What is the Natural Abstraction Level of an Algorithm?

Andreas Prinz(✉)

Department of ICT, University of Agder, Grimstad, Norway
Andreas.Prinz@UIA.no

Abstract. Abstract State Machines work with algorithms on the natural abstraction level. In this paper, we discuss the notion of the natural abstraction level of an algorithm and how ASM manage to capture this abstraction level. We will look into three areas of algorithms: the algorithm execution, the algorithm description, and the algorithm semantics. We conclude that ASM capture the natural abstraction level of the algorithm execution, but not necessarily of the algorithm description. ASM do also capture the natural abstraction level of execution semantics.

Keywords: Abstract state machine · Abstraction · Execution · Description · Semantics

1 Introduction

Abstract state machines (ASM) [9], originally called Evolving algebras, [22] enable a high-level and abstract description of computations. ASM can be considered formalized pseudo-code, such that ASM programs are readable even without much introduction. The original purpose of ASM was to improve on the low-level abstraction provided by Turing machines [43], in order to be able to reason better about computability. This original purpose was achieved with the sequential ASM thesis [23]. It was later extended with an ASM thesis for parallel [3,5,40] and distributed computations [13].

From there, ASM were developed into different directions. Egon Börger understood very early that ASM are not only a mathematical tool for computability, but also a tool for system design and analysis. For the practical applicability, several more features were needed for ASM beyond [23], for example time [34] and distributed computations [38].

Another major ingredient for system design is a method to design systems, in this case the ground model approach [10]. This approach enables step-wise systems design, keeping correctness all the way to the final system.

The theoretical track of ASM has achieved a lot of success, and even though there are still details to be sorted out [36,38], this work is well under way. The major difference between the Turing machines approach and ASM is that ASM promise to work on the natural abstraction level for the computation.

A. Raschke et al. (Eds.): Börger Festschrift, LNCS 12750, pp. 199–214, 2021.
https://doi.org/10.1007/978-3-030-76020-5_12

It is of essence for every engineer to work on the right level of abstraction, as problem descriptions are simpler, and solutions are understandable at the right level of abstraction. Sometimes, solutions introduce high level of complexity by being at the wrong level of abstraction [15].

This paper tries to review the concept of abstraction level and identifies the meaning of natural abstraction in three dimensions: algorithm execution, algorithm description, and algorithm (language) semantics. We focus on sequential algorithms, although the conclusions also apply to other kinds of algorithms.

This paper starts with a discussion of the concept of algorithm in Sect. 2 before introducing abstract state machines in Sect. 3. Then we look into abstraction levels in executions in Sect. 4, in descriptions in Sect. 5, and in language semantics in Sect. 6. We conclude in Sect. 7.

2 What is an Algorithm?

Before looking into abstraction levels, we need to agree what an algorithm is. Harold Stone proposes the following definition "...any sequence of instructions that can be obeyed by a robot, is called an algorithm" (p. 4) [42]. Boolos et al. offers a similar definition in [7]: "... explicit instructions such that they could be followed by a computing machine".

This definition includes computer programs, bureaucratic procedures and cook-book recipes. Often, the condition that the algorithm stops eventually is included. In our context, also infinite loops are permitted because we also want to include server programs. Besides, termination is undecidable. Please note that the notion of algorithm relies on a basic set of elementary operations or functions.

Turing machines formalize this informal definition. Gurevich writes in [23]: "... Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine ... according to Savage [1987], an algorithm is a computational process defined by a Turing machines".

From the considerations so far, we conclude that an algorithm has a *description* ("a sequence of instructions") and an *execution* ("a computational process"). It is the *semantics* of the description that leads to the execution.

As an example, let's look at the Euclidean algorithm which computes the greatest common divisor *gcd* from two natural numbers n_1 and n_2. It can be expressed in ASM as follows.

```
IF n1 > n2 THEN
    DO IN-PARALLEL
        n1 := n2
        n2 := n1
    ENDDO
ELSEIF n1 = 0 THEN
    gcd := n2
ELSE
    n2 := n2 − n1
ENDIF
```

2.1 Algorithm Execution

For the execution of the Euclidean algorithm we have to know that the parallel execution of assignments is the standard mode in ASM [9]. The ASM code for the Euclidean algorithm describes one step of the algorithm, and it is repeated until there are no more changes.

A sample execution of the ASM algorithm for the numbers $n_1 = 1071$ and $n_2 = 462$ leads to the following sequence of pairs (n_1, n_2): $(1071, 462)$, $(462, 1071)$, $(462, 609)$, $(462, 147)$, $(147, 462)$, $(147, 315)$, $(147, 168)$, $(147, 21)$, $(21, 147)$, $(21, 126)$, $(21, 105)$, $(21, 84)$, $(21, 63)$, $(21, 42)$, $(21, 21)$, $(21, 0)$, $(0, 21)$. The result is then $gcd = 21$. Please note that gcd would be present in all states.

We will look at the execution of algorithms in Sect. 4.

2.2 Algorithm Description

Algorithms can be expressed in many kinds of notation, including natural languages, pseudo code, flowcharts, programming languages or control tables.

When we use Java to express the same algorithm, it looks something like that. Please note the extra temporary variable t for swapping $n1$ and $n2$. In addition, there is an enclosing `while` loop which is not needed in ASM. Java does not provide natural numbers as types; we silently assume that the parameters are non-negative.

```
public static int gcd(int n1, int n2) {
    while (n1 > 0) {
        if (n1 > n2) {
            int t = n1;
            n1 = n2;
            n2 = t;
        } else {
            n2 = n2 − n1;
        }
    }
    return n2;
}
```

Now we look at a version of the algorithm in Lisp. The solution is recursive as is customary in Lisp. Again, we assume that the typing of the parameters is correct.

```
(defun gcd (n1 n2)
    (if (= n1 0)
        n2
    (if (> n1 n2)
        (gcd n2 n1)
        (gcd n1 (- n2 n1))
    )
)
```

Finally, we can also use Prolog for the algorithm as follows. Here, we need to use an extra parameter for the result. This solution is again recursive and the typing is assumed to be correct.

```
gcd(0, N2, Result):- !, N2=Result.
gcd(N1, N2, Result):- N1 > N2, gcd(N2, N1, Result).
gcd(N1, N2, Result):- N1 =< N2, N2New is N2-N1, gcd(N1, N2New, Result).
```

We will look at the description (languages) of algorithms in Sect. 5.

2.3 Algorithm Semantics

All possible executions of an algorithm are the semantics of the algorithm. This means that the semantics of the algorithm connects the description of the algorithm with the execution of the algorithm. More precisely, the description is written in a language, and the semantics of the language provides the execution(s), see also [30] and [19].

This way, the semantics of the algorithm description is implied by the semantics of the language which is used for the description. There is not a semantics for each and every description, but a general semantics for the language of the descriptions. Therefore, algorithm semantics is in fact language semantics.

We will look at semantics in Sect. 6.

3 Abstract State Machines

The central concepts in abstract state machines (ASMs) are (abstract) states with locations and transition rules with updates. Their definitions can be found in many sources, including, but not limited to [2,4,6,8,9,12].

An abstract state machine (ASM) program defines abstract states and a transition rule that specifies how the ASM transitions through its states. ASM states are defined using a *signature* of names (function symbols), where each name has an arity. This allows to construct expressions using the names in the usual way. ASM names can be typed in the usual way.

The states are then interpretations of the names over a base set of values. Each name with arity zero is interpreted as a single element of the base set, while each name with arity n is interpreted as an n-ary function. Expressions are interpreted recursively.

ASM names are classified into *static* names whose interpretation does not change (e.g. True), and *dynamic* names which are subject to updates. Each ASM signature includes the predefined static names *True*, *False* and *Undefined*, interpreted as three distinct values. All ASM functions are total, and the special value *Undefined* is used to model partial functions.

An ASM transition rule (program) looks like pseudo-code and can be read without further explanation. The rules include assignments, if, forall, and some other statements. We refer to [9] for a formal definition.

The basic unit of change is an assignment, written as $loc := e$. Executing this assignment means to change the interpretation of the location loc to the value of the expression e in the given state.

Locations ($loc = f(e_1, \ldots, e_n)$) are constructed of an n-ary name (f) and n expressions e_i. More precisely, a unary function symbol u is a location, and any function symbol f with a number of locations l_i as arguments of f is a *location* as well. In each state, each location has a value.

An *update* is given by two locations, one on the right-hand side and one on the left-hand-side. The value of the left-hand side location is replaced by the value of the right-hand side location, such that $lhs = rhs$ will be true in the new state, unless the value of rhs is also changed in the state change. Formally, the assignment creates an update, which is a pair of a location and a value. All the applicable updates are collected into an update set, thereby implementing the parallel execution mode. Applying the update set to the current state (executing it) leads to the changes in the next state.

An ASM run starts with an *initial state*, being a state as defined above. For each state, the transition rule produces an update set which leads to the next state, thereby creating a sequence of states. Each state change (or step or move) *updates* the interpretation of its symbols given by a set of updates according to the assignments used.

4 Executing Algorithms

For the execution of algorithms, we need to look at the runtime, which is basically the same as operational semantics [18,26,35]. Runtime has two aspects, namely runtime structure including a set of initial states and runtime changes (steps) [37,39]. These same aspects are also identified in the sequential time postulate in [23], which postulates the existence of a set of states including initial states, and a one-step transformation function between states. We look into states and steps in the sequel.

4.1 Runtime Structure (States)

There is agreement between the theoretical [23][1] and the practical [39] understanding of runtime states as follows.

- States have a structure (States are first-order structures).
- The possible runtime states are fixed (All states have the same vocabulary and no transformations change the base set of states).
- There are several ways to implement states (They are closed under isomorphisms).

[1] This is given by the abstract state postulate.

The difference between theoretical and practical runtime states is that states are object structures in [39], while they are value structures in [23]. This difference is not serious, as objects can be considered as object IDs, and properties of objects are then functions over objects IDs. As usual, methods of objects just get an implicit parameter which is their enclosing object.

There are two perspectives to runtime structure, namely low-level (defined by the machine), and high-level (defined by the language). Low-level structure is given by the general von-Neumann architecture [33] which involves a CPU, a memory unit and input and output devices. High-level structure depends on the language used. As an example, for Java the runtime structure includes a set of objects, a program counter, threads, a stack, and exception objects [29]. There is also a part of the high-level runtime structure that depends on the algorithm itself, for example objects of Java classes. For Prolog, the runtime structure includes a (local) stack with environments and choice points, a heap (global stack) with terms, and a trail with variable bindings as described in the Warren Abstract Machine [44].

Of course, a computation cannot be run on an abstract or a virtual machine, some real (physical) machine has to be there to do the work. For example, the Java virtual machine (JVM) is typically implemented on top of a general-purpose machine, which again is based on machine code, which again is based on circuits, which again is based on electronics, which again is based on electrons, which again is based on quarks. A similar argument can be made for ASM, where the semantics of ASM has to be implemented on a standard computer. In ASM, a state change is done as one step, whereas in an implementation on a real computer, it would amount to a series of steps.

Which of these levels is the natural level for the algorithm? We can safely assume that the natural level is the highest of them, in the JVM example it would be the level of JVM operations. This means that the language of formulating the algorithm is essential, as the runtime structure can be different for different languages. Writing the same algorithm in Prolog versus in Java would imply serious changes in the runtime, i.e. the execution of the algorithm is different.

Although it is easy to forget, the runtime structure also needs a specification of the initial runtime state.

We see that ASMs provide the flexibility to use or define structures that fit the user's natural understanding of the algorithm. ASM makes explicit the implicit runtime elements of typical programming languages, e.g. the program counter. This is possible because ASM does not have any fixed runtime elements implied by the language.

4.2 Runtime Changes

Based on the runtime structure, runtime changes define what happens at runtime (dynamics), i.e. what is a computation step and what changes are done. As the runtime structure is given, only the changes are relevant. This relates to a finite set of changes on locations in the runtime structure, as already defined in [23] by the bounded exploration postulate. Bounded exploration has not been important

for practical considerations of runtime structure, as boundedness is implied by the underlying machine. In practical terms, ways to express the changes have been more important.

Minsky [31] has demonstrated that Turing completeness requires only four instruction types: conditional GOTO, unconditional GOTO, assignment, and HALT. There is one implied instruction which is sequential composition. Nowadays, GOTO is considered bad and related to "spaghetti code", so ASM introduce the same level of Turing completeness using structured programming with update (assignment), parallel composition, and if-then-else. For ease of writing, also a let and a forall are provided. In ASM, sequential composition is not available because there is no predefined program counter. HALT is implicit as the execution stops when there are no more changes, i.e. the update set is empty[2].

The ASM algorithm could also be written using different syntax, for example traditional programming language syntax. Using Java syntax, we can express the ASM Euclidean algorithm (syntactically) as follows. Warning: This is *not* Java, just Java syntax for ASM.

```
if (n1 > n2) {
    n1 = n2;
    n2 = n1;
} else if (n1 == 0) {
    gcd = n2;
} else {
    n2 = n2 − n1;
}
```

Remember that the execution mode is parallel here. We have changed the names such that they fit the Java conventions.

This formulation reveals that the syntax is not too important for ASM and it has not been focused upon much. Instead, constructs in ASM are often considered abstract syntax that can be written in different ways, as is customary in mathematics. In the abstract syntax of ASM, we need locations, updates, choices and parallel blocks.

In each state, the complete ASM program is executed. This deviates slightly from the idea of regular programming languages, where only the current statement is executed, identified by the program counter. For example, during the execution of the Java code as given in Sect. 1, the program counter keeps track of the current code position during execution. In addition, there is an extra temporary variable t for the swap between $n1$ and $n2$.

The execution in Lisp includes a number of function activation records to keep track of all the recursive calls. We have a similar situation with Prolog execution, which adds a number of variable unifications into the runtime structure.

[2] In some sense, this turns HALT from a syntactic element into a semantic element. Minsky would have been able to avoid the HALT if there was a rule that the execution stops when moving (GOTO) out of the program.

In-place state transformations can also be expressed by transformation languages like QVT [16] or ATL [25]. We do not go into more detail of those languages, as they do not add more possibilities than the languages we already discussed.

Operational semantics languages also provide possibilities to express runtime changes. The Euclidean algorithms can be expressed using SOS [35] as follows.

$$\frac{\langle n_1 > n_2, s\rangle \Rightarrow true}{(s) \longrightarrow (s \uplus \{n_1 \mapsto n_2, n_2 \mapsto n_1\})}$$

$$\frac{\langle n_1, s\rangle \Rightarrow 0}{(s) \longrightarrow (s \uplus \{gcd \mapsto n_2\})}$$

$$\frac{\langle n_1 > n_2, s\rangle \Rightarrow false, \langle n_1 > 0, s\rangle \Rightarrow true}{(s) \longrightarrow (s \uplus \{n_2 \mapsto n_2 - n_1\})}$$

In this situation, ASM are an effective formalization of pseudo-code, as they are dedicated to describing one transition only[3].

What is needed for the runtime changes is navigation of the runtime structure for reading and writing of locations. In SOS [35] and also in ASM, the current program is outside the runtime state. That is possible in ASM, as the program is constant and it is always applied as one. SOS wants to keep the current execution position, but does not use a program counter. Instead, the program is a parameter of the SOS rules. Changing the PC amounts to changing the program for the next runtime state.

We see that ASM allow an explicit description of the runtime changes based on the explicit description of the runtime structure. This works for all kinds of runtime changes, be it program executions or movements of knitting needles. If the algorithm includes a sequence of actions, advancing through the sequence step by step can be considered the natural level of abstraction, such that an implicit program counter would be needed. This is standard in imperative programming languages, and can also be provided by extended ASM variants. We discuss this aspect of the user perspective in the next section.

5 Describing Algorithms

When describing an algorithm, the focus is not on the execution of the algorithm, but the understanding on the part of the user. Algorithms are ubiquitous, and we find them nearly in all aspects of life. What is the natural level of abstraction in this case? In a first attempt, we distinguish three abstraction levels of algorithm descriptions, see also [41].

High-level descriptions are given by prose and ignore implementation details like a UML use case description.

[3] There are also advanced ASM concepts to handle structured executions, often called Turbo-ASM [9].

An implementation description is still prose and details how the high-level description is turned into something executable, for example as UML classes and activity diagrams.

A formal description gives all the detail and makes the algorithm executable, for example in Java.

It is possible to have an executable understanding of all these levels, but they differ in the level of abstraction and detail. ASMs contribute to lifting the level of formality and executability even to the high-level descriptions. The same is achieved with model-driven development, see Sect. 6.

One might argue that ASM fit the bill again, as they are proven to provide the natural level of abstraction for algorithms as discussed in Sect. 4. However, this is only true from the point of view of the machine. In many cases, this is the same point of view as for the designer and the user. As an example, computer scientists often think of algorithms in terms of the machine, and there the argument with ASM applies.

However, many other users do not look at algorithms from the point of view of the machine. Examples are algorithms that describe recipes, or knitting algorithms, or algorithms to calculate loan security, or how to assemble a piece of furniture, see Fig. 1. Typically, there are experts that know how to cook or to knit, and they will describe their algorithms in a way related to their expertise. This is usually connected to the area of domain-specific languages (DSL) [20].

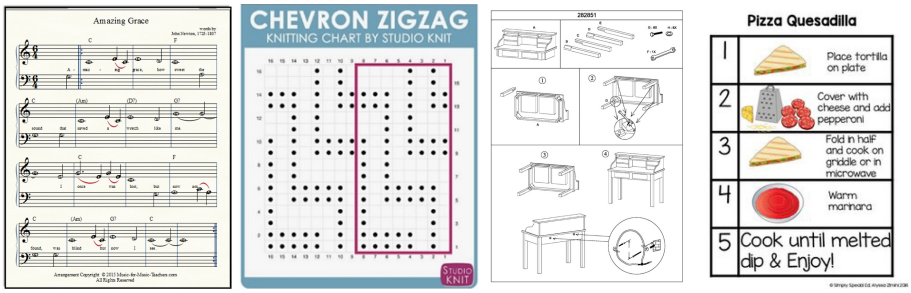

Fig. 1. Different sample algorithm descriptions

There is extensive research in the area of DSL, and the general result is that a good DSL captures the concepts of the domain in question, rather than the concepts of the underlying machine. Instead, there is a transformation process from the DSL to some lower-level language in terms of model-drivel development (MDD) [1], which is standardized in the model-driven architecture (MDA) of OMG [27]. ASM can be related here as a transformation target language.

ASM cannot and will not be the universal description language, because it is impossible to have just one language for all purposes. The language with the natural level of abstraction has to be found and developed in the domain where it

is used, together with its users. The examples in Fig. 1 are algorithm descriptions in DSLs that are not readily captured by ASM syntax.

Of course, ASMs were never intended to replace languages, especially not their concrete syntax. However, ASMs can be used to take the abstract syntax of a language and define its semantics. This has been done with UML state diagrams [11], the programming language Java [14], the specification language SDL [21], and could also be done with the music sheet and a knitting pattern. We discuss this aspect in the next section.

Even for regular algorithms, ASM are missing several essential language features of modern languages, for example classes, exceptions, namespaces, generics, inheritance, and iterators. These features might not be needed for simple algorithms, but they are essential for system-level complex algorithms.

We see that ASMs do not provide the description of algorithms in a concrete syntax on the natural level of abstraction in the same way as DSLs. Of course, this is not the intention of ASMs. Using ASM, we can define the behavior, as discussed in Sect. 4. We look at how ASMs can define languages (DSLs) in the next section.

6 Language Semantics

In connection with DSLs, there is a need to describe languages formally. How else would a DSL come to life if not using a description. Typically, meta-languages are used to describe languages, see for example the well-known OMG stack of modelling languages in Fig. 2.

Level	Example	Description
M3	MOF	Defines a language for specifying metamodels
M2	UML	Defines a language for specifying models
M1	model of a bank	Defines a language that describes a semantic domain
M0	a runtime state of the bank model	Contains runtime instances of the model elements defined in the model

Fig. 2. OMG stack

In the OMG stack, specifications (descriptions of algorithms - Sect. 5) are placed on level M1, while the language they are written in is on level M2. An algorithm written in ASM would be on M1, while ASM itself is on M2. The execution of the algorithm (Sect. 4) is on the base level M0. The level M3 is dedicated to meta-languages, i.e. languages that are used to describe languages. Often, meta-languages are already languages on their own, such they could be placed both on M2 and on M3. The definition of the meta-languages themselves

is done using the same meta-languages, where the language definition languages used are found on M3, while the languages defined are on M2 (bootstrapping).

Is ASM a good language to describe languages? To answer this question, we have to consider what it takes to define a language, i.e. which meta-languages we need. As it turns out, there are several elements that need to be described for a language, namely abstract syntax (also called structure), concrete syntax, constraints being a part of structure, and semantics (translational or operational) [24,32], see Fig. 3. We will consider these aspects one by one.

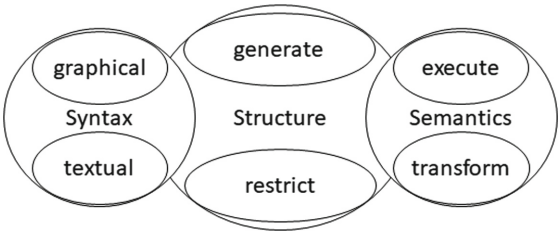

Fig. 3. Language aspects

6.1 Structure (Abstract Syntax)

The abstract syntax of a language contains the concepts of the language and their relationships with each other. Class diagrams are the method of choice to describe abstract syntax, as shown in MOF [17]. Even though ASM also allow describing structure, class diagrams are not supported in ASM. However, it is possible to use MOF diagrams to show ASM structure definitions. This way, MOF gets an ASM semantics. It should be noted that abstract syntax typically entails an abstract syntax tree, and tree structures can be expressed using ASM.

This way, ASM has support for the abstract syntax of structure definitions, but not the concrete syntax as given by MOF. Moreover, classes with inheritance are not supported by ASM, which is mainly a typing issue.

A second part of Structure is related to constraints, often expressed as OCL formulas. Logical formulas are well within the capabilities of ASM, so this part would be possible to express. The main part of the logical formulas is a way to navigate the syntax tree, and this is commonly done using expressions. More advanced DSLs for name resolution [28] are beyond the capacities of ASM. Still the semantics of all these languages can be formalized using ASM.

6.2 (Concrete) Syntax

Concrete syntax has two main forms, namely textual syntax and graphical syntax. Textual syntax is commonly given by grammars, which ASMs do not provide. Again, using grammars for analysis will finally lead to syntax trees, which can be expressed by ASM. Still, the notation of choice in this case would be

grammars. Graphical syntax could be given by graph grammars, which again cannot be written in ASM. A similar argument as before applies also here.

As for concrete syntax, ASMs do not provide the concrete syntax on the natural abstraction level. However, the semantics of grammars can be described using ASM.

6.3 Semantics

We consider two essential kinds of semantics, translational semantics and execution semantics (operational semantics), see [32] for a more detailed discussion of other kinds of semantics.

Translational Semantics refers to semantics that is given as a translation into a languages which has a given semantics already. Semantically, a translation is a simple function and it could be given by various forms of function definition. It has become customary to define transformations between abstract syntax, such that the connection between the language constructs becomes visible. In principle, ASM can define functions, but in order to define structural transformations, more dedicated languages should be used [16,25].

Dedicated transformation languages allow the specification of input and output patterns for the transformation. In addition, templates can be used to specify the result of the transformation. The semantics of transformation languages is often a function or a series of functions.

As with the previous language definition elements, ASM are able to capture the aspects semantically, but do not provide the syntax on the natural abstraction level.

Execution Semantics describes how a program is executed at runtime. It includes the runtime structure and the runtime changes as discussed in Sect. 4. ASM are very well suited to describe runtime with both runtime state and runtime changes. This is already discussed in Sect. 4. This is also the way that language semantics is given using ASM, see for example [14] and [21].

There are only few dedicated languages for the definition of execution semantics, and ASM provides all features that are needed. For application of ASM in an object-oriented language definition context, where both the language structure and the runtime environment are object-oriented, the availability of classes and inheritance in ASM would be an advantage.

SOS [35] is a DSL for the description of execution semantics. The example of SOS in Sect. 4 shows that its expressive power is comparable to ASM.

6.4 Summary

ASM shines for the formulation of execution semantics on the natural level of abstraction, which relates very well to its power in describing algorithm executions. This implies that the semantics of all meta-languages can be formalized using ASM. On the syntax side, DSLs are on a more natural abstraction level. The same applies to transformations.

7 Conclusion

We have considered the abstraction level of Abstract State Machines and whether the ASM capture the natural abstraction level of an algorithm. We have looked into three aspects of natural abstraction level, namely abstraction of executions, abstraction of descriptions, and abstraction of language semantics.

As it turns out, ASM are on the correct level of abstraction for algorithm execution, which is already established in [23]. The consideration of runtime environments brings the same result from a different perspective.

For the description of algorithms, ASMs cannot provide the correct abstraction level, as this depends on the application domain of the algorithm. Domain-specific languages are the way to provide such good descriptions, and no single language can provide the correct abstraction level.

This leads to the discussion how languages can be formalized, and whether ASM are on a natural abstraction level as a meta-language. Language design has several areas, and ASM are not on the right abstraction level for abstract syntax and concrete syntax. ASM can be used for some aspects of constraints and of transformation semantics. However, the strength of ASM is that it is on the natural abstraction level for operational semantics, which essentially is the same as repeating that ASM are on the natural abstraction level for algorithm execution.

When we connect these results to the OMG modelling levels as presented in Fig. 2, then ASM is strong on level M0 (executions), and not strong on level M1 (descriptions). On level M2, the strength of ASM is again on the execution semantics side, i.e. the connection of the description with the executions. We can interpret this such that ASM is a semantic language with little concern for syntax. It provides support to explicitly capture executions on the correct level of abstraction, and it avoids predefined execution patterns like a program counter.

This way, ASMs give just the right level of freedom for describing all execution situations on the natural level of abstraction.

References

1. Bennedsen, J., Caspersen, M.E.: Model-driven programming. In: Bennedsen, J., Caspersen, M.E., Kölling, M. (eds.) Reflections on the Teaching of Programming. LNCS, vol. 4821, pp. 116–129. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77934-6_10. http://link.springer.com/book/10.1007%2F978-3-540-77934-60
2. Blass, A., Gurevich, Y.: Ordinary interactive small-step algorithms I. ACM Trans. Comput. Log. **7**(2), 363–419 (2006)
3. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: correction and extension. ACM Trans. Comput. Log. **9**(3) (2008). https://doi.org/10.1145/1352582.1352587
4. Blass, A., Gurevich, Y., Rosenzweig, D., Rossman, B.: Interactive small-step algorithms II: abstract state machines and the characterization theorem. Log. Methods Comput. Sci. **3**(4) (2007). https://doi.org/10.2168/LMCS-3(4:4)2007

5. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Log. (TOCL) **4**(4), 578–651 (2003). https://doi.org/10.1145/937555.937561
6. Blass, A., Gurevich, Y.: Persistent queries in the behavioral theory of algorithms. ACM Trans. Comput. Log. (TOCL) **12**(2), 16:1–16:43 (2011). https://doi.org/10.1145/1877714.1877722
7. Boolos, G.S., Burgess, J.P., Jeffrey, R.C.: Computability and Logic, 5th edn. Cambridge University Press, Cambridge (2007). https://doi.org/10.1017/CBO9780511804076
8. Börger, E., Cisternino, A. (eds.): Advances in Software Engineering. LNCS, vol. 5316. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89762-0
9. Börger, E., Stärk, R.: Abstract State Machines - A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
10. Börger, E.: Construction and analysis of ground models and their refinements as a foundation for validating computer-based systems. Appl. Formal Methods **19**(2), 225–241 (2007). https://doi.org/10.1007/s00165-006-0019-y
11. Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASMs. Inf. Softw. Technol. **46**(5), 287–292 (2004). https://doi.org/10.1016/j.infsof.2003.09.009
12. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
13. Börger, E., Schewe, K.D.: Concurrent abstract state machines. Acta Informatica **53**(5), 469–492 (2016). https://doi.org/10.1007/s00236-015-0249-7
14. Börger, E., Schulte, W.: A programmer friendly modular definition of the semantics of Java. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 353–404. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48737-9_10
15. Brooks Jr, F.P.: No silver bullet essence and accidents of software engineering. Computer **20**(4), 10–19 (1987). https://doi.org/10.1109/MC.1987.1663532. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1663532
16. Editor OMG: Meta object facility (MOF) 2.0 query/view/transformation specification, version 1.1. Object Management Group (2011). http://www.omg.org/spec/QVT/1.1/
17. Editor OMG: Meta object facility (MOF). Object Management Group (2016). https://www.omg.org/spec/MOF
18. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992)
19. Fischer, J., Møller-Pedersen, B., Prinz, A.: Real models are really on M0- or how to make programmers use modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development-Volume 1: MODELSWARD, pp. 307–318. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008928403070318
20. Fowler, M.: Domain Specific Languages, 1st edn. Addison-Wesley Professional, Boston (2010)
21. Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of SDL-2000: status and perspectives. Comput. Netw. **42**(3), 343–358 (2003). https://doi.org/10.1016/S1389-1286(03)00247-0
22. Gurevich, Y.: Evolving algebras 1993: lipari guide. In: Specification and Validation Methods, pp. 231–243. Oxford University Press (1995)

23. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. (TOCL) **1**(1), 77–111 (2000). https://doi.org/10.1145/343369.343384
24. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? Computer **37**(10), 64–72 (2004). https://doi.org/10.1109/MC.2004.172
25. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008). https://doi.org/10.1016/j.scico.2007.08.002
26. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0039592
27. Kleppe, A., Warmer, J.: MDA Explained. Addison-Wesley, Boston (2003)
28. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Declarative name binding and scope rules. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 311–331. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_18
29. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional, Boston (2014)
30. Madsen, O.L., Møller-Pedersen, B.: This is not a model. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 206–224. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_13
31. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Inc., Englewood Cliffs (1967)
32. Mu, L., Gjøsæter, T., Prinz, A., Tveit, M.S.: Specification of modelling languages in a flexible meta-model architecture. In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA 2010, pp. 302–308. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1842752.1842807
33. von Neumann, J.: First draft of a report on the EDVAC. In: Randell, B. (ed.) The Origins of Digital Computers. MCS, pp. 383–392. Springer, Heidelberg (1982). https://doi.org/10.1007/978-3-642-61812-3_30
34. Ouimet, M., Lundqvist, K.: The timed abstract state machine language: abstract state machines for real-time system engineering. J. UCS **14**, 2007–2033 (2008)
35. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004). https://doi.org/10.1016/j.jlap.2004.05.001. Structural Operational Semantics
36. Prinz, A.: Distributed computing on distributed memory. In: Khendek, F., Gotzhein, R. (eds.) SAM 2018. LNCS, vol. 11150, pp. 67–84. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01042-3_5
37. Prinz, A., Møller-Pedersen, B., Fischer, J.: Object-oriented operational semantics. In: Grabowski, J., Herbold, S. (eds.) SAM 2016. LNCS, vol. 9959, pp. 132–147. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46613-2_9
38. Prinz, A., Sherratt, E.: Distributed ASM - pitfalls and solutions. In: Aït-Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 210–215. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_18
39. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72901-3_12

40. Schewe, K.-D., Wang, Q.: A simplified parallel ASM thesis. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 341–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_27
41. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Course Technology, Boston (2013)
42. Stone, H.S.: Introduction to Computer Organization and Data Structures. McGraw-Hill, New York (1972)
43. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **s2-42**(1), 230–265 (1937). https://doi.org/10.1112/plms/s2-42.1.230. https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230
44. Warren, D.S.: WAM for everyone: a virtual machine for logic programming, pp. 237–277. Association for Computing Machinery and Morgan & Claypool (2018). https://doi.org/10.1145/3191315.3191320