Unified Syntax for Abstract State Machines*

Paolo Arcaini[2], Silvia Bonfanti[3], Marcel Dausend[1], Angelo Gargantini[3], Atif Mashkoor[4], Alexander Raschke[1], Elvinia Riccobene[5], Patrizia Scandurra[3], and Michael Stegmaier[1]

[1] Ulm University, Germany, {marcel.dausend, alexander.raschke, michael-1.stegmaier}@uni-ulm.de
[2] Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic, arcaini@d3s.mff.cuni.cz
[3] Università degli Studi di Bergamo, Italy, {silvia.bonfanti, angelo.gargantini, patrizia.scandurra}@unibg.it
[4] Software Competence Center Hagenberg, Austria, atif.mashkoor@scch.at
[5] Università degli Studi di Milano, Italy, elvinia.riccobene@unimi.it

Abstract. The paper presents our efforts in defining UASM, a unified syntax for Abstract State Machines (ASMs), based on the syntaxes of two of the main ASM frameworks, CoreASM and ASMETA, which have been adapted to accept UASM as input syntax of all their validation and verification tools.

1 Introduction and goals of the project

Abstract State Machines (ASMs) are a flexible, yet mathematically well-founded method and language for rigorous system engineering. The formalism can be seen as "pseudocode over abstract data" [2]. Although this pseudocode notation is formally defined, in practice many ways exist to encode algebraic concepts and many abbreviations can be used to improve model conciseness and readability.

Among the different frameworks for the ASM method (like AsmL, ASM Workbench, ASMGofer, KIV), two of the main ones are ASMETA [1] and CoreASM [3]. These platforms provide industrial strength tools to specify, verify, simulate, and test ASM models. However, they implement different dialects of the pseudocode notation and support slightly different extensions of the original definition. For example, AsmetaL (the textual notation of the ASMETA framework [4]) provides the concept of module that is not present in CoreASM, while CoreASM allows the use of abstract rules, a feature that is not present in AsmetaL. There are some differences on the way they represent signature: CoreASM is not typed, and so it permits an agile modeling style, while AsmetaL is strongly typed. Moreover, the two syntaxes are sometimes slightly different in representing the same concept: for example, the keyword for a sequential

* The research reported in this paper has been partly supported by the Charles University research funds PRVOUK, and by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

block is `seq ...endseq` in AsmetaL and `seqblock ...endseqblock` in CoreASM.

Therefore, while the availability of multiple support platforms is obviously an advantage, it may also be confusing for new adopters of the method. Moreover, designers cannot share models among the tools (unless a translator or adapter is defined) and thus can not easily take advantage of each tool's strengths.

To overcome these limitations, the idea of a common syntax definition "Unified ASM" (UASM), driven by the community, open to any actors, has grown in the last two years. This paper presents the activities performed so far. The challenges of this project are both to preserve various useful extensions of the different tools and support a variety of application scenarios. On the one hand, the UASM language should be usable for communication with customers and non-experts, and, on the other hand, precise enough to allow automatic analysis (like type checking, property verification, etc.). Moreover, we will try to identify unifying solutions for those ASM aspects for which the two frameworks made different design decisions. Two examples are the syntax and semantics for state initialization and the definition of basic data types.

2 Insights into the UASM grammar

As mentioned in the previous section, the applications of ASMs are manifold. Due to this, we decided to keep the new common grammar as flexible as possible. We tried to include as many useful constructs from the contributing languages as possible, but naturally, some design decisions had to be made.

In order to allow for a more legible specification, UASM offers textual notations for basic constructs. Instead of keywords for mathematical constructs, we also allow Unicode characters (e. g., $\in$ instead of `in`, $\forall$ instead of `forall`).

UASM does not require type annotations. If no type information is given, the types are checked dynamically during runtime (e. g., $*$ (multiplication) can only be applied on two numbers). Whenever type information is required at a later date, it can be added on demand. If a type information is given, the type correctness is checked when the specification is parsed. Currently, only a few basic (boolean, numbers, chars, strings) and set-based types (set, list, bag, map) are defined. For the future, we plan to integrate a notation for algebraic data types in order to allow for arbitrary complex (recursive) data types.

The module concept of ASMETA was adopted to allow for a better modularization of large specifications.

UASM also provides definition for some aspects that have been left open up to now, e. g., the new keyword `exec` followed by a rule name defines the rule to be executed by the initial agent. Usually, this rule introduces new agents and their programs and initializes the abstract state. Alternatively, initial values for all locations or only specific locations of a function can be described as part of its definition (see example below). The defined constructs allow in CoreASM as well in ASMETA to write and execute multi-agent specifications. Despite of

that, it might be useful in the future to define special constructs for creating, removing, or assigning programs to agents.

In the following, we introduce an excerpt of the UASM language definition[1]. We focus on the definition of functions and their initialization. First, this part is a substantial supplement to the existing syntax and semantic definition of the underlying literature; second, this part reflects some design decisions originating from different existing realizations of ASM languages.

The aforementioned decision that types are optional strongly influences the UASM language. This is reflected, for example, in the definition of function parameters that allows identifiers as well as domains or a combination of both.

```
ParamameterDef ::= '(' (Id 'in' Domain | Id | Domain)
   (',' (Id 'in' Domain | Id | Domain))* ')'
```

For the initialization of a function, we support a fixed value for all its locations, or specific location values by using maps and terms. The following example illustrates these different concepts by the initialization of the controlled function *gateStatus* of a rail road crossing that is defined according to the following definition.

```
ControlledFunction ::= ( .. | 'controlled' 'function') IdFunction
   ParamameterDef? ('->' Domain)? ('initially' 'from'? Term )?
```

controlled function *gateStatus*(gate **in** GATES) **initially from**
(1) {gate1 → ***open***, gate2 → ***closed***}
(2) **if** *isTrainApproaching*(gate) **then** ***closed*** **else** ***open***
where *isTrainApproaching*(g) = ∃ s **in** SENSORS **with**
g ∈ *observedGates*(s) **and** *trainOnTrack*(s)

The above example (1) illustrates how maps can be used to assign different initial values to specific locations, i. e., gate status for specific gates is different. A more flexible approach is the dynamic initialization of the state based on derived functions. In this case, the initialization is done lazily, i. e., before a function is accessed (read), it is checked whether this particular location has been previously initialized or updated. If not, the given derived function is evaluated returning the initial value. Under the assumption that a railroad crossing control module should take control at a random time, i. e., under different circumstances, we use this dynamic initialization for the status of the rail road gates (2). The initial values of *gateStatus* are computed on demand by the derived function *isTrainApproaching*, whose result is based on current sensor data.

UASM allows the declaration and definition of static functions. For instance, a function `sum` that takes two integers and returns the sum, can be defined as:

static function *sum*(a **in** INTEGER, b **in** INTEGER) **always** a + b

[1] The syntax of our language definition is conform to the W3C EBNF notation.

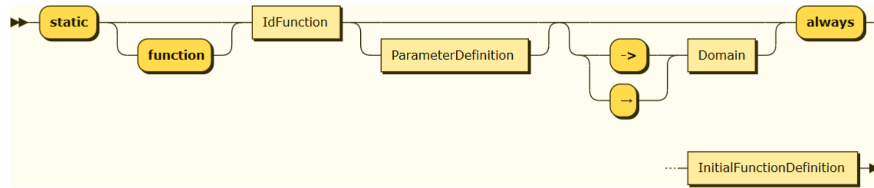

Fig. 1. Static Function definition

Several tools can be used to build a visualization of the grammar rules by means of syntax diagrams (or railroad diagrams)[2]. For instance, the definition of static functions is shown in Fig. 1.

3 Re-engineering existing tools

In order to allow the CoreASM and the ASMETA frameworks to accept the new common ASM syntax, we had to do some re-engineering, as described as follows.

Reference parser and integration into CoreASM CoreASM is an open-source project defining an ASM language implementing tools that focus on high-level design and experimental validation of ASM specifications.

The CoreASM tool architecture defines a highly modular system based on a minimal kernel. This architecture enables to seamlessly integrate additional language constructs as well as tools and yields in manifold extensions and domain specific applications.

CoreASM's major strengths are in the creation, refinement, and debugging of specifications. For example, starting with an abstract and untyped or only partially typed specification that can already be executed, refining this specification, and performing comprehensive debugging and testing.

As our goal is to provide an easy to read and understandable definition of the UASM syntax. The current grammar definition is not optimized for automatic processing, yet. It contains ambiguities resulting from i. e., optional end-constructs and operator precedences which are not reflected by the grammar. Hence, we are going to derive a grammar definition for UASM that facilitates automatic processing like using parser generators.

Other than usual bottom-up or top-down parsers, JParsec-Framework can deal with our grammar definition as it can handle left recursion and it resolves ambiguities by applying strategies that make parsing deterministic. Therefore, we implemented a reference parser for UASM using the JParsec-Framework by merely transcribing our grammar into the JParsec syntax. This parser is already publicly available[3]. Because JParsec is a parser combinator, the reference implementation can be easily extended. We also integrated it into CoreASM

[2] We use the web service `http://www.bottlecaps.de/rr/ui`
[3] `https://github.com/uasm/uasm-reference-parser`

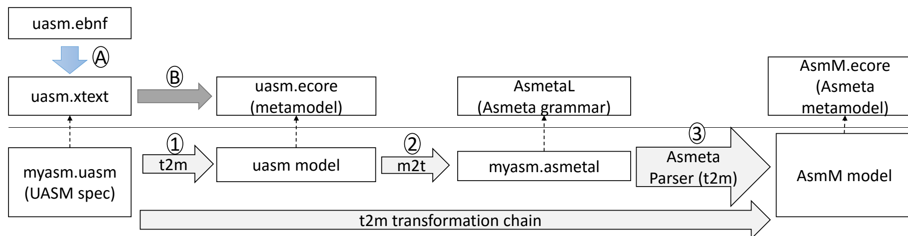

Fig. 2. UASM and ASMETA integration

without any limitations to the application of existing tools, e. g., the interpreter and the debugger.

Integration into ASMETA The ASMETA framework [1] is based on the *ASM Metamodel* (`AsmM`) [4], an abstract syntax description (defined with the Eclipse Modeling Framework (EMF)) of a language for ASMs. From AsmM, a concrete textual syntax (`AsmetaL`), a parser, Java APIs, etc., have been developed for model editing, storage, and manipulation. On the top of these, more complex tools for validation, verification, and testing have been developed. They all manipulate AsmM models (i.e., instances of AsmM). So, in order to use ASMETA tools on UASM specifications, we must map UASM specifications to AsmM models.

We followed the approach shown in Fig. 2. We derived an Xtext grammar starting from the UASM EBNF grammar (step A); from the Xtext grammar, a metamodel and a parser have been obtained automatically (step B). Transforming a UASM specification to an AsmM model (bottom of Fig. 2) consists in parsing the specification with the Xtext parser (step 1), producing an AsmetaL specification from the ecore objects produced by Xtext (step 2), and finally obtaining an AsmM model with the AsmetaL parser (step 3).

4 Validation of the approach

After the integration of UASM in the existing frameworks (see Sect. 3), we have devised two validation activities that one has to apply to check whether a tool correctly supports the new syntax. These activities will be initially applied to CoreASM and ASMETA, but, in the future, to any tool willing to support UASM (to get a sort of UASM compliance certification).

As a first validation activity, we plan to create a repository of syntactically correct and non-correct UASM specifications. They should be representative of the kind of models that can be written in UASM, i.e., they should cover all grammar elements. We will then check that a UASM compliant tool correctly accepts/rejects the specifications.

As a second validation activity, we want to check that the semantics is the same in any framework. We will establish a way for accepting a sequence

of inputs and saving the machine behavior as sequences of update sets (and outputs). Any UASM tool must be able to produce the behavioral sequences in that format. We will have a way to compare if two behaviors are identical. We will save, together with the benchmarks, also their expected behaviors and we will then check if CoreASM and ASMETA (or any other tool) correctly capture the intended semantics. This approach can only validate deterministic single-agent ASMs that, given an input sequence, produce only one possible output sequence. As future work, we plan to devise ways to validate our tools also using nondeterministic and/or multi-agents ASMs. In that case, we could record the output in terms of trees representing all possible evolutions of the system. However, the approach could not scale or even be not applicable in case of infinite-state models. A different approach could be to simulate the model with a framework (either CoreASM or ASMETA) and, step by step, check whether the produced update set is allowed also in the other framework (in a kind of runtime monitoring). This approach would have the advantage of being scalable also to infinite-state systems, although it could miss some faults.

5 Conclusion and Future Work

We have presented our efforts in defining UASM, a unified syntax for ASMs, based on the syntaxes of the main two ASM frameworks, CoreASM and ASMETA, which have been adapted to accept UASM as input syntax of all their validation and verification tools.

As further future work, to check that the two frameworks interpret the UASM models in the same way, we plan to apply some validation activities based on comparison of simulation traces. Moreover, we also plan to extend UASM with constructs not part of the two starting syntaxes, but that are part of other ASM syntaxes (e.g., classes of AsmL).

References

1. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience 41, 155–166 (2011)
2. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
3. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. Fundamenta Informaticae 77(1-2), 71–104 (2007)
4. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for Abstract State Machines. J. UCS 14(12), 1949–1983 (2008)