From Concept to Code: Unveiling a Tool for Translating Abstract State Machines into Java Code*

Andrea Bombarda[1 [0000−0003−4244−9319]], Silvia Bonfanti[1 [0000−0001−9679−4551]], and Angelo Gargantini[1 [0000−0002−4035−0131]]

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione,
Università degli Studi di Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it

Abstract. Formal methods play a crucial role in modeling and quality assurance, but to be deployed on real systems, formal specifications need to be translated into implementation. Manually converting formal models into code poses challenges such as increased costs, limitations in specification reuse, and the potential for introducing errors. To overcome these limitations, Model-Driven Engineering (MDE) approaches enable developers to generate software code automatically. This paper proposes the `Asmeta2Java` tool for the automatic translation of formal `Asmeta` specifications into executable Java code. The designers start at an abstract level and perform refinement steps and verification activities. At the end, they automatically generates the code by applying the model-to-code transformation. Moreover, a process to validate and evaluate the transformation is presented.

1 Introduction

Formal methods are frequently employed to abstractly model the requirements of a system. These formal requirement models can undergo various quality assurance processes, including dynamic analysis such as simulation or animation, and static analysis like property verification. Nevertheless, establishing a clear connection between the ultimate implementation and its abstract specification can be challenging, as the implementation may incorporate platform-specific details, be expressed in a programming language, and operate on specific hardware. Engaging in the manual conversion of formal models into code not only raises expenses but also restricts the potential for reusing a formal specification. This approach is susceptible to errors, even if validation and verification activities have been performed at the model-level, as faults may be introduced during the code-writing process, and it can act as a hindrance to the broader acceptance of formal methods. Assisting designers during this critical final phase of the development process would be highly beneficial.

Abstract State Machines (ASMs) [16,10] are a formal method proposing a rigorous software and systems development process, starting from the modeling to the model simulation and animation, from the model validation to its verification, and, finally, to

* The work of Andrea Bombarda is supported by PNRR - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 – CUP: B53C22006700001 - Spoke 1 - Pilot 1.4.

the transformation of informal requirements into implementation. The Asmeta framework [2] supports the user during the complete process and allows him/her to work by refinement: all the details of the system under analysis are captured by a sequence of refined models with decreasing levels of abstraction and, during the process, validation and verification (V&V) activities are executed. At the end of this iterative process, the final step consists in translating the formal specification into the actual implementation, e.g., in Java. Classically, this activity is performed manually but, as mentioned above, having a way to automatize it is advisable (e.g., in the case in which the specification error is discovered while running the implementation, and having a way to automatically regenerate the code after a fix in the specification can facilitate developers' work). However, translating ASM specifications into a classical programming language such as Java, is challenging as ASMs support complex constructs (e.g., parallelism of rules, non-determinism, abstract domains, etc.) that have to be correctly mapped to general-purpose constructs in the chosen programming language.

In this paper, we present our approach for automatically translating Asmeta specifications into Java code. This approach is supported by the Asmeta2Java tool. As our previous effort [9] addressed the translation from Asmeta to C++, Asmeta2Java automatically maps Asmeta functions, rules, and domains into Java functions, methods, variables, and objects. The presented approach is a typical application of the model-driven engineering (MDE) paradigm [17] where, starting from the model of the system (the Asmeta specification), the executable source code embeddable into the final system is obtained. For this reason, the rigorous quality and the correctness of the whole automatic process have to be assured. Thus, we have spent considerable effort not only in defining suitable transformations but also in building a framework for the validation of such transformations. Indeed, we have assured the correctness by checking whether the generated code was compilable and did not contain any syntactic errors, and by statically analyzing the source code.

The remainder of this paper is structured as follows. In Sect. 2, we introduce the basics of ASMs and of the Asmeta framework. The generation of Java code is explained in Sect. 3, where we explain the principles guiding our translation, the process we follow, and we show how Asmeta constructs and concepts are translated into Java. Sect. 4 shows our proposal for the validation of the transformation of Asmeta specification into Java code. Finally, Sect. 5, Sect. 6, and Sect. 7, respectively, present related work, discuss future work, and conclude the paper.

2 The ASMETA framework

In this section, we introduce Abstract State Machines (ASMs) and the Asmeta framework [2], which embeds the Asmeta2Java tool presented in this paper. The Asmeta framework comprises various tools designed to assist developers in different stages of the software life-cycle: design, development, and operation (see Fig. 1). The *design* phase involves conducting activities such as modeling, validation, and verification, starting from the system requirements. Once the requirements are formalized and verified, they can be enhanced by incorporating additional details into the existing model in a chain of refinements. Subsequently, *development* and *operation* can proceed inde-

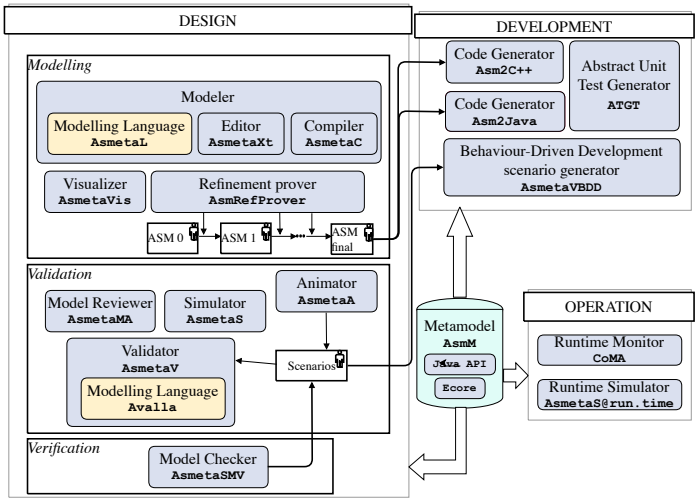

Fig. 1: The ASM development process powered by the `Asmeta` framework

pendently. The former entails generating code from models, while the latter facilitates runtime simulation and monitoring activities. In this paper, we will focus on the *development* phase, and, in particular, we introduce the `Asmeta2Java` tool.

The `Asmeta` framework supports Abstract State Machines (ASMs), a precise state-based formal method that transforms abstract *states* through the use of *transition rules*. Each state comprises algebraic structures, representing domains of objects with functions and predicates defined on them. Specifically, an `Asmeta` specification can be seen as an executable pseudo-code, as the well-known example shown in Listing 1. This specification models the behavior of a simple coffee vending machine. It can dispense three different types of products, namely coffee, tea, and milk, and requires the user to insert one or a half coin. If a half coin is inserted, the machine can dispense milk, if it is still available. Otherwise, if the user inserts a coin, the machine randomly chooses tea or coffee, if they are available. As in the case of the example of the coffee vending machine, an `Asmeta` specification comprises four different sections:

– The signature section contains the declaration of the *domains* and *functions*. The StandardLibrary provides predefined domains (such as Integers, Boolean, String, and so on), but customized and static/dynamic ones can be declared by the user (e.g., the CoinType, Product, QuantityDomain and CoinDomain in Listing 1). Functions are classified into three categories: static, derived, and dynamic. More specifically, dynamic functions are further classified in: controlled, monitored, out, and shared. Controlled functions (as the coins and available ones in Listing 1) are read in the current state and updated in the next one by the machine, while monitored functions (as insertedCoin in Listing 1) are updated by the environment and read by the machine. Finally, out functions are written by the machine and sent as output to the environment, and shared functions are written and read both by the machine and the environment.

```
1  asm coffeeVendingMachine
2  import StandardLibrary
3
4  signature:
5  enum domain CoinType = {HALF, ONE}
6  enum domain Product = {COFFEE, TEA, MILK}
7  domain QuantityDomain subsetof Integer
8  domain CoinDomain subsetof Integer
9
10 controlled coins: CoinDomain
11 controlled available: Product -> QuantityDomain
12 monitored insertedCoin: CoinType
13
14 definitions:
15 domain QuantityDomain = {0 : 10}
16 domain CoinDomain = {0 : 25}
```

```
18 rule r_serveProduct($p in Product) = par
19   available($p) := available($p) - 1
20   coins := coins + 1
21  endpar
22
23 main rule r_Main = if(coins < 25) then
24   if(insertedCoin = HALF) then
25    if(available(MILK) > 0) then
26     r_serveProduct[MILK] endif
27   else choose $p in Product with $p != MILK and
         available($p) > 0 do r_serveProduct[$p]
28   endif endif
29
30 default init s0:
31  function coins = 0
32  function available($p in Product) = 10
```

Listing 1: Example of an Asmeta specification for the Coffee Vending Machine

- The definitions section contains the specification of *static/derived functions*, *domains*, and *transition rules*. Various transition rules can be employed based on the system's behavior. For instance, the update rule assigns a specific value to a function, while guarded updates (if-then and switch-case) execute actions conditionally. Simultaneous updates (par) allow for parallel execution, while sequential execution is allowed by the seq construct. For example, Listing 1 utilizes the par rule to concurrently perform two updates in the r_serveProduct rule. Asmeta supports the non-determinism by means of the choose rule (such as at line 27 in Listing 1). With this rule, a random element, possibly satisfying a defined condition, is picked out from a certain domain (the Product domain, in our case).
- The main rule is the entry point of each computation step. When the main rule is executed, the other rules are called depending on the implemented behavior.
- The default init section initializes the value of the dynamic functions (e.g., in Listing 1, the coins function is initialized to 0 in the initial state).

After having modeled the requirements, the developers can validate the Asmeta specification by using the simulator AsmetaS [4] or animate it with AsmetaA [7]. Additionally, scenario-based validation is supported by the validator AsmetaV [11]. Besides validation, system properties can be verified with the model checker AsmetaSMV [3]. When executable source code is needed, e.g., to be embedded in other software systems, it can be automatically obtained starting from Asmeta specifications. In the case C++ code is needed, the tool Asm2C++ [8] can be used.

3 Generation of Java code

In this section, we describe the process of Java code generation from Asmeta specifications. First, we present the principles we have considered when implementing the Asmeta2Java tool and the design process. Then, we describe how Asmeta concepts are mapped into Java ones.

3.1 Principles for the translation

First, we discuss some principles we wanted to follow when defining the translation and implementing it. We came up with the following general principles.

1. *Quality* of the produced code: we want that the Java code produced from `Asmeta` specifications is not only correct, but it must have some code qualities, like readability, which normally are not sought in generated code. The reason these qualities are sometimes neglected is that the generated code is generally not meant to be read or understood. In our opinion, instead, we believe that also generated code must be understandable and easy to inspect.
2. *Traceability* refers to the possibility of easily mapping Java constructs in the generated code with specification elements. For example, we will try not to change the names of entities (like rules, functions, and so on). This facilitates understanding how `Asmeta` behavior is translated into Java.
3. *Minimality* consists in avoiding adding useless code. We have observed in the past that generated code often contains some boilerplate code that is always present, regardless of the specification that is translated. We tried to put in the generated code only what is necessary and leave out what is useless.
4. *Extendability* of the generated code: we assume that the programmer may want to extend the generated code, by adding some details (for instance, data structures) that do not have the abstractness necessary to be considered in the formal specifications. For example, we rarely use the `final` keyword, so the programmers can extend the code without limitations. We recognize that allowing the user to extend the generated code may lead to changing or interfering with the actual behavior of the system. However, considering the higher expressiveness of high-level programming languages and the complexity of actual systems, extending the code is sometimes needed.

Other qualities, like efficiency, were not considered as important as those presented above. During the definition of the translation, we relied on these principles: when multiple alternatives to translate the same concept were available, we chose the one maximizing them. However, sometimes the principles are in contrast. For example, what should be done with variables declared but never used? Should they be added to Java code or not? If one favors traceability, they must be added; if one favors minimality instead, they can be safely skipped.

3.2 Design of the process for implementing the transformations

Different approaches could be applied to implement the transformation process from `Asmeta` specification to Java code. We have analyzed both the Model to Text (M2T) and Model to Model + Model to Text (M2M + M2T) [13] approaches. We noted that only the meta-model of the source model (`Asmeta`) is available, while the target meta-model, i.e, Java, is not. For this reason, and to avoid the intermediate steps required by the M2M + M2T approach to translate the original specification into executable Java code, we have selected the M2T approach.

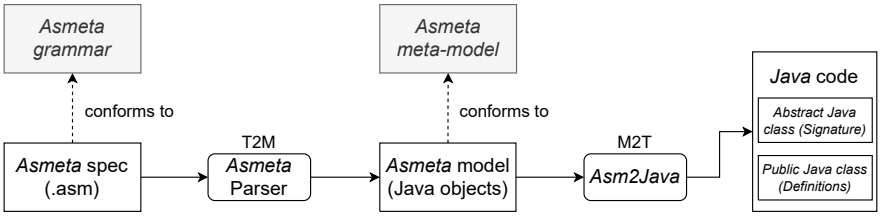

Fig. 2: Transformation process: from `Asmeta` specification to Java code

The approach is presented in Fig. 2. Given an `Asmeta` specification conforming to the `Asmeta` grammar, by applying the Text to Model (T2M) transformation by means of the `Asmeta` parser, it is possible to quickly get the `Asmeta` model (in terms of Java objects) conforming to the `Asmeta` meta-model. Given the `Asmeta` model, the `Asmeta2Java` tool employs a M2T transformation to automatically generate the Java code.

Asm2Java The translation performed by the `Asmeta2Java` tool is implemented in the `Xtend` programming language, a flexible and expressive dialect of Java that is integrated into the `Xtext`[1] framework. It is based on the Eclipse modeling framework (EMF) and provides code generation facilities like a meta-model (Ecore) to describe the models, a set of Java classes for the components of the model, a set of adapter classes that enable viewing and command-based editing of the model, a basic editor, and the integrated development environment (IDE).

When translating an `Asmeta` specification with the name `asmName`, the `Asmeta2Java` tool generates two classes in a single file (`asmName`.java): an abstract class (`asmName`Sig) which mainly contains the signature of the `Asmeta` specification, and a public Java class (in `asmName`) which contains the rule definitions and extends the class `asmName`Sig.

3.3 Translation of `Asmeta` into Java

In this section, we describe how an `Asmeta` specification is translated into Java. First, we need to introduce the differences between the state evolution in the two approaches, as many design decisions have been taken to address these differences and obtain the same behavior between the formal model and its code implementation.

ASM State evolution in Java. One of the main differences between ASM and Java is the execution of a machine in ASM or of a program in Java. In ASM, the execution consists of a sequence of steps where each step executes the main rule and builds the update set, which is applied in order to obtain the next state. On the other hand, in Java, the execution corresponds to a sequence of instructions executed step by step, and the values are updated at each instruction. The emulation of an ASM step has been implemented by using the method `updateASM()` in Java. This method first calls the

[1] `http://www.eclipse.org/Xtext/`

translation of the main rule, which computes the update set. Then, it applies with the method `fireUpdateSet()` the update set to the values of the locations in the new state.

The translation of the coffee vending machine example is shown in Listing 2. The `Asmeta` signature is translated into the abstract class `coffeeVendingMachineSig` (see line 1), which contains the definition of domains and functions, and the rule declaration. Then, the rules are implemented as Java methods in the `coffeeVendingMachine` class (see line 52), which extends the abstract class `coffeeVendingMachineSig`, and overrides the previously defined abstract methods. In addition, this class implements the methods performing the application of the update set (`fireUpdateSet()` at line 94), and the execution of the simulation step (`updateASM()` at line 99). As shown in the code, `updateASM()` calls the translation of the main rule as the first instruction (`r_Main()` at line 76). The execution of the `coffeeVendingMachine` can be performed by instantiating an object of that class and repeatedly calling (e.g., in a loop) the `updateASM()` method.

In the following, we explain in detail how the different `Asmeta` structures and concepts are translated in Java. Note that only a subset of the `Asmeta` language is supported, at the moment, by `Asmeta2Java`. If an `Asmeta` specification contains some unsupported constructs, domains, or terms, the translation fails. For example, the following are unsupported domains: Complex, Agent, and rule domains. We are working on extending `Asmeta2Java` to fully support all `Asmeta` constructs, domains, and terms.

Functions. To translate functions, we introduce four inner classes: `Fun0Ctrl` and `Fun0` for 0-ary functions (respectively controlled or not) and `FunNCtrl` and `FunN` for n-ary functions (respectively controlled or not), shown in Listing 3. These classes have a method `set` that sets the value to be taken by the function in the next step for the controlled one or in the current state for the monitored ones. They have a method `get` that returns the value of the function in the current state.

Each `Asmeta` function is translated into an object of these classes, by using the mapping shown in Tab. 1. This specific way to translate `Asmeta` functions in Java has been chosen to implement the same state-evolution process in both approaches, namely in `Asmeta` and Java (see Sect. 3.4 for additional details).

Domains. The translation of `Asmeta` domains into Java datatypes or classes is reported in Tab. 2. `Asmeta` domains are implemented using Java datatypes when there is a straightforward mapping, e.g., for primitive types or collections. In the case of an abstract domain, its translation is a regular class with the constructor that takes only the name of the abstract element. In `elems`, the class *D* keeps track of all the elements belonging to the abstract domain.

Also in the case of concrete domains (defined as subset of other domains), the translation in Java requires to declare, for each of them, a new class. Note that, in `Asmeta` elements of a concrete domain *D*1 subset of another domain *D*2 can be interchangeable with elements of the parent domain *D*2, and vice versa, while in Java this is not possible, since *D*1 and *D*2 are unrelated types in Java. We could not use the inheritance in Java, because several Java types we use (like Integer) are `final`. Thus, the *D*1

```
abstract class coffeeVendingMachineSig {
 // Enumerative domains
 static enum CoinType {HALF,ONE}
 static enum Product {COFFEE,TEA,MILK}

 // Concrete domain
 static class QuantityDomain {
  static List<Integer> elems = new ArrayList<>();
  Integer value;

  static QuantityDomain valueOf(Integer val) {
   QuantityDomain n = new QuantityDomain();
   n.value = val;
   return n;
  }

  static QuantityDomain valueOf(QuantityDomain val) {
          return val; }

  @Override public boolean equals(Object obj) {
   if (!(obj instanceof QuantityDomain))
    return false;
   return value.equals(((QuantityDomain) obj).value);
  }

  @Override public int hashCode() {
    return value.hashCode(); }
 }

 // Concrete domain
 static class CoinDomain { ... }
 CoinDomain_elem CoinDomain_elem = new CoinDomain();

 // Classes for controlled functions
 class Fun0Ctrl<C> {...}
 static class FunNCtrl<D, C> {...}

 // Classes for monitored functions
 class Fun0<C> {...}
 class FunN<D, C> {...}

 // Controlled functions
 Fun0Ctrl<CoinDomain> coins = new Fun0Ctrl<>();
 FunNCtrl<Product, QuantityDomain> available = new
       FunNCtrl<>();

 // Monitored functions
 Fun0<CoinType> insertedCoin = new Fun0<>();

 // RULE DEFINITION
 abstract void r_serveProduct(Product _p);
 abstract void r_Main();
}
public class coffeeVendingMachine extends
      coffeeVendingMachineSig {
 coffeeVendingMachine() {
  // Static domain initialization
  QuantityDomain.elems = Collections.unmodifiableList
        (Arrays.asList(0, 1, ..., 10));
  ...

  // Function initialization
  CoinDomain_elem.value = 0;
  coins.oldValue = coins.newValue = CoinDomain_elem;
  for (Product _p:Product.values()) {
   QuantityDomain a = new QuantityDomain();
   a.value = 10;
   available.oldValues.put(_p, a);
   available.newValues.put(_p, a);
  }
 }

 // Translation of ASM rules into Java
 @Override
 void r_serveProduct(Product _p) { { //par
   QuantityDomain qD = new QuantityDomain();
   qD.value = available.get(_p).value - 1;
   available.set(_p, qD);
   ... } //endpar
 }

 @Override
 void r_Main() {
  if ((coins.get().value < 25)) {
   if ((insertedCoin.get() == CoinType.HALF)) {
    if ((available.get(Product.MILK).value > 0))
     r_serveProduct(Product.MILK);
   } else {
    List<Product> point0 = new ArrayList<Product>();
    for (Product _p : Product.values())
     if ((_p != Product.MILK) && (available.get(_p).
           value > 0))
      point0.add(_p);
    int rndm = ThreadLocalRandom.current().nextInt(0,
          point0.size());
    Product _p = point0.get(rndm);
    if (point0.size() > 0) r_serveProduct(_p);
   }
  }
 }

 // Update set applying
 void fireUpdateSet() {
  coins.oldValue = coins.newValue;
  available.oldValues = available.newValues;
 }

 void updateASM() {
  r_Main();
  fireUpdateSet();
 }
}
```

Listing 2: Translation of the Coffee Vending Machine `Asmeta` specification to Java

class includes, for each object, the corresponding value in the parent domain $D2$ and some utility method that allows the conversion from $D1$ and $D2$, such as `valueOf`, and the comparison between objects in $D1$ with those in $D2$, like `equals` and `hashCode`. Additionally, a list of all the "primitive" values is kept and is `static`. Finally, in the case of the ProductDomains (i.e., domains defined over the Cartesian product of more domains) they are translated in Java as `Pairs`, `Triplets`, and so on. We import these types from the Apache Commons library.

Terms. Tab. 3 reports the mapping between `Asmeta` terms and Java constructs. Some mapping is straightforward, such as for the ConditionalTerm and CaseTerm, which can be easily transformed into conditional operations in Java. However, `Asmeta` implements different types of terms, but some of them are not available in classical program-

```
class Fun0Ctrl<C> {
  C oldValue;
  C newValue;

  void set(C d) { newValue = d; }
  C get() {return oldValue;}
}
class FunNCtrl<D, C> {
  Map<D, C> oldValues = new HashMap<>();
  Map<D, C> newValues = new HashMap<>();

  void set(D d, C c) {
    newValues.put(d, c); }
  C get(D d) {
    return oldValues.get(d); }
}
```

(a) Classes for controlled functions

```
class Fun0<C> {
  C value;

  void set(C d) {value = d; }
  C get() { return value; }
}
class FunN<D, C> {
  Map<D, C> values = new HashMap<>();

  void set(D d, C c) {
    values.put(d, c);
  }
  C get(D d) {
    return values.get(d);
  }
}
```

(b) Classes for monitored functions

Listing 3: Java classes for controlled and monitored functions

Asmeta	**Translation in Java**
static F: C	`static` $\tau(C)$ F;
static F: D-> C	`static` $\tau(C)$ $F(\tau(D))$;
derived F: C	$\tau(C)$ $F()$;
derived F: D-> C	$\tau(C)$ $F(\tau(D))$;
dynamic monitored F: C	`Fun0<`$\tau(C)$`>` F;
dynamic monitored F: D-> C	`FunN<`$\tau(D)$, $\tau(C)$`>` F;
dynamic controlled F: C	`Fun0Ctrl<`$\tau(C)$`>` F;
dynamic controlled F: D-> C	`FunNCtrl<`$\tau(D)$, $\tau(C)$`>` F;
dynamic out F:C	`Fun0<`$\tau(C)$`>` F;
dynamic out F: D-> C	`FunN<`$\tau(D)$, $\tau(C)$`>` F;

Table 1: Function - translation from `Asmeta` to Java. F: function name, D: function domain, C: function codomain, $\tau(C)$: codomain translation, $\tau(D)$: domain translation

ming languages like Java. This is the case of ForAllTerm, ExistsTerm, and LetTerm, which do not have a direct correspondence in Java. To solve this limitation, we adopt the `java.util.function.Function<A,B>` interface, which allows one to define a function that takes an object of type `A` and returns an object of type `B`. To execute the function, we call the `apply` method.

Transition Rules. Transition rules are translated into class methods, where every rule is implemented using the Java libraries and basic constructs. The translation is linear most of the time. For example, the conditional rule is translated with the if-else construct in Java, as shown in Tab. 4, where terms and rules are also translated accordingly. A particular translation has been adopted for the update rule, which may be critical in the presence of parallelism (see Sect. 3.4).

Asmeta	Translation in Java
Natural	`Integer`
Integer	`Integer`
Real	`Double`
String	`String`
Char	`char`
Boolean	`Boolean`
Powerset	`HashSet`
Bag	`HashBag`
Sequence	`ArrayList`
Map	`HashMap`
enum	`enum`

abstract domain D	`public class` D `{`
	`  static List<`D`> elems = new ArrayList<>();`
	`  `D`(String name){...}`
	`  ...`
	`}`

[dynamic] domain $D1$ subsetof $D2$
$D1$: name of the concrete domain
$D2$: type-domain which identifies the structure of the elements of $D1$
	`[static] class` $D1$ `{`
	`  static List<`$\tau(D2)$`> elems = new ArrayList<>();`
	`  `$\tau(D2)$` value;`
	`  D1(`$\tau(D2)$` val) { ... }`
	`  static D1 valueOf(`$\tau(D2)$` val) {...}`
	`  static D1 valueOf(D1 val) {...}`
	`  public boolean equals(Object obj) {...}`
	`  public int hashCode() {...}`
	`}`
	$D1$: name of the concrete domain
	$\tau(D2)$: translation of type-domain $D2$

Prod($D1$, $D2$)
Prod($D1$, $D2$, $D3$)
...
Prod($D1$, $D2$, ..., $D10$)
$D1$, $D2$, $D3$..., $D10$: domains over which the cartesian product is defined
	`Pair<`$D1$`, `$D2$`>`
	`Triplet<`$D1$`, `$D2$`, `$D3$`>`
	...
	`Decade<`$D1$`, `$D2$`, ..., `$D10$`>`
	$D1$, $D2$, $D3$,..., $D10$: domains over which the cartesian product is defined

[dynamic] domain $D1$ subsetof t_D
domain $D1 = \{5,9,12\}$
$D1$: name of the concrete domain
t_D: type-domain which identifies the structure of the elements of $D1$
{5,9,12}: domain elements
	`[static] class` $D1$ `{ .. }`
	`List<`$\tau(t_D)$`> `$D1$`_elems =`
	`  Collections.unmodifiableList(`
	`  Arrays.asList(5,9,12);`
	$D1$: name of the concrete domain
	$\tau(t_D)$: translation of type-domain t_D
	{5,9,12}: domain elements

Table 2: Domain definition - translation from `Asmeta` to Java

3.4 Mapping `Asmeta` parallelism and non-determinism in Java

ASMs have two semantic concepts that do not have a direct implementation in Java. The first one is the parallel execution of all rules, while the other problem is non-determinism. How these two issues have been addressed by our approach is explained below.

Parallelism. Parallelism is a fundamental point of ASMs because the operations performed in the same transition step are done in parallel. Since we did not want to introduce multithreading (which may result in unforeseen behavior and complicate the translation), we decided to use standard Java. We have implemented a parallel-like execution by creating, when necessary, a copy of the current state and using that in a sequentially executed thread, following the approach proposed by others, including J. Schmid [22]. ASMs run in discrete steps, where each step consists of four operations: acquire inputs, perform the main rule, update the state, and release the outputs. The

Term	`Asmeta`	**Translation in Java**

Conditional Term	if *cond* then
	t_{then}
	[else
	t_{else}]
	endif
		`if` $\tau(cond)$ `{`
		$\tau(t_{then})$`;`
		`[}` `else {`
		$\tau(t_{else})$`;]`
		`}`

	cond: term (Boolean condition) t_{then}, t_{else}: terms	$\tau(cond)$: conditional term translation $\tau(t_{then})$, $\tau(t_{else})$: terms translation

Case Term	switch t
	case t_1 : s_1
	...
	case t_k : s_k
	[otherwise
	s_{k+1}]
	endswitch
		`if (`$\tau(t)$`==`$\tau(t_1)$`) {`
		$\tau(s_1)$`;`
		...
		`} else if (`$\tau(t)$`==`$\tau(t_k)$`) {`
		$\tau(s_k)$`;`
		`} [else {`
		$\tau(s_{k+1})$`;`
		`}]`

	t_1,...t_k: terms s_1, ..., s_k, s_{k+1}: terms	$\tau(t)$, $\tau(t_1)$, ..., $\tau(t_k)$: terms translation $\tau(s_1)$, ..., $\tau(s_k)$, $\tau(s_{k+1})$: terms translation

Tuple Term	(t_1,t_2)
	...
	(t_1,...t_{10})
		`Pair.with(`$\tau(t_1)$`, `$\tau(t_2)$`)`
		...
		`Decade.with(`$\tau(t_1)$`,...`$\tau(t_{10})$`)`

	t_1,...t_{10}: terms	$\tau(t_1)$, ..., $\tau(t_{10})$: terms translation

ForAll Term	forall v_1 in D_1, ..., v_k in D_k
	with $G_{v_1,...,v_k}$
		`new Function<Void, Boolean>() {`
		`@Override`
		`public Boolean apply(Void input) {`
		`for(`$\tau(D_1)$ $\tau(v_1)$`: `$\tau(D_1)$`.elems) {`
		...
		`for(`$\tau(D_k)$ $\tau(v_k)$`: `$\tau(D_k)$`.elems) {`
		`if(!`$\tau(G_{v_1,...,v_k})$`) {`
		`return false;`
		`}}...}`
		`return true;`
		`}}.apply(null);`

	v_1,...v_k: variables D_1,...D_k: domains where v_i take values $G_{v_1,...,v_k}$: term (condition over v_1,...v_k)	$\tau(v_1)$,..., $\tau(v_1)$: variables translation $\tau(D_1)$, ..., $\tau(D_k)$: domains translation $\tau(G_{v_1,...,v_k})$: term translation

Exist Term	exist v_1 in D_1, ..., v_k in D_k
	with $G_{v_1,...,v_k}$
		`new Function<Void, Boolean>() {`
		`@Override`
		`public Boolean apply(Void input) {`
		`for(`$\tau(D_1)$ $\tau(v_1)$`: `$\tau(D_1)$`.elems) {`
		...
		`for(`$\tau(D_k)$ $\tau(v_k)$`: `$\tau(D_k)$`.elems) {`
		`if(`$\tau(G_{v_1,...,v_k})$`) {`
		`return true;`
		`}}...}`
		`return false;`
		`}}.apply(null);`

	v_1,...v_k: variables D_1,...D_k: domains where v_i take value $G_{v_1,...,v_k}$: term (condition over v_1,...v_k)	$\tau(v_1)$,..., $\tau(v_1)$: variables translation $\tau(D_1)$, ..., $\tau(D_k)$: domains translation $\tau(G_{v_1,...,v_k})$: term translation

Let Term	let (v_1=t_1, ..., v_k=t_k) in $t_{v_1,...,v_k}$
	endlet
		`new Function<Void, `$\tau(D)$`>() {`
		`@Override`
		`public `$\tau(D)$` apply(Void input) {`
		$\tau(D_1)$ $\tau(v_1)$ = $\tau(t_1)$`;`
		...
		$\tau(D_k)$ $\tau(v_k)$ = $\tau(t_k)$`;`
		`return `$\tau(t_{v_1,...,v_k})$`;`
		`}.apply(null);`

	v_1,...v_k: variables t_1, ..., t_k: terms $t_{v_1,...,v_k}$:term with free occurrences of v_1, ..., v_k	$\tau(D)$: transl. of the domain where $t_{v_1,...,v_k}$ takes value $\tau(D_1)$, ..., $\tau(D_k)$: transl. of the domains of v_i $\tau(v_1)$, ..., $\tau(v_k)$: variables translation $\tau(t_1)$, ..., $\tau(t_k)$: terms translation $\tau(t_{v_1,...,v_k})$: terms translation
Size Of Enumerable Term	|cSet| or |cBag| or |cMap| or |cList|	`cSet.size()` or `cBag.size()` or `cMap.size()` or `cList.size()`

Table 3: Terms - translation from `Asmeta` to Java

machine state (represented by controlled functions) is modified only during the execution of the main rule. To simulate parallel execution, the controlled part of the state is duplicated: the present state (i.e., oldValue) and the future state (i.e., newValue). Modifications made to controlled functions will affect only the future state, while status readings will refer to the current state. The modification made to controlled functions

Rule	Asmeta	Translation in Java
Definition	rule r_1 (x_1 in D_1, ..., x_k in D_k) =	`void` r_1 `(`$\tau(D_1)$ x_1`,...,`$\tau(D_k)$ x_k`) {...}`
	$r1$: rule name x_1,...,x_k: parameters of the rule D_1,...,D_k:domains where parameters take value	r: method name $\tau(D_1)$, ..., $\tau(D_k)$: translation of domains x_1, ..., x_k: parameters of the method
Update	$l := t$	$\tau(l)$`.set(`$\tau(t)$`.get());`
	l: location term or variable t: generic term	$\tau(l)$: translation of location term or variable $\tau(t)$: term translation

Conditional	if *cond* then
	R_{then}
	[else
	R_{else}]
	endif
		`if (`τ(*cond*)`){`
		$\tau(R_{then})$`;`
		`[}else {`
		$\tau(R_{else})$`;]`
		`}`

	cond: term representing a boolean condition R_{then}, R_{else}: transition rules	τ(*cond*): conditional term translation $\tau(R_{then})$, $\tau(R_{else})$: rules translation

Table 4: Rules - translation from Asmeta to Java

```
controlled x: Integer
controlled y: Integer
...
rule r_parallel =
 par
  x := y
  y := x
 endpar
```

(a) Asmeta

```
void r_parallel() { //par
        x.set(y.get());
        y.set(x.get());
} //endpar
void fireUpdateSet() {
        x.oldValue = x.newValue;
        y.oldValue = y.newValue;
}
```

(b) Java code

Listing 4: Parallelism - translation in Java

will take place only when the execution of the main rule is finished, which means in the `fireUpdateSet()` function. This approach guarantees the proper evolution of the machine state, even though it is not true parallelism. The code reported in Listing 4 shows a simple example of parallelism, swapping the values of variables `x` and `y`. If we translated this example in Java as a simple sequence `x=y; y=x;`, the result would have not been the same as in ASM: both variables `x` and `y` would contain the value of `y`. For this reason, as presented in Sect. 3.3, we represented the controlled functions by using classes and embedding into them both the present state and the future one. In this way, we translate each assignment in Java following this rule: we set the next state by using the `set` method, and we access to the current state of controlled functions by using the `get` method. After the execution of the rules in the current state (the `mainRule()` method in Java), we assign all the values of the next state to the current state with the `fireUpdateSet()` method. The new current state is used in the next step of execution. We do not consider the case of inconsistent updates (a function assumes two different values in the same state) because the methodologies applied during the analysis of the ASMETA model guarantee that inconsistent updates will not occur as the model should be checked priorly its translation with the `AsmetaMA` tool (see [2] for further information).

Note that for the example given above, a simple translation like `temp=x; x=y; y=temp;` which uses a temporary variable will also suffice, and this is the approach

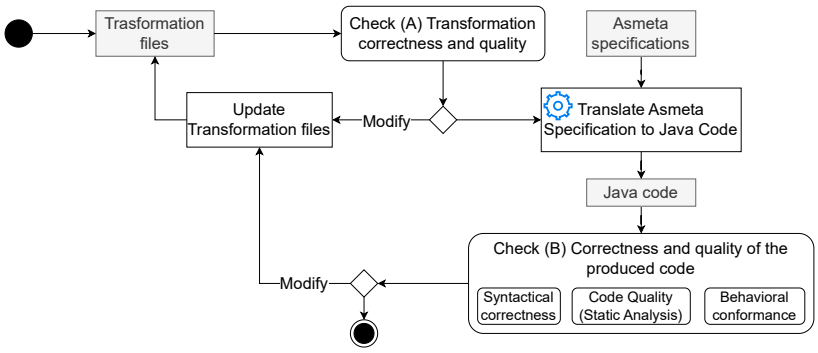

Fig. 6: Quality assurance process followed during the `Asmeta2Java` development

proposed by other research works, such as Dalvandi et al. [14]. However, in an `Asmeta` specification, we cannot lose the current value of controlled functions that may be used in other rules.

Non-determinism. Non-determinism in `Asmeta` is implemented by the ChooseRule, which allows for picking a random element out of a domain (i.e., the set of terms of the same nature). This random choice is performed by generating a random number, which corresponds to the index of the element, to be picked inside the domain. When translating `Asmeta` specifications into Java code, we keep track of all objects created in a domain through a list. Therefore, we implement non-determinism in Java by generating a random integer index in the range between 0 and the number of terms belonging to the domain of interest. An example is shown in the coffee vending machine translation (see Listing 1 at line 27), where a drink is randomly chosen from the available products list by excluding the milk. In the generated source code shown in Listing 2, a list of selectable drinks is populated (those belonging to the `Product` domain except milk) at line 83, then a random index is generated at line 86 and, finally, an element is taken from the list of selectable products (see line 87).

4 Validation of the transformation

We have devised a process to validate the transformations presented so far, depicted in Fig. 6. The process is iterative; initially, we have modified several times the transformation code in order to improve its quality (A). Then, we have applied the transformation to several `Asmeta` specifications, and we have analyzed the correctness[2] and quality of produced code (B). Both the results obtained from steps (A) and (B) have been used to improve the quality and correctness of the code used for the transformation of `Asmeta` specification to Java code. In the following, we explain in detail steps (A) and (B).

[2] With *correctness* we indicate both syntactical correctness and behavioral conformance of the produced code w.r.t. the `Asmeta` specification.

(A) Transformation correctness and quality. First, we check that the transformation programs written in `Xtend` do not contain errors and satisfy the quality attributes themselves, and they can successfully be compiled into their corresponding Java code. Second, we check that the code of the transformations in Java does not contain errors and passes the quality checks the designer desires to establish. In our case, to check the quality of the Java source code, we used the SonarLint tool[3]. SonarLint is a static analyzer and is capable of detecting possible bugs, code smells, and vulnerabilities. It contains around 5000 rules that check the quality of the code in terms of reliability, maintainability, readability, and security. With an iterative process, such as that represented in Fig. 6, we have improved our transformation code and fixed all the 57 warnings signaled by SonarLint (see Tab. 5).

The checks carried out in (A) refer to the quality of the programs performing the transformations. Although they play a very important role, there is no guarantee that a *good* transformation code according to our criteria (A) will produce a *good* code, since the produced code may contain errors or other issues. For this reason, we have decided to check the correctness and quality of the code *produced* by the transformation (B). An error or some low-quality warning of the produced code must not prompt the modification of the generated code itself; instead, it requires a modification of the transformation code that caused the insertion of that error or the low-quality code.

(B) Correctness and quality of the produced code. We can distinguish three main activities we can perform to check the generated code:

(B.1) Syntactical correctness: checks that the generated code compiles.
(B.2) Code quality: applies quality checks to the produced code.
(B.3) Behavioral conformance: aims to check whether the `Asmeta` specification and the Java code implement the same behavior.

While for (B.1) a Java compiler suffices, for (B.2) we have used SonarLint as in (A). When we initially checked the generated code, we found $1,809$ warnings (see Tab. 5). Thus, we have modified the transformation code to fix most of them and reduced the warnings on the generated code to 560. On the one hand, this analysis has allowed us to better implement the principles that have guided our implementation and are described in Sect. 3.1. For example, we have removed many useless imports that were always added but used only in particular situations (such as those of the classes used for the tuples or structured domains). However, on the other hand, given the principles we have described in Sect. 3.1, we were not able to remove all warnings. For example, to maintain traceability, we decided to keep the classes with the same name as the ASM, and, when these start with a lowercase character, the corresponding classes start with a lowercase character too, leading to warnings signaled by Sonarlint. Similarly, the same warnings are signaled by Sonarlint for the name of the fields in the classes: whenever the functions were declared with a leading uppercase character, we decided to keep the same initial character for the fields as well. A last example concerns the name of the

[3] `https://www.sonarsource.com/products/sonarlint/`

Warning Severity	A - Transformation code Before	After	B - Generated code Before	After
Critical	4	0	106	22
Medium	31	0	51	51
Low	22	0	1652	456
Total	57	0	1809	560

Table 5: Quality warnings on transformation and generated code

rules. In `Asmeta` rules are required to start with `r_`, and, to guarantee traceability in the Java code, the rules are translated into methods having the same name. However, the Java guidelines suggest not to have an underscore in the method's name.

In this paper, we skip the activities to perform behavioral conformance analysis (B.3), which will be presented as future work (see Sect. 6). To guarantee the maximization of the results obtained in (B), the `Asmeta` specifications used as input for the validation process must cover most of the constructs available in `Asmeta`. To do this, we have measured, by using Eclemma[4], the coverage of the `Asmeta` constructs when translating all the `Asmeta` specifications available in our considered examples. The result obtained is that 89% of the `Asmeta` constructs handled by `Asmeta2Java` are covered by our experiments.

5 Related work

Automatic code generation from formal specifications is available as a part of tool support for several formal methods, e.g., MATLAB/Simulink[5] provides this feature as a commercial off-the-shelf solution. However, other free tools are available for other formal methods. For B users [1], executable C, C++, Java, and Ada code can be generated from formal specifications by the Atelier B platform[6], which is free in its Community Edition. Additionally, for the Event-B formal method, the EventB2Java tool integrated into the Rodin platform allows developers to generate executable code [12]. The generation of Java code is also provided by other non-formal approaches. For example, executable Java code can be generated starting from UML class and sequence diagrams [18], or state-charts [20]. However, the advantages of these approaches are limited only to the reuse of (non-formal) models, but no guarantee on safety properties or system behavior is given.

For what concerns Abstract State Machines, the `Asmeta` framework already provides the `Asm2C++` translator [9] which, starting from `Asmeta` specifications, produces executable C++ code, possibly also providing the source code for its integration with

[4] `https://www.eclemma.org/`
[5] `https://www.mathworks.com/products/simulink.html`
[6] `https://www.atelierb.eu/en/`

Arduino. The approach presented in [9] is very similar to that we present in this paper. However, if developers work with Java code, `Asmeta2Java`, as presented in this paper, is preferable since no manual intervention is needed to integrate the source code produced by the `Asmeta` environment while deploying the final system. Another preliminary work that translates CASM specification to C code is presented in [21,19]. In these works, the authors present an optimized compiler for ASM specifications that could be easily retargeted to different programming languages and hardware target domains (but currently only C/C++ code is supported). They present a compilation scheme and an implementation of a runtime system supporting inconsistent updates checking and efficient execution of ASMs. Another approach supported by a tool called JASMine, is presented in [15]. The design goal of JASMine is to allow interaction between ASM and Java, rather than full integration. The goal of their work is not the translation, but rather to access Java objects and classes from inside an ASM specification. The approach presented in this last work differs from that we propose, especially for what concerns the handling of updates. In `Asmeta2Java`, we build the next update set for each function and we apply it when the step is executed, by keeping two copies of the ASM state in Java. In JASMine, instead, the authors collect all the *deferred* updates and apply them after checking their consistency.

6 Future work

According to the best practices of MDE, the implementation of a system should be obtained from its model through a systematic model-to-code transformation, as we have presented in this paper. Furthermore, following the same approach, unit tests should be obtained automatically from abstract tests. With `Asmeta`, users can validate their specifications by means of `Avalla` scenarios, which can be considered abstract tests that may be concretized to obtain executable unit tests. Thus, as previously done for C++ [6], we plan to enhance the `Asmeta2Java` tool by adding an automatic translation of `Avalla` scenarios into jUnit test cases. Specifically targeting the behavior of the system, these test cases can also be used to verify the *semantical* correctness of the code produced by `Asmeta2Java` (see Sect. 4).

When it comes to embedding the modeled specification in the final executable, possibly having a GUI, two different approaches may be used. On the one hand, the formal specification can be executed by embedding the `Asmeta` simulator in the final artifact, as shown in [5]. On the other hand, as we have presented in this paper, the executable source code can be generated from the formal specification and, then, directly used in the final executable. In this last case, having a way to visualize the behavior of the obtained source code is advisable. Thus, we plan to introduce a procedure for the automatic generation of a UI, allowing the user to interact with the generated Java code in a simplified way.

7 Conclusion

In this paper, we have presented the translation of formal models into Java code to automatically obtain the system implementation. This process follows the MDE paradigm:

the source code is obtained from Asmeta specifications by applying a set of M2T transformations. Furthermore, we have presented the principles guiding our approach and introduced activities for the validation of the transformation to guarantee the correctness of the process. The Java code generation process starts from the ASM specification, and using Asmeta2Java, the Java code of the model is automatically generated. We have shown how a simple case study of a coffee vending machine is translated, and then, for each Asmeta construct, we have displayed the translation we have adopted. We have dealt with two characteristics of ASM models, namely the parallelism and the non-determinism, that are non-trivial in Java, and we have found the solution to maintaining in the Java code the same ASM behavior. The application of the activities presented in this paper guarantees that, starting from a validated and verified Asmeta specification, users can automatically generate a code implementation compliant with the validated and verified behavior. As future work, we are working on further extending the functionalities offered by the Asmeta2Java tool, and allowing him to generate jUnit test cases, which can be useful both to validate the translation process and to perform regression testing when manual intervention on the Java code is needed.

References

1. Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.
2. Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. The ASMETA approach to safety assurance of software systems. In *Logic, Computation and Rigorous Methods*, pages 215–238. Springer, 2021.
3. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the Second International Conference on Abstract State Machines, Alloy, B and Z*, ABZ'10, pages 61–74, Berlin, Heidelberg, 2010. Springer-Verlag.
4. Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
5. Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. *formal MVC: A Pattern for the Integration of ASM Specifications in UI Development*, page 340–357. Springer Nature Switzerland, 2023.
6. S. Bonfanti, A. Gargantini, and A. Mashkoor. Generation of C++ Unit Tests from Abstract State Machines Specifications. In IEEE Computer Society, editor, *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 185–193, April 2018.
7. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. *AsmetaA: Animator for Abstract State Machines*, page 369–373. Springer International Publishing, 2018.
8. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Validation of Transformation from Abstract State Machine Models to C++ Code. In *ICTSS 2018: Testing Software and Systems*, pages 17–32. Springer International Publishing, 2018.
9. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Design and validation of a C++ code generator from Abstract State Machines specifications. *Journal of Software: Evolution and Process*, 32(2), November 2019.
10. E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., 2003.

11. Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. *A Scenario-Based Validation Language for ASMs*, page 71–84. Springer Berlin Heidelberg.
12. Néstor Cataño and Víctor Rivera. EventB2Java: A Code Generator for Event-B. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods: 8th International Symposium, NFM 2016*, pages 166–171, 2016.
13. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
14. Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi. Verifiable code generation from scheduled Event-B models. In *Lecture Notes in Computer Science*, pages 234–248. Springer International Publishing, 2018.
15. Vincenzo Gervasi and Roozbeh Farahbod. *JASMine: Accessing Java Code from CoreASM*, page 170–186. Springer Berlin Heidelberg, 2009.
16. Yuri Gurevich. *Evolving algebras 1993: Lipari guide*, page 9–36. Oxford University Press, Inc., USA, 1995.
17. John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE11. ACM, May 2011.
18. Preyanoot Kluisritrakul and Yachai Limpiyakorn. *Generation of Java Code from UML Sequence and Class Diagrams*, pages 1117–1125. Springer Singapore, 2016.
19. Roland Lezuo, Philipp Paulweber, and Andreas Krall. Casm: optimized compilation of abstract state machines. *ACM SIGPLAN Notices*, 49(5):13–22, May 2014.
20. Iftikhar Azim Niaz and Jiro Tanaka. Code generation from UML statecharts. In *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*, pages 315–321, dec 2003.
21. Philipp Paulweber and Uwe Zdun. A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications. In *5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, ABZ 2016*, Lecture Notes in Computer Science 9675, pages 250–255. Springer, 2016.
22. Joachim Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.