed by a real machine a term, expression or even a value will have a concrete type. Even Gurevich [2] suggested his ASM language definition lacks explicit typing, and it would be more practical to introduce such. Therefore, in the center of our

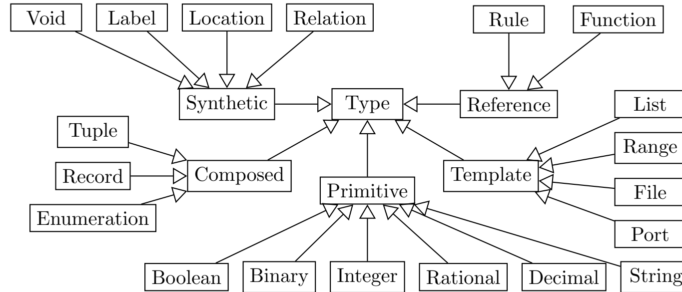

Fig. 3. CASM-IR Type System (Inheritance Tree)

CASM-IR model stands the type system with all of its possible type domains, which we will call from now on just types[4]. An overview is depicted in Figure 3. We can observe that the type system defines very basic (*Primitive*) types like *Boolean* or *Integer* up to very abstract ones (*Template*) like *List* or *File*.

Notable to mention here in contrast to other ASM languages is that CASM-IR always tries to be as close as possible to the mathematical foundation of a type. This means e.g. the *Integer* representation is represented as an arbitrary precise Integer with range $]-\infty, \infty[$. There is even the possibility – similar to the Ada programming language – to restrict the type to a certain sub-range. Furthermore, CASM-IR introduces a *Binary* type which can be used to represent any binary bit-precise value with defined bit-size. Along with this type CASM-IR features a set of *Binary* built-in[5] arithmetic operations. In the implementation of Lezuo et al. [12] this type was indirectly specified with Integer types by limiting built-in operations over a predefined bit-size values and the language itself was not aware of these operations. Another novel type in CASM-IR compared to other languages is that it features a *Reference* type. All references to rules, functions, and derived functions have to be typed to ensure type safety for indirect calls. Due to the mathematical foundation of ASMs, all typed CASM-IR constants[6] can have besides the type-specified (domain) content, an undefined value. Furthermore, we directly include in CASM-IR the notion of symbolic values that enable a clear definition of numeric as well as symbolic execution, whereas the symbolic values are its own domain value as suggested by Lezuo [6].

States are modeled through the function definitions[7]. As defined in [17] every ASM function has a name and an arbitrary type relation. By default every function – accordingly to the ASM definition – is undefined over its type relation domain and needs to be explicitly initialized in CASM-IR. Listing 1.5 on Page 8 depicts a constant `@c0` of type *Integer* and value 123, a constant `@c1` of type *Rule Reference* with relation $:\rightarrow Void$ and an *undefined* value, and a function `foo` with relation $: Boolean * Rational \rightarrow Integer$.

---

[4] for CASM-IR type specification, see: `http://casm-lang.org/ir/types`
[5] for CASM-IR built-in specification, see: `http://casm-lang.org/ir/builtins`
[6] for CASM-IR constant specification, see: `http://casm-lang.org/ir/constants`
[7] for CASM-IR function specification, see: `http://casm-lang.org/ir/functions`

```
1 ;; integer constant '123'
2 @c0 = i 123
3 ;; 'undefined' rule reference
4 ;;  constant of relation : -> Void
5 @c1 = r< -> v > undef
6 ;; function definition 'foo'
7 ;; with relation:
8 ;; Boolean * Rational -> Integer
9 @foo = < b * q -> i >
```

Listing 1.5. Constants and Functions

```
1 ;; enumeration type definition
2 bar = { A, B, C }
3 ;; setting agent type domain
4 ;; to enumeration type 'bar'
5 .agent = bar
6 ;; function definition 'program'
7 ;; with relation:
8 ;; bar -> RuleRef< -> Void >
9 @program = < bar -> r< -> v > >
```

Listing 1.6. Enum. and Agents

3.3 Agents, Rules, and Deriveds

ASM specifications can either be single or multi execution agent-based systems [1]. Therefore we provide a model instance to declare only the agent type domain that directly results in the desired behavior. For instance, if we would define the agent type domain to a *Boolean* type, we would define two operational agents. The agent type domain has an important role in the execution of all ASM specifications because starting from a defined agent rule the nested rules get called and so on. Furthermore, the defined agent domain is also used in a special internal *function* named `program` to store the current agent top-level rule as a rule reference. Listing 1.6 depicts how to set the model instance of the current agent type domain. In Line 2 an enumeration type `bar` gets defined with enumerators `A`, `B`, and `C`, and in Line 5 the agent type domain gets set to the type `bar`. Therefore we have specified in this example a multi-agent ASM with three agents. As already mentioned in Section 3.1 there is a special function named `program` that controls the execution of the agents in its kernel of ASM specifications which heavily depends on the agent type domain. Line 9 shows the corresponding `program` function definition with the agent type domain `bar`. The actual computation in ASMs is specified through transition rules. CASM-IR also has the notion of rules, but only for the *named rule* definitions. Other ASM rules like *Update*, *Conditional*, *Forall*, *Choose*, etc. are represented in CASM-IR through nested blocks and instructions (see Section 3.4).

Another important specification component in CASM-IR are *derived* functions or *deriveds* for short. It can be seen as a kind of typed macro to reuse *state-less* or *side-effect free* calculations. This means, that in *deriveds*, no state changes are allowed to be performed; ergo, no *Update* rules are allowed in derived function definitions.

3.4 Blocks, Instructions, and Registers

All basic expressions and state-modifying rules are represented in CASM-IR as *Instructions* in a Single Static Assignment (SSA) form. So produced results of instructions are stored in registers and the type is directly yielded from the specified instruction. This conceptual idea is borrowed from the Low Level Virtual Machine (LLVM) compiler IR design by Lattner and Adve [18]. So any instruction call can be specified by a resulting unique register name, an instruction name

```
1 %r0 = ;; ... calculation which yields result of type 'i'
2 %r1 = location < i -> i> @foo, i %r0 ;; yields type 'loc'
3 %r2 = lookup loc %r1 ;; yields type 'i'
4 %r3 = ;; ... calculation which yields result of type 'i' and uses '%r2'
5 update loc %r1, i %r3 ;; produces an update to function 'foo'
```

Listing 1.7. Location-, Lookup-, and Update-Instruction

and possible instruction operands with explicit types. This also indicates that the CASM-IR follows a register machine design and implementation approach.

Basic ASM rules like *skip*, *choose*, or the definition of execution semantics (*fork* and *merge*) are represented as single instructions. Novel in CASM-IR is that it explicitly models the reading (*lookup*) and writing (*update*) of ASM function states by dedicated instructions. This allows to analyze and optimize CASM-IR specifications as suggested by Lezuo et al. [12]. A *location* instruction performs the function location calculation. How the location is calculated is not fixed and has to be decided in the run-time implementation. E.g. a common technique would be the calculation of a function location by a certain hashing algorithm. The *lookup* instruction determines at a certain point in the specification, which state value is assigned to a certain function depending on the nested parallel and sequential execution semantics. The argument needed to perform a lookup is a location constant. An *update* instruction produces a new *location* and *value* pair, which gets applied to the surrounding (local) function state also known as *pseudo state* [12]. Therefore, an update instruction needs, besides the exact calculated function location, a value operand.

Listing 1.7 depicts an example usage of the *location*, *lookup*, and *update* instruction. In Line 2 a *location* calculation is performed for the function `foo` which has accordingly to the type one Integer argument. At Line 3 the actual *lookup* of the function value is performed. And in Line 5, a new *update* is performed to the same location were the lookup was performed. Similar to traditional assembler languages, the CASM-IR includes a *call instruction* as well, but this call instruction is used for multiple invocation types. It is used to call specified rules, derived functions, and pre-defined built-ins either directly by its name or indirectly through a register value of a reference type. Besides the generic *call instruction* there exist several instructions to perform intermediate calculations of arithmetic, logical, and comparison operations[8].

Multiple instructions are compound to a Statement Block (SB) whereas the execution semantics of the instructions is always sequential. Several blocks are grouped together and form an Execution Semantics Block (ESB) which can either have a *parallel* or *sequential* execution semantics. Additionally, every ESB contains, besides the sub-blocks, an *entry* and an *exit* SB, in which the actual execution semantics is specified by appropriate *fork* and *merge* instructions. Figure 4 on Page 10 depicts the composition of rules, the ESB and SB blocks as well as instructions.

[8] for CASM-IR instr. specification, see: `http://casm-lang.org/ir/instructions`

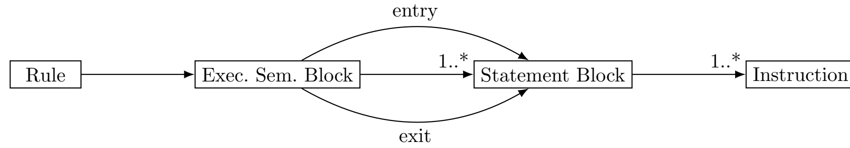

Fig. 4. CASM-IR Rules, Blocks, and Instructions

```
1  CASM-IR                             ;; CASM-IR specification header
2  a = { $ }                           ;; definition of enum. type 'a'
3  .agent = a                          ;; set agent type domain to type 'a'
4  @swap < -> v>                       ;; declaration of rule 'swap'
5  @c0 = r< -> v> @swap                ;; 'swap' rule reference
6  @c1 = a $                           ;; agent constant of single agent
7  @c2 = r< -> v> undef                ;; undefined rule reference
8  @c3 = a $                           ;; agent constant of single agent
9  @program = <a -> r< -> v>>          ;; 'program' function definition
10 @x = < -> i>                        ;; definition of function 'x'
11 @y = < -> i>                        ;; definition of function 'y'
12 @init -> v = {                      ;; definition of 'init' rule
13   lbl0: entry                       ;; ESB entry block of lbl0
14     fork par                        ;; fork instruction parallel
15
16   lbl1: %lbl0                       ;; SB lbl1 in ESB lbl0
17     %r0 = location <a -> r< -> v>> @program, a @c1
18     update loc %r0, r< -> v> @c0
19
20   exit: %lbl0                       ;; ESB exit block of lbl0
21     merge par                       ;; merge instruction parallel
22 }
23 @swap -> v = {                      ;; definition of 'swap' rule
24   lbl2: entry                       ;; ESB entry block of lbl2
25     fork par                        ;; fork instruction parallel
26
27   lbl3: %lbl2                       ;; SB lbl3 in ESB lbl2
28     %r1 = location < -> i> @y
29     %r2 = lookup loc %r1            ;; lookup of function 'y'
30     %r3 = location < -> i> @x
31     update loc %r3, i %r2           ;; update of function 'x'
32
33   lbl4: %lbl2                       ;; SB lbl4 in ESB lbl2
34     %r4 = location < -> i> @x
35     %r5 = lookup loc %r4            ;; lookup of function 'x'
36     %r6 = location < -> i> @y
37     update loc %r6, i %r5           ;; update of function 'y'
38
39   lbl5: %lbl2
40     %r7 = location <a -> r< -> v>> @program, a @c3
41     update loc %r7, r< -> v> @c2
42
43   exit: %lbl2                       ;; ESB exit block of lbl2
44     merge par                       ;; merge instruction parallel
45 }
```

Listing 1.8. Swap Example (*CASM-IR*)

3.5 Motivating Swap Example in CASM-IR

In this section we present an example output of the transformed motivating example `swap` CASM specification from Listing 1.1 to our CASM-IR. The performed model-to-model transformation is implemented in the CASM front-end

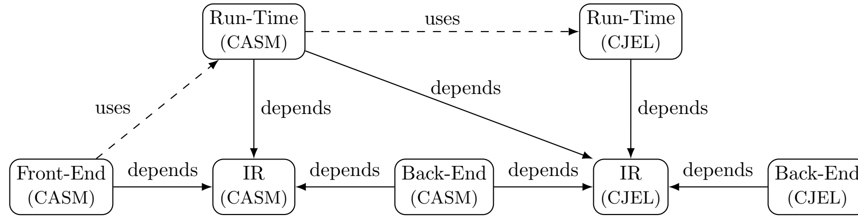

Fig. 5. CASM System Implemenation (Library Dependency Graph)

(see Section 4). It shall summarize several of the presented concepts and sketch some optimization possibilities, which can be obtained through the representation of ASM specifications in the CASM-IR. Note that this presented transformed motivated example is valid for the other presented ASM `swap` specifications as well (Listing 1.2, Listing 1.3, and Listing 1.4).

Listing 1.8 on Page 10 visualizes a CASM-IR instance, where the missing definitions and implicit behaviors from Listing 1.1 are explicitly specified. In the transformed specification we can observe that first of all the agent type domain gets set to a enumeration type named `a` (Line 3) with the name `$` (Line 2). This means that the agent type domain consists of only one concrete value and hence we have a single execution agent ASM specification. Thereafter, a forward declaration of the rule `swap` is specified (Line 4) because the next listed constants (Line 5-8) contain the symbol of the `swap` rule to define a rule reference constant. Next, three functions are defined. The `program` function (Line 9) with the previous defined agent type domain that stores the ASM agent top-level rule reference. After that the functions `x` (Line 10) and `y` (Line 11) are defined accordingly to the originally input specification. Before the definition of the `swap` rule gets defined, the initialization of the ASM state has to be specified, which at least has to set the correct starting rule of the agents. Note that all function states in ASMs are by default undefined. Last but not least the rule `swap` gets defined. It contains a parallel execution semantics block with three trivial statements and several location, lookup and update instructions.

Regarding the optimization potential in this revised example we can detect several possible ASM-related optimizations. The most obvious one would be a hoisting optimization of redundant location calculations, because the location of nullary functions will always be the same. The calculation e.g. of the location of function `y` at register `%r1` (Line 28) could be moved up before the fork instruction of the entry section at `lbl2` (Line 24). And the location calculation of function `y` at the register `%r6` (Line 36) can be removed and all its uses can be replaced by `%r1`. The same applies for the location of function `x` and register `%r3` and `%r4` (Line 30, Line 34).

4 Implementation and Integration

Figure 5 depicts the CASM system implementation libraries visualized as a library dependency graph. The CASM run-time and back-end libraries are based

on corresponding CASM unaware Just-in-time Emitting Language (CJEL) libraries (situated one layer below the CASM libraries). The CJEL layer is not described in this paper. All libraries are implemented in C++11/14 standard.

The implementation of the CASM-IR model consists of two major base classes - *Type* and *Value*. The type system and type hierarchy is implemented according to the definition presented in Section 3.2. All other model instances are sub-classes of the *Value* class. This design approach was borrowed again from the LLVM compiler project where *everything is a value* [18]. Furthermore, every value has a type. The CASM-IR implementation provides a rich Application Programming Interface (API) to provide certain information to front-end implementations. To be more precise here, for every instruction and built-in, it is possible to fetch all defined type relations through an internal type map structure. This enables a clean separation between a front-end language definition and the IR internals.

Based on the CASM-IR, we have designed our CASM language front-end. Compared to the CASM language implementations from Lezuo et al. [12] the AST has resulted in a much simpler and clearer design then before, because all type, operator, and built-in design decisions were already made in the CASM-IR implementation. Therefore the AST only focuses on the input language itself. CASM is a statically strong inferred typed language. Hence, the difference between the front-end CASM input specification language and the CASM-IR model is that the front-end language requires a symbol resolver, type checker and type inference pass to fully type the parsed input specification AST representation. In the analyzer passes we use the provided API of the CASM-IR to query and check if certain types, built-ins, and operators exist. Furthermore, during type inference, the front-end can infer the correct type through the pre-defined type relations of the specified CASM-IR operators. E.g. if a type is not possible to be inferred in the front-end, the possible types can be retrieved from the CASM-IR and used as helpful debugging information for language users.

Besides type inference and other analyzes done by the front-end implementation, the most important benefit of targeting the CASM-IR is that a language front-end engineer can directly call evaluation instrumentation functions of the CASM-IR to perform calculations of operator instructions and built-ins.

5 Related Work

One of the best-known ASM implementations is the *Asmeta*[9] tool-set with the *AsmetaL* language [16]. The core of *Asmeta* is designed and implemented using the Eclipse Modeling Framework (EMF) *Ecore* meta-model[10]. Based on the *Ecore* meta-model, the ASM language model of *Asmeta* is directly described as an instance (model). Therefore, the execution and precise calculation of the implemented ASM simulator is bound to the run-time implementation of the *Ecore* meta-model and its EMFs Java interface realizations.

[9] for Asmeta project, see: `http://asmeta.sourceforge.net`
[10] for EMF project, see: `http://eclipse.org/modeling/emf`

Another notable ASM design and implementation is *CoreASM*[11] originally developed by Farahbod et al. [15]. The focus of CoreASM is to provide a flexible and extensible ASM implementation and to be as near as possible to the described ASM method by Börger [17]. CoreASM is implemented in Java and its IR and run-time is directly bound to the Java Virtual Machine (JVM). Microsoft research designed and implemented an ASM language named *AsmL*[12] [14]. AsmL is implemented and based to the `.NET` framework.

Besides CASM-IR, which solves a uniform ASM intermediate representation to be language front-end independent, Arcaini et al. [19] proposed a Unified Abstract State Machines (UASM) language syntax. Their approach is to unify the front-end ASM syntax representation and this is in the perspective of CASM-IR yet another ASM front-end input specification. Similar to the ASM language proposed by Anlauff [20], the eXtensible ASM (XASM) language[13], which compiles XASM specifications to C.

Lezuo et al. [21] introduced in 2013 the CASM language. The origin of this language was that all the (publicly available) existing ASM tools were impracticable for industrial sized applications [22]. The tool-chain presented by Lezuo et al. [12] focuses like the other ASM designs only on the input specification itself, thus those research results were not directly usable by other ASM-based language frameworks. The latter motivated, as already stated in Section 2, to rethink the proposed ASM language engineering designs and implementations, leading to our model-based transformation approach [13] for the CASM language[14].

Different representation and transformation approaches have been investigated in the *AsmGofer* language by Schmid [23], which is based on the programming language Gofer (similar to Haskell), and the *ASM Workbench* with the *ASM-SL* language introduced by Del Castillo [24], which is implemented in Standard ML. The *ASM-SL* has been explored further by Schmid [25] to represent and encode specifications in C++. The translation (compilation) scheme was limited to a *double buffering* concept and therefore unable to encode mixing sequential and parallel rules. CASM-IR solves this by using block-level nested *fork* and *merge* instructions to control the update-set behavior.

Another transformation scheme for ASMs was presented by Bonfanti et al. [26] to represent and encode *AsmetaL* specifications in C++ code targeting Arduino platforms. Their code generator directly converts the ASM specification to the desired target language and run-time environment. By targeting a different target run-time environment, platform, or architecture the encoded and implement ASM behavior would have to be re-implemented in every code generator.

Important to point out is that CASM-IR tries to establish a mid-end IR for ASM-based languages similar to the approach for classical programming language IR models such as GCCs *GENERIC* and *GIMPLE* by Merrill [27] or the LLVM IR by Lattner and Adve [18].

[11] for CoreASM open-source project, see: `http://github.com/coreasm`
[12] for AsmL documentation and project, see: `http://asml.codeplex.com`
[13] for XASM documentation, see `http://sourceforge.net/projects/xasm`
[14] for CASM open-source project, see: `http://github.com/casm-lang`

6 Conclusion

We have presented in this paper CASM-IR, a statically and strongly typed, well-formed ASM-based IR, to provide the ability for ASM-based language engineers to specify the internals of their ASM language in a well-defined representation model. Besides the type system, agent, functions, deriveds, rules, blocks, and instruction semantics, we discussed ASM properties, which are indirectly represented in ASM source languages and made explicitly and typed in the CASM-IR. There are several other issues regarding implicit behavior in ASM-based high-level languages we could point out, but it would go beyond of the scope of this paper. We have given a short overview of our implementation, corresponding libraries, and discussed the usefulness of our approach.

Regarding the CASM-IR itself, there is a lot of future work in the direction of the type system. The providing of types like *trees*, *sets*, *bags*, and so on, is still an open topic. We are already working on the implementation, formal definition and verification of ASM-related optimization transformations based on the gained knowledge from Lezuo et al. [12] for our CASM-IR. Another research direction we are working on is the byte-code representation of the CASM-IR. This would allow the implementation of very compact virtual machines for ASM-based specifications.

References

[1] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods," pp. 9–36, New York, NY, USA: Oxford University Press, Inc., 1995.

[2] Y. Gurevich, "Sequential Abstract-State Machines Capture Sequential Algorithms," *ACM Transactions on Computational Logic (TOCL)*, vol. 1, no. 1, pp. 77–111, 2000.

[3] E. Börger, "The Origins and the Development of the ASM Method for High Level System Design and Analysis," *Journal of Universal Computer Science*, vol. 8, no. 1, pp. 2–74, 2002.

[4] R. F. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer Berlin Heidelberg, 2001.

[5] H. Sasaki, "A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '99, (New York, NY, USA), ACM, 1999.

[6] R. Lezuo, *Scalable Translation Validation; Tools, Techniques and Framework.* PhD thesis, 2014. Wien, Techn. Univ., Diss.

[7] M. Barnett and W. Schulte, "Spying on Components: A Runtime Verification Technique," in *Proceedings of the Workshop on Specification and Verification of Component-Based Systems*, SAVCBS'01, pp. 7–13, 2001.

[8] U. Glässer and M. Veanes, "Universal Plug and Play Machine Models," in *Design and Analysis of Distributed Embedded Systems*, pp. 21–30, Springer, 2002.

[9] J. K. Huggins and D. V. Campenhout, "Specification and verification of pipelining in the arm2 risc microprocessor," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 3, no. 4, pp. 563–580, 1998.

[10] A. G. Kleppe, "Software Language Engineering: Creating Domain-Specific Languages using Metamodels," *Addisson-Wesley, 2009.*

[11] M. Völter, *Generic Tools, Specific Languages.* PhD thesis, Delft University of Technology, June 2014.

[12] 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.

[13] P. Paulweber and U. Zdun, "A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, pp. 250–255, Springer, 2016.

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

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

[16] A. Gargantini, E. Riccobene, and P. Scandurra, "A metamodel-based language and a simulation engine for abstract state machines," *Journal of Universal Computer Science*, vol. 14, no. 12, pp. 1949–1983, 2008.

[17] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Science & Business Media, 2003.

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

[19] P. Arcaini, S. Bonfanti, M. Dausend, A. Gargantini, A. Mashkoor, A. Raschke, E. Riccobene, P. Scandurra, and M. Stegmaier, "Unified Syntax for Abstract State Machines," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, pp. 231–236, Springer, 2016.

[20] M. Anlauff, "XASM – An Extensible, Component-based Abstract State Machines Language," in *Abstract State Machines-Theory and Applications*, pp. 69–90, Springer, 2000.

[21] R. Lezuo and A. Krall, "Using the CASM Language for Simulator Synthesis and Model Verification," in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, p. 6, ACM, 2013.

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

[23] J. Schmid, "Introduction to AsmGofer," http://www.tydo.de/AsmGofer, 2001.

[24] G. Del Castillo, "The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models," in *Tools and Algorithms for the Construction and Analysis of Systems - 7th International Conference, TACAS 2001, Proceedings*, pp. 578–581, Springer Berlin Heidelberg, 2001.

[25] J. Schmid, "Compiling Abstract State Machines to C++," *Journal of Universal Computer Science*, vol. 7, no. 11, pp. 1068–1087, 2001.

[26] S. Bonfanti, M. Carissoni, A. Gargantini, and A. Mashkoor, "Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino," in *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pp. 295–301, Springer International Publishing, 2017.

[27] J. Merrill, "GENERIC and GIMPLE: A New Tree Representation for Entire Functions," in *Proceedings of the 2003 GCC Developers' Summit*, pp. 171–179, 2003.