Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

TECHNOLOGICAL

Design and validation of a C++ code generator from Abstract State Machines specifications

Silvia Bonfanti*[1] | Angelo Gargantini[1] | Atif Mashkoor[2,3]

[1]Department of Management, Information and Production Engineering, University of Bergamo, Bergamo, Italy
[2], Software Competence Center Hagenberg GmbH, Hagenberg, Austria
[3]Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria

Correspondence
*Silvia Bonfanti
Email: silvia.bonfanti@unibg.it

Present Address
Department of Management, Information and Production Engineering, viale Marconi, 5 - 24044 Dalmine BG -Italy

Summary

According to best practices of model-driven engineering, the implementation of a system should be obtained from its model through a systematic model-to-code transformation. We present in this paper a methodology supported by the `Asm2C++` tool that allows the users to generate C++ code from Abstract State Machine models. Thanks to `Asm2C++`, the implementation is generated in a seamless manner with an assurance of potential bug freeness of the generated code. Following the same approach, model-based testing suggests deriving also (unit) tests from abstract models. We extend the `Asm2C++` tool such that it can automatically produce unit tests for the generated code. Abstract test sequences, either generated randomly or through model checking, are translated to concrete C++ unit tests using the BOOST library. In a similar manner, also scenarios are generated in a Behavior-Driven Development (BDD) approach. To guarantee the correctness of the transformation process, we define a mechanism to test the correctness of the model-to-code transformation with respect to two main criteria: syntactical correctness and semantic correctness, which is based on the definition of conformance between the specification and the code. Using this approach, we have devised a process able to test the generated code by reusing unit tests. The process has been used to validate our model-to-code transformations.

KEYWORDS:
Abstract State Machine, C++, model-driven engineering, unit tests generation, automatic code generation, transformation validation, behavior-driven development

1 | INTRODUCTION

Formal methods are often used to model requirements of a system under construction at the abstract level. Formal requirements models are then amenable to a series of quality assurance activities including dynamic analysis like simulation and static analysis like property verification. However, it may remain unclear how to properly link the final implementation to its abstract specification, since the implementation may contain platform-centric details, be written in a programming language, and running on a specific hardware. Performing a manual transformation of formal models into code increases costs, limits the reuse of a formal specification, is error prone as some faults can be introduced in the code writing process, and can be a barrier for a wider adoption of formal methods. It would be very useful to assist designers also in this very delicate last phase of the development process.

Abstract State Machines (ASMs)[1] is a formal method that proposes a rigorous software and systems development process for seamless transformation of informal requirements into implementation. The ASM method is based on the design concept

of *refinement* that allows to capture all details of a system design by a sequence of refined models till the desired level of abstraction encompassing validation and verification (V&V) activities. The final step of this refinement process consists in manually realizing the implementation – generally code that is compiled and deployed on the real system. Having an automatic translator from ASM to code would be extremely useful for the motivations mentioned above, but several aspects that make ASMs very powerful when modeling (like parallel execution of rules, non determinism, and abstract domains), pose a challenge when such constructs have to be translated to a general purpose programming language.

In this article, we present a methodology supported by the `Asm2C++` tool[†] (part of the `Asmeta` framework[2]) that is able to automatically generate the desired C++ source code from ASM specifications. Compared to the previous work[3], where we have presented a partial translation from UASM to C++, in this work we present the translation of the `AsmetaL` language (used by the `Asmeta` framework) to C++. In our approach, the designer starts at an abstract level to specify the desired system in `Asmeta`, then s/he performs a series of refinement steps and V&V activities, and concludes the process by obtaining a detailed `Asmeta` model which can be automatically translated into C++ code using `Asm2C++`. Besides transforming `Asmeta` specifications into source code, we also generate unit test cases and scenarios[4,5]. The goal is to provide the user not only with the code but also with all the tests that are generally written by hand and that can be very useful especially during the regression testing process.

The approach presented in this paper is an application of the model-driven engineering (MDE) paradigm to ASMs: requirements models are platform-independent, there is a clear distinction between platform-specific details, original user and system requirements, the code generation process is seamless and automatic, and last but not least, the rigorous quality and correctness assurance is embedded within the development process. For this reason, we have spent a considerable effort not only in defining the suitable transformations, but also in building a framework for the validation of such transformations. Moreover, it is very important that also model-to-code transformations are tested and validated in an automatic way[6].

The article is structured as follows. In Section 2, we introduce `Asmeta` and MDE. The generation of C++ code is explained in Section 4, where we show how `Asmeta` constructs (like functions and rules) and concepts (like parallelism and nondeterminism) are translated into C++. We report how `Asm2C++` can target a hardware platform – namely Arduino (https://www.arduino.cc/) – and we specify where platform-specific information are added to the source code. Our code generator is able to produce also unit tests and scenarios in a behavior-driven development (BDD) style. Section 5 presents our proposal for the validation of the transformation by means of tests. We employ an *indirect* testing approach on the translation where we do not check that the resulting code is exactly as excepted, rather we (automatically) check that the behavior of the code is correct. Section 6 presents the related work. The article is concluded in Section 7.

2 | BACKGROUND

2.1 | ASM method

Abstract State Machines (ASMs) are an extension of Finite State Machines (FSMs), where unstructured control states are replaced by states with arbitrary complex data. ASM *states* are mathematical structures, i.e., domains of objects with functions and predicates defined on them. An ASM *location* – defined as the pair (*function-name*, *list-of-parameter-values*) – represents the abstract ASM concept of basic object containers. The ordered pair (*location*, *value*) represents a machine memory unit.

Location values are changed by firing *transition rules*. They express the modification of functions interpretation from one state to the next one. Note that the algebra signature is fixed and that functions are total (by interpreting undefined locations $f(x)$ with value *undef*). Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v is its new value. They are the basic units of rule construction. There is a limited but powerful set of *rule constructors* to express: guarded actions, simultaneous parallel actions, sequential actions, nondeterminism, and unrestricted synchronous parallelism.

An ASM *computation* or *run* is, therefore, defined as a finite or infinite sequence of states $s_1, s_2, \ldots, s_n, \ldots$ of the machine. s_1 is an initial state and each s_{i+1} is obtained from s_i by firing the unique *main rule*, which could fire other transition rules (see Figure 1). All the operations are executed in parallel in the same transition, and the functions are modified during the update set. The update set consists in change the value of functions in state s_{i+1} with the new value computed in the state s_i after the main rule execution.

During a machine computation, not all the locations can be updated. Functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished

[†]The tool and the experiments are available on the `Asmeta` repository (http://asmeta.sourceforge.net/download/asm2c++.html)

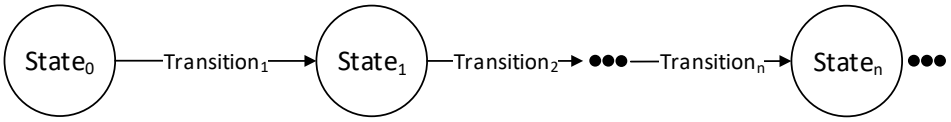

FIGURE 1 An ASM run with a sequence of states and state-transitions (steps)

between *monitored* (only read by the machine and modified by the environment) and *controlled* (read in the current state and updated by the machine in the next state). A further classification is between *basic* and *derived* functions, i.e., those coming with a specification or computation mechanism given in terms of other functions.

An ASM can be *nondeterministic* due to the presence of monitored functions (*external* nondeterminism) or choose rules (*internal* nondeterminism). Our code translation supports both types of nondeterminism, however, testing the generated code in the presence of internal nondeterminism is challenging as explained in Section 5.4.

To better understand the translation of ASMs written in `AsmetaL`, here we show the structure of an ASM model. An ASM specification contains a single ASM definition which is structured into four sections: header, body, main rule and initialization. There exist two types of ASM files: **asm** which contains a complete ASM and must contain the main rule and **asmmodule** which contains an ASM without the main rule (thus, it is not executable by itself).

The header contains the imported/exported functions/domains/rules from other ASMs/ASMs module and the signature. The signature contains the declarations of domains and functions used in the model. The body holds the definition section which consists of three elements: domains definition which defines using Terms the elements of domains defined in the signature, functions definition which defines the functions using `AsmetaL` Terms, and rules declaration and definition. The main rule defines the transition rule invoked at the beginning of each step and identifies itself from the others using the keyword "main rule". When the ASM computation starts, the initial state s_1 is automatically chosen by the simulator unless the developer specifies in the initialization section the values of dynamic domains and functions. Code 1 shows the structure of an ASM using `AsmetaL` language. The complete `AsmetaL` grammar can be found at http://asmeta.sourceforge.net.

In this article, we show the application of the process on an example, a simple counter, whose `AsmetaL` specification is shown in Code 2. The counter increments its value when a signal is true. There are two functions, the monitored function `signal` which reads the environment (normally, it acquires the user input) and the controlled function `counter` which is the counter value. Initially the counter is set to 0, and as long as the user inserts "true" (see main rule) the counter is incremented, otherwise the counter keeps its value.

```
asm asmname
  //import
  //export

  signature:
    //domains declarations
    //functions declarations

  definitions:
    //domains definition
    //functions definition
    //rule definition

  main rule r_Main = //rule definition

  default init i:
    //functions initialization
    //dynamic domain initialization
```

Code 1 AsmetaL structure

```
asm Counter

import StandardLibrary

signature:

  monitored signal: Boolean
  controlled counter: Integer

definitions:

  main rule r_count =
    if signal then
      counter := (counter+1)
    endif

default init s1:
  function counter = 0
```

Code 2 Counter ASM model

2.2 | Asmeta framework

The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Figure 2 shows the development process based on ASMs.

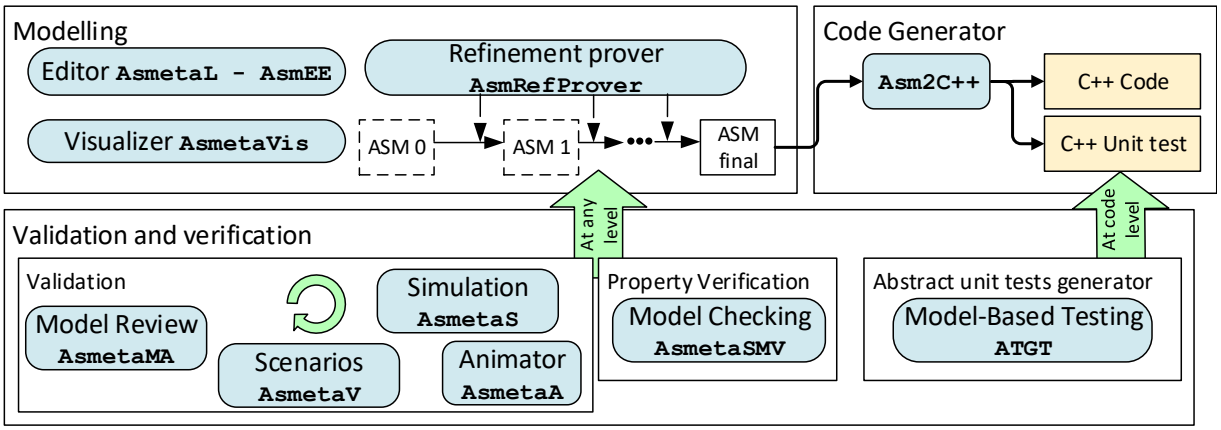

FIGURE 2 The ASM development process powered by the Asmeta framework

The process is supported by the Asmeta (ASM mETAmodeling) framework[‡2] which provides a set of tools to help the developer in various activities:

- **modeling**: the system is modeled using the AsmetaL language . During the modeling the user is supported by the editor AsmEE and by AsmetaVis, the ASMs visualizer which transforms the textual model into a graphical representation. The user can directly define the last ASM model or s/he can reach it through refinement. The refinement process is adopted in case the model is complex. In this case, the designer can start from the first model (also called the ground model) and can refine it through the refinement steps by adding details to the behavior of the ASM. The AsmRefProver tool ensures whether the current ASM model is a correct refinement of the previous ASM model. It is based on Satisfiability Modulo Theories (SMT) solvers that are used to automatically prove correctness of model refinement.

- **validation**: the process is supported by the model simulator AsmetaS, the animator AsmetaA, the scenarios executor AsmetaV, and the model reviewer AsmetaMA. The simulator AsmetaS allows to perform two types of simulation: interactive simulation and random simulation. The difference between the two types of simulation is the way in which the monitored functions are chosen. During interactive simulation the user inserts the value of functions, while in random simulation the tool randomly chooses the value of functions among those available. AsmetaA allows the same operation of AsmetaS, but the states are shown using tables which make the readability of the state easy. AsmetaV executes scenarios written using the AvalLa language. Each scenario contains the expected system behavior and the tool checks whether the machine runs correctly. The model reviewer AsmetaMA performs static analysis. It determines whether a model has sufficient quality attributes (e.g., minimality – the specification does not contain elements defined or declared in the model but never used, completeness – requires that every behavior of the system is explicitly modeled, and consistency – guarantees that locations are never simultaneously updated to different values).

- **verification**: the properties derived from the requirements document are verified to check whether the behavior of the model complies with the intended one. The AsmetaSMV tool supports this process.

- **testing**: the tool ATGT generates abstract unit tests starting from the ASM specification by exploiting the counter example generation of a model checker.

- **code generation**: given the final ASM specification, the Asm2C++ automatically translates it into C++ code. Moreover, the abstract tests, generated by the ATGT tool, are translated into C++ unit tests. The Asm2C++ tool is presented in this paper.

[‡]http://asmeta.sourceforge.net/

3 | METHODOLOGY

The approach we present in this paper adheres to MDE principles[7]. In our approach, the modeler produces a chain of models in a stepwise manner. As a last step of this rigorous development process, the implementation is derived from the model by applying suitable transformations. In our approach, like any other typical MDE process, requirements models are platform-independent, there is a clear distinction between platform-specific details and original user and system requirements, and the code generation process is seamless and automatic. The code produced in the final step is, however, platform-specific (we will see how we target the Arduino platform in Sec. 4.3). One additional benefit of our approach is that quality and correctness assurance is already embedded within the rigorous development process. As another additional goal, we aim at producing a code which is *understandable* such that the code instructions can be easily *traced back* to the specification concepts and constructs[8]. Although this may decrease the code efficiency, we believe that it increases the maintainability and the usability of the `Asm2C++` tool.

As shown in Figure 3, the starting point of the transformation is a formal specification written in ASM conforming to the `Asmeta` grammar. After that, the specification is translated into an `Asmeta` model conforming to the `Asmeta` meta-model using a text-to-model (T2M) transformation. The T2M transformation, also called parsing, is performed by the parser. The parser takes as input a string (for example, the text file for the specification), verifies whether the string conforms to the grammar, and creates the corresponding model. In case the text file does not conform to the grammar, an exception is raised and the parsing process is stopped.

The `Asmeta` model is later on translated into the C++ code through a model-to-text (M2T) transformation (also known as model-to-code transformation[9]). In this transformation, a model is translated into a textual representation, i.e., source code of a specific language. There are different technologies to support the M2T transformation process, the most common are:

- **Xpand:**[§] It is based on 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, and a basic editor. The main disadvantage of Xpand is that the output generation is slow, the code is not maintained, and it is not well tested.
- **Xtext:**[¶] It is a successor of Xpand based on the Java dialect called Xtend. It has the following advantages: it is faster than Xpand, it is debuggable, and it has its own integrated development environment (IDE).

Consequently, we exploit Xtext in order to transform the `Asmeta` model into the C++ code in our approach.

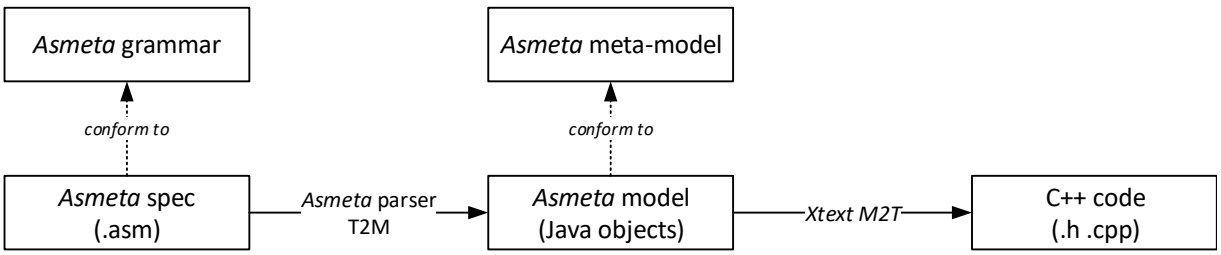

FIGURE 3 M2T Transformation: from ASM to the C++ code

4 | GENERATION OF C++ CODE

The tool `Asm2C++` is based on Xtext[#], a framework for the development of domain-specific languages, which provides facilities for parsing and code generation and is fully compatible with EMF. The code generator has been developed as a M2T transformation.

[§] https://eclipse.org/modeling/m2t/?project=xpand
[¶] http://www.eclipse.org/Xtext/
[#] https://www.eclipse.org/Xtext/

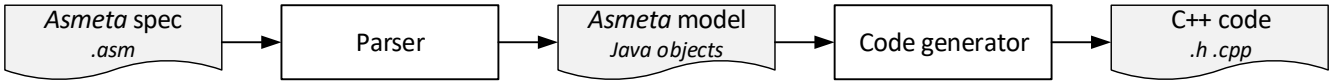

FIGURE 4 Code generation process

4.1 | Generation process of C++ code

The generation process is designed as described in Figure 4. The steps implementing the translation are parsing and code generation.

Parsing

The generation process, as described in Figure 4, starts from the `Asmeta` specification. The specification is written in a textual notation conforming to the `Asmeta` grammar and contains all the information about the model. The `Asmeta` parser takes as input the `Asmeta` specification and creates the abstract syntax tree of the model as Java Objects. This is easier to process as compared to the textual representation.

Code generation

The code generation step (see Figure 4) generates C++ code starting from the `Asmeta` model adopting M2T transformation.The output is the C++ code that can be compiled and executed by any compiler that supports the C++ 11 (or more recent) standard. The C++ code is composed by the header (.h) and the source (.cpp) files.

4.2 | Translation from `Asmeta` to the C++ code

The `Asm2C++` tool transforms an `Asmeta` specification to a C++ code. We have decided to translate an ASM to a C++ class and split the class into a header (*.h*) and a source (*.cpp*) file. The header file (see Code 3) contains the interface of the source file. Given an ASM model, the interface corresponds to the translation of the ASM signature: domains declaration, domains definition, functions declaration, and rules declaration. The rules implementation, the functions/domains initialization, and the functions definitions are contained in the source file (see Code 4). In the following, we explain how the semantics of ASM steps are translated in C++ in Sect. 4.2.1, while in Sect. 4.2.2 we explain how the state (domains, functions, and terms) is translated. In the following, we refer to the transformation from ASM to C++ as τ.

4.2.1 | ASM state evolution in C++

One of the main difference between ASM and C++ is the way the current state is changed, i.e. the execution of a machine in ASM or of a program in C++. In ASM, as explained before, the execution consists in a sequence of steps where each step executes the main rule and perform the update set. On the other hand, in C++ the execution corresponds to a sequence of instructions that can be repeatedly called. The simulation of an ASM step has been implemented using two suitable methods in C++: the `mainRule()` method and the `fireUpdateSet()` method. The `mainRule()` method corresponds to the translation of the main rule into C++ code, while the `fireUpdateSet()` method updates the locations to the next state values.

ASM has two semantics concepts that do not have the direct implementation in C++. The first semantic problem is the parallel execution of all rules while the other problem is the nondeterminism. The solutions adopted are explained below. Furthermore, in this section we introduce the translation of ASM rules to corresponding C++ instructions.

Parallelism

Parallelism is a fundamental point of ASMs because the operations performed in the same transition are done in parallel. Our goal is to provide a translator that generates C++ code usable both on platforms which do not support the use of C++ parallel programming, like Arduino, and on platforms which support the use of C++ parallelism. Nevertheless, it is still possible to run ASMs in a parallel-like way by showing the same copy of the current state to a sequentially-executed thread following the approach proposed by others, including J. Schmid[10]. 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 machine state (represented by controlled functions) is modified only during the main rule execution. To simulate parallel execution, the controlled part of the state is

```
#ifndef  asmSpecification_H
#define  asmSpecification_H

#include ... // include libraries

// Domain declaration
namespace  asmSpecificationnamespace{
  // enumerative domain
  enum domainName {value0, value1, ...};
  // concrete domain
  typedef  domainType  domainName;
}
using namespace asmSpecificationnamespace;
class asmSpecification{
  // Domain containers declaration:
  //concrete domain and enumerative domain
  const std::set<domainName>
  domainName_elems;
  public:
  // Function declaration
  domainName functionName[2];
  // controlled function
  domainName functionName;
  // monitored function
  // Rule declaration
  void ruleName (parameters);
  asmSpecification();
  void mainRule();
  void fireUpdateSet();
};

#endif
```

Code 3 .h code

```
//asmSpecification.cpp automatically generated
#include ``asmSpecification.h"
using namespace asmSpecificationnamespace;

// Conversion of ASM rules in C++ functions
void asmSpecification::ruleName (parameters){
  // implementation
}

void asmSpecification::mainRule(){
  // implementation
}

// Function and domain initialization
asmSpecification::asmSpecification():

// Static domain initialization
domainName_elems(value0,value1 ,...) ,
{
//Function initialization
functionName[0] = functionName[1] = value;
}

// Apply the update set
void  asmSpecification::fireUpdateSet(){
  functionName[0] = functionName[1];
}
```

Code 4 .cpp code

duplicated: the present state and the future state. Modification made on controlled functions will affect only the future state, while status readings will refer to the current state. The modification made to controlled functions will take place only when the execution of the main rule is finished, this means in `fireUpdateSet()` function. This approach guarantees the proper evolving of the machine state, even though it is not a true parallelism. Table 1 shows a simple example of parallelism, it swaps the values of variables `x` and `y`. If we translated this example in C++ as a simple sequence `x=y; y=x;`, the result is not the same as in ASM. Both variables `x` and `y` would contain the value of `y`. For this reason, we introduce an array of two values for each controlled function; the first value corresponds to the value of the function in the current state of the ASM execution and the second value corresponds to the value of the function in the next state of the ASM execution. In this way, we translate an assignment in C++ following this rule: we use the next state (the second element of the array) to the left side of the assignment; we use the current state (the first element of the array) to the right side of the assignment; after the execution of the rules in the current state (the `mainRule()` function in C++) we assign all the values of the next state to the current state. 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 ASM model guarantee that the inconsistent updates will not occur.

Asmeta	Main in C++	fireUpdateSet in C++

`controlled x: Integer`
`controlled y: Integer`
`rule parRule =`
`par`
`x:=y`
`y:=x`
`endpar`
	`int x[2];`
	`int y[2];`
	`void parRule(){`
	`x[1]=y[0];`
	`y[1]=x[0];`
	`}`
		`void fireUpdateSet(){`
		`x[0]=x[1];`
		`y[0]=y[1];`
		`}`

TABLE 1 Parallelism: translation in C++

Note that for the example given above, also a simple translation like `temp=x; x=y; y=temp;` which uses a temporary variable will suffice, and this is the approach proposed for example by Dalvandi et al.[11]. However in the ASM we cannot loose the current value of controlled functions which may be used in other rules.

Nondeterminism

Nondeterminism in `Asmeta` is implemented in the ChooseRule where a random element is picked out from a certain domain (set of terms of the same nature). This random choice is performed by generating a random index, number that corresponds to the index of the element to be picked inside the domain. C++ provides functionality for pseudo-random number generation that generates an integral number in the range between 0 and the number of terms. An example of nondeterminism using the choose rule is shown in Table 2.

Asmeta	**domain** D2 **subsetof** Integer **domain** D2 = {5,9,12} **rule** chooseRule = **choose** x **in** D2 **with** x>7 **do** *ruleChoose* **ifnone** *ruleIfnone*

C++	`typedef int D2;`
	`std::set<D2> D2_elems = {5,9,12};`
	`void chooseRule(){`
	`  std::vector<decltype(D2_elems)::value_type const*> point0;`
	`  for(auto const& x : D2_elems)`
	`  if(x>7) point0.push_back(&x);`
	`  int rndm = rand % (point0.size());`
	`  auto x = *point0[rndm];`
	`  if(point0.size()>0){`
	`  ruleChoose();`
	`  } else{`
	`  ruleIfnone();`
	`  }`
	`}`

TABLE 2 Nondeterminism: translation in C++

Transition rule

Transition rules are translated into class functions where every rule is implemented using the C++ basic constructs. The evaluation strategy adopted in ASMs is the pass-by-name, while in C++ the strategy is pass-by-value. For now, to guarantee the consistency, we translate only rule calls taking as parameters location variables that are not updated in the rule. The translation of basic rules in C++ is straightforward most of times. For example, ConditionalRule and CaseRule are implemented respectively with the if-else and case-switch construct. For the rules that do not have an equivalent construct in C++, some translation blocks have been defined. For example, the ForallRule is a high level construct defined by `Asmeta` that executes a rule in parallel for each element of one (or more) EnumerableTerm that satisfies a given condition. The translation of rules into C++ is shown in Table 3.

4.2.2 | Functions, Domains, and Terms

As explained in the next paragraph, functions are translated as members of the class whose type depends on the corresponding domain. Domains are translated using those available in C++ and sometimes may happen that the correspondence is not exact as explained in the domain definition paragraph. Furthermore, terms are translated with C++ operators.

Function definition

To declare an ASM function it is necessary to specify the name, the domain (optional field), and the codomain of the function. Moreover, the function name must be preceded by one of the keywords static, dynamic or derived, depending on its kind. Dynamic functions are further classified in monitored, controlled and out.

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

Conditional	`if` $cond$ `then`
	R_{then}
	`[else`
	R_{else}`]`
	`endif`
	$cond$: term representing a boolean condition
	R_{then}, R_{else}: transition rules
		`if (`$\tau(cond)$`){`
		$\tau(R_{then})$`;`
		`[}else {`
		$\tau(R_{else})$`;]`
		`}`
		$\tau(cond)$: conditional term translation
		$\tau(R_{then})$, $\tau(R_{else})$: rules translation

Case	`switch` t
	`case` t_1 `:` R_1
	`...`
	`case` t_k `:` R_k
	`[otherwise`
	R_{k+1}`]`
	`endswitch`
	t_1,...t_k: terms
	R_1, ..., R_k, R_{k+1}: transition rules
		`if (`$\tau(t)$`==`$\tau(t_1)$`) {`
		$\tau(R_1)$`;`
		`...`
		`} else if (`$\tau(t)$`==`$\tau(t_k)$`) {`
		$\tau(R_k)$`;`
		`} else {`
		$\tau(R_{k+1})$`);`
		`endif`
		$\tau(t)$, $\tau(t_1)$, ..., $\tau(t_k)$: terms translation
		$\tau(R_1)$, ..., $\tau(R_k)$, $\tau(R_{k+1})$: rules translation

ForAll	`forall` v_1 `in` D_1`, ..., `v_k `in` D_k
	`with` $G_{v_1,...,v_k}$ `do` $R_{v_1,...,v_k}$
	v_1,...v_k: variables
	D_1,...D_k: domains where v_1,...v_k take their values
	$G_{v_1,...,v_k}$: term representing a boolean condition over v_1,...v_k
	$R_{v_1,...,v_k}$: transition rule which contains occurrences of variables v_1,...v_k
		`for(auto` $\tau(v_1)$`: `$\tau(D_1)$`) {`
		`...`
		`for(auto` $\tau(v_k)$`: `$\tau(D_k)$`) {`
		`if(`$\tau(G_{v_1,...,v_k})$`) {`
		$\tau(R_{v_1,...,v_k})$`;`
		`}...}}`
		$\tau(v_1)$,..., $\tau(v_1)$: variables translation
		$\tau(D_1)$, ..., $\tau(D_k)$: domains translation
		$\tau(G_{v_1,...,v_k})$: term translation
		$\tau(R_{v_1,...,v_k})$: rule translation

Extend	`extend` D `with` v_1`, ..., `v_k `do` R D: abstract domain to be extended v_1,...v_k: variables which are bound to the new elements imported in D R: transition rule	D_{elems}`.insert(`$\tau(v_1)$`, ..., `$\tau(v_k)$`);` $\tau(R)$`;` D_{elems}: variable containing all elements of domain D $\tau(v_1)$, ..., $\tau(v_k)$: variables translation $\tau(R)$: rule translation

While	`while` G `do` R
	G: term representing a boolean condition
	R: transition rule
		`while (`$\tau(G)$`){`
		$\tau(R)$`;`
		`}`
		$\tau(G)$: conditional term translation
		$\tau(R)$: rule translation

Call Rule	r `[`t_1`, ..., `t_k`]` r: rule name t_1, ..., t_k: terms representing the arguments	r `(`$\tau(t_1)$`, ..., `$\tau(t_k)$`);` r: method name $\tau(t_1)$, ..., $\tau(t_k)$: terms translation
Skip	`skip`	`;`

Seq	`seq`
	R_1
	...
	R_k
	`endseq`
		`{`
		$\tau(R_1)$`;`
		`...`
		$\tau(R_k)$`;`
		`}`

Let	`let (`v_1`=`t_1`, ..., `v_k`=`t_k`) in` $R_{v_1,...,v_k}$ `endlet`
	v_1,...v_k: variables
	t_1, ..., t_k: terms
	$R_{v_1,...,v_k}$: transition rule which contains occurrences of variables v_1, ..., v_k
		`{`
		`auto (`$\tau(v_1)$ `=` $\tau(t_1)$`);`
		`...`
		`auto (`$\tau(v_k)$ `=` $\tau(t_k)$`);`
		$\tau(R_{v_1,...,v_k})$`;`
		`}`
		$\tau(v_1)$, ..., $\tau(v_k)$: variables translation
		$\tau(t_1)$, ..., $\tau(t_k)$: terms translation
		$\tau(R_{v_1,...,v_k})$: rule translation

TABLE 3 Rules: translation from `Asmeta` to C++

Each ASM function is translated as a members of the class as shown in Table 4. ASM static functions are translated to either constants if the do not have arguments or static methods if they have arguments. Derived functions are always translated as member functions. Dynamic functions are member variables, that in case of functions with a domain, they become C++ `maps`. Controlled functions, for the reason explained before, need to be translated to an array of two elements.

Asmeta	Translation in C++
`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	$\tau(C)$ F`;`
`dynamic monitored` F`: `D`-> `C	`map<`$\tau(D)$`,` $\tau(C)$`>` F`;`
`dynamic controlled` F`: `C	$\tau(C)$ F`[2];`
`dynamic controlled` F`: `D`-> `C	`map<`$\tau(D)$`,`$\tau(C)$`>` F`[2];`
`dynamic out` F`:`C	$\tau(C)$ F`;`
`dynamic out` F`: `D`-> `C	`map<`$\tau(D)$`,`$\tau(C)$`>` F`;`
F`: function name` D`: function domain` C`: function codomain` $\tau(C)$`: codomain translation` $\tau(D)$`: domain translation`	

TABLE 4 Function: translation from `Asmeta` to C++

Domain definition

ASM domains are considered mathematical set (or a subset of them) and are implemented using C++ data types. Inevitably, the translation may not preserve the content of some domains which in ASM are infinite or unbounded, while in any implementation domains or types are finite. These implementations of the mathematical sets is named *BasicTD* which includes *Real*, *Integer*, *Natural*, *String*, *Char*, *Boolean* and *Rule*. Besides BasicTDs, there are `EnumTD` (enumerative domains), `ConcreteDomain`, `ProductDomain`, `SequenceDomain`, `PowersetDomain`, `BagDomain`, and `MapDomain`. For the last four domains, there is a porting of the STL library for C++[‖] which provides structured data types. Respectively they are mapped to the STL equivalents `std::list`, `std::set`, `std::multiset`, `std::map`. EnumerativeDomains are directly mapped to C++ with `enums`. ConcreteDomain can be defined as dynamic (new elements can be added to the domain) or static (no new elements are allowed). The machine must keep trace of the elements included in the domain and every time an element is added, the include set must be extended. For this reason, a set of elements is assigned to each ConcreteDomain. ProductDomain is defined as a Cartesian product of two or more domains. It is translated into C++ as a `tuple` of elements, where each element is the domain of the Cartesian product. Patterns of translation are shown in Table 5.

Terms

In `Asmeta` different kinds of terms are implemented, while in imperative languages, such as C++, only few of them are present. The problem is how to translate `Asmeta` terms which are not implemented in C++. For example, consider CaseTerm, that depending on the value assumed by a specific term returns a different value. The first approach has been to translate it into the C++ switch-case construct, but if we apply this translation we violate the one-to-one mapping design goal since it is already used in the translation of CaseRule. A better approach is using lambda functions – an example of function declaration is shown in Code 5. A lambda function is a function that can be written in-line in the source code and called directly without declaring it. The main advantages are the flexibility, any kind of function can be defined, and the one-to-one mapping from `Asmeta` to C++ is guaranteed. The disadvantages of this approach are the worsening of readability and efficiency. The translation of the `Asmeta` terms into C++ is shown in Table 6.

```
[&](){
//place your code here
return <some_value>;
}
```

Code 5 Example of lambda function

[‖]The porting of the STL library is available at https://github.com/rpavlik/StandardCplusplus

Asmeta	Translation in C++
`Natural`	`unsigned int`
`Integer`	`int`
`String`	`String`
`Char`	`char`
`Boolean`	`boolean`
`rule`	Not supported
`Powerset`	`std::set`
`Bag`	`std::multiset`
`Sequence`	`std::list`
`Map`	`std::map`
`enum`	`enum`
`domain` $D1$ `subsetof` t_D $D1$: name of the concrete domain t_D: type-domain which identifies the structure of the elements of $D1$	`typedef` $\tau(t_D)$ $D1$ $D1$: name of the concrete domain $\tau(t_D)$: translation of type-domain which identifies the structure of the elements of $D1$
`Prod(`$D1$`,` $D2$`,` $D3$`)` $D1$, $D2$, $D3$: domains over which the cartesian product is defined	`tuple<`$D1$`,` $D2$`,` $D3$`>` $D1$, $D2$, $D3$: domains over which the cartesian product is defined

`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
	`typedef` $\tau(t_D)$ $D1$`;`
	`std::set<`$D1$`>` $D1$`_elems = {5,9,12}`
	$D1$: name of the concrete domain
	$\tau(t_D)$: translation of type-domain which identifies the structure of the elements of $D1$
	{5,9,12}: domain elements

TABLE 5 Domain definition: translation from `Asmeta` to C++

StructuredTerm

StructuredTerms are identified by comma-separated elements enclosed by brackets inside an `Asmeta` model. In C++, these terms are implemented with the container class of the STL library. Unfortunately, there is no standard notation to refer to them in C++ as for the `Asmeta` language. For this reason, structured terms are implemented with lambda functions that create and return the desired term containing the elements. An example is shown in Code 6 – a *Powerset* of values is assigned to the output function defined in *Powerset(Integer)* domain.

```
out myset: Powerset(Integer)

...

myset := {1, 2, 3}
```

```
std::set<int> myset;
...
myset= [](){
  auto x = {1,2,3};
  std::set<int> s(x.begin(),x.end());
  return s;
}();
```

Code 6 Translation of a structured term

4.2.3 | Translation of the Example in C++

By applying the `Asm2C++` tool, the C++ code is automatically generated, in details the tool generates two files: the header file as shown in Code 8 and the class implementation as shown in Code 9. The header contains the functions and rules definition. The monitored function is translated as a bool variable, while the controlled function counter is translated using an array (see Parallelism paragraph). In our example there is only one rule (that corresponds to the main rule) which is translated as C++ method. The .cpp file contains the `r_count()` method implementation, the counter is incremented of one, and the `fireUpdateSet()` implementation which updates the value of controlled functions.

Term	Asmeta	Translation in C++

Conditional Term	`if` *cond* `then`
	*t*_{then}
	`[else`
	*t*_{else}`]`
	`endif`
	cond: term representing a boolean condition
	*t*_{then}, *t*_{else}: terms
		`if` $\tau(cond)$ `then`
		$\tau(t_{then})$;
		`[else`
		$\tau(t_{else})$;`]`
		`endif`
		$\tau(cond)$: conditional term translation
		$\tau(t_{then})$, $\tau(t_{else})$: terms translation

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

Tuple Term	$(t_1,...t_k)$ t_1,...t_k: terms, that can have a distinct nature	`make_tuple(` $\tau(t)$, $\tau(t_1)$, ..., $\tau(t_k)$`)` $\tau(t)$, $\tau(t_1)$, ..., $\tau(t_k)$: terms translation

ForAll Term	`forall` v_1 `in` D_1, ..., v_k `in` D_k
	`with` $G_{v_1,...,v_k}$
	v_1,...v_k: variables
	D_1,...D_k: domains where v_1,...v_k take their values
	$G_{v_1,...,v_k}$: term representing a boolean condition over v_1,...v_k
		`[&]() {`
		`for(auto` $\tau(v_1)$: $\tau(D_1)$`) {`
		...
		`for(auto` $\tau(v_k)$: $\tau(D_k)$`) {`
		`if(!`$\tau(G_{v_1,...,v_k})$`) {`
		`return false;`
		`}...}}`
		`return true;`
		`}`
		$\tau(v_1)$,..., $\tau(v_1)$: 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` $G_{v_1,...,v_k}$
	v_1,...v_k: variables
	D_1,...D_k: domains where v_1,...v_k take their values
	$G_{v_1,...,v_k}$: term representing a boolean condition over v_1,...v_k
		`[&]() {`
		`for(auto` $\tau(v_1)$: $\tau(D_1)$`) {`
		...
		`for(auto` $\tau(v_k)$: $\tau(D_k)$`) {`
		`if(`$\tau(G_{v_1,...,v_k})$`) {`
		`return true;`
		`}...}}`
		`return false;`
		`}`
		$\tau(v_1)$,..., $\tau(v_1)$: 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`
	v_1,...v_k: variables
	t_1, ..., t_k: terms
	$t_{v_1,...,v_k}$:term containing free occurrences of variables v_1, ..., v_k
		`[&]() {`
		`auto` $(\tau(v_1) = \tau(t_1))$;
		...
		`auto` $(\tau(v_k) = \tau(t_k))$;
		$\tau(t_{v_1,...,v_k})$;
		`}()`
		$\tau(v_1)$, ..., $\tau(v_k)$: variables translation
		$\tau(t_1)$, ..., $\tau(t_k)$: terms translation
		$\tau(t_{v_1,...,v_k})$: terms translation

Size Of Enumerable Term	|`containerSet`| or |`containerBag`| or |`containerMap`| or |`containerList`|	`containerSet.size()` or `containerBag.size()` or `containerMap.size()` or `containerList.size()`

TABLE 6 Terms: translation from `Asmeta` to C++

4.3 | Integration with Arduino

Here, we would like to show how easily we can adapt our code generation process for a specific platform. As a target platform, we have chosen Arduino because it supports C++ language, it is cheap to buy, and it is very accessible. Starting from the generation process as shown in Figure 4, we have added some steps to generate the Arduino project as shown in Figure 5. The new steps (explained below) are the HW configuration and integration, the ASM runner generation and the merging of all generated files.

```
// Counter.h automatically generated
#ifndef COUNTER_H
#define COUNTER_H

#include <string.h>
#include <iostream>

using namespace std;

typedef std::string String;
#define ANY String

// DOMAIN DEFINITIONS
namespace Counternamespace {}

using namespace Counternamespace;
```

Code 7 Counter.h (part 1)

```
// class declaration
class Counter {
  // DOMAIN CONTAINERS
  public:
  // FUNCTIONS
  bool signal;
  int counter[2];
  // RULE DEFINITION
  void r_count();
  Counter();
  void getInputs();
  void setOutputs();
  void fireUpdateSet();
};
#endif
```

Code 8 Counter.h (part 2)

```
// Counter.cpp automatically generated
#include "Counter.h"

using namespace Counternamespace;

//Conversion of ASM rules in C++ methods
void Counter::r_count() {
  if (signal) {
  counter[1] = (counter[0] + 1);
  }
}

// Function and domain initialization
Counter::Counter() {
// Function initialization
  counter[0] = counter[1] = 0;
}

// Apply the update set
void Counter::fireUpdateSet() {
  counter[0] = counter[1];
}
```

Code 9 Counter .cpp

Base HW configuration, HW integration and user change

The *HW configuration* goes in parallel to the code generation phase and aims to produce the HW-specific part of the Arduino project which is missing in the automatic C++ code generator. The C++ code containing the *HW integration* is composed by three functions: the setup function, which contains the hardware initial settings, and the two functions responsible for acquiring inputs and setting outputs. Starting from the `Asmeta` model the tool automatically generates the base *.a2c* configuration file. At this step all functions are defined and they are distinguished between monitored and out functions. Monitored functions are assigned to the inputs while out functions to the outputs. Furthermore, the configuration file contains all hardware settings including Arduino version, which is the machine step delay, the path to the `Asmeta` specification, and the list of bindings. The base configuration file is the minimal skeleton and the user has to complete this file in the *user change* phase. In this phase, the user links monitored and out functions to physical hardware pins and further details can be added. Once the *.a2c* configuration file is complete, the HW integration code is automatically generated.

ASM runner generation

The generated C++ code is a complete translation of the ASM specification. However, to run the ASM on an Arduino board, the *loop()* function (a function continuously called) must be implemented. This part is kept in a separated *.ino* file to split the ASM behavior and the execution policy. The `loop()` function executes iteratively the following functions: `getInputs()` – reads the data from the input devices like sensors; `mainRule()` – contains the behavior described in the `Asmeta` specification; `fireUpdateSet()` – updates the state at the end of each loop; and `setOutputs()` – sets the output values like the current state of light-emitting diode (LED).

Merge

In previous steps, different source codes have been produced. The C++ code containing the translation of the `Asmeta` specification, the C++ code for the HW integration, and the *.ino* code that executes the ASM. The merge process simply links all the previously generated artifacts together to create the Arduino project. This is done using the include directive of C++.

Generation of Arduino code on the Counter example

The Arduino project is generated by running three process in parallel as shown in Figure 5. We have seen in the previous section the automatic generation of the C++ code (.h and .cpp files) starting from the ASM model. At the same time three files are generated: .ino, .a2c and .cpp which contains the hardware integration. The .ino file is generated and no user actions are required. For the .a2c and hw integration .cpp files the user has to map the ASM functions with the hardware to make the system executable.

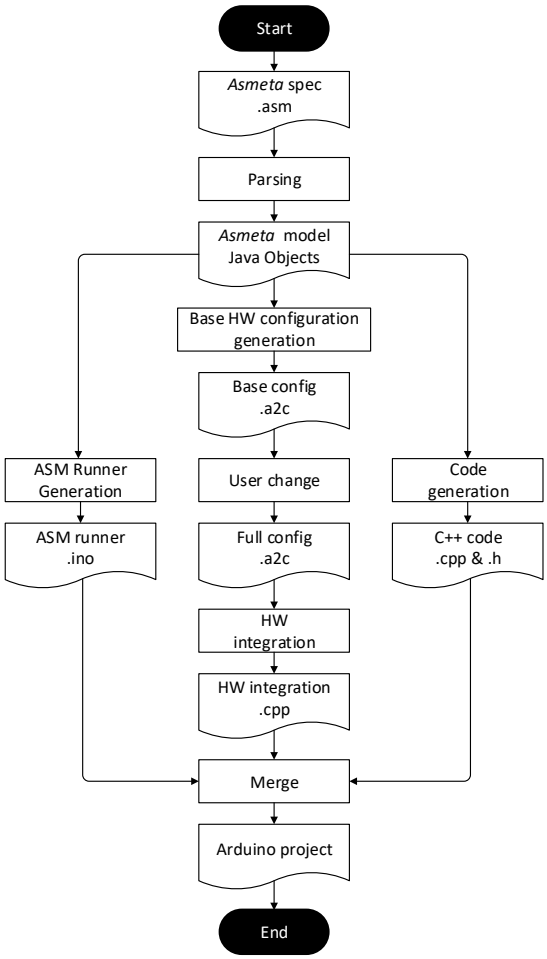

FIGURE 5 Merge process

4.4 | Generation of unit tests

The automatic test generation from models has been advocated by researchers in the area of model-based testing (MBT) for several years. MBT reuses the specification to generate test cases to substitute or at least complement other testing processes, like manual, capture/replay, or script-based ones.

The complete process for MBT applied to ASMs is presented in Figure 6. The user starts with an ASM specification that is already validated and verified. By applying the `Asm2C++` transformation presented in Section 4.1, the C++ code is obtained and compiled. At the same time, the abstract test cases are generated starting from the specification and translated into C++ unit tests. The generation of abstract tests can be done in several ways as explained in Section 4.4.1. The translation of abstract tests to concrete tests is done by the C++ unit tests builder, which follows transformation rules presented in Section 4.4.2. Once the abstract tests are translated, the C++ unit tests are compiled. All the compiled code (system implementation and tests) is linked together by the test executor and the C++ tests can be run. The test executor produces some test results that include possible failures and coverage information, if required. To be more precise, the complete generation of C++ unit tests from ASM specification, requires at least two steps:

- The generation of abstract tests that are sequences of abstract states.
- The translation of abstract tests to concrete tests done by the C++ unit tests builder.

4.4.1 | Generation of abstract tests

We currently support two different ways to generate abstract test sequences from ASM models. The first one is based on the use of the `Asmeta` simulator. The simulator chooses randomly the values of monitored functions (when they are required to perform

```
#include "Counter.h"
void setup(){
}

Counter counter;

void loop(){
  counter.getInputs();
  counter.r_count();
  counter.fireUpdateSet();
  counter.setOutputs();
}
```

Code 10 Counter .ino

```
{
"arduinoVersion": "UNO",
"stepTime": 200,
"bindings": [
  {
    "mode": "USERDEFINED",
    "function": "counter",
    "minval": 0.0,
    "maxval": 1023
  },
  {
    "mode": "USERDEFINED",
    "function": "signal",
    "minval": 0.0,
    "maxval": 1023
  }
]
}
```

Code 11 Counter .a2c

```
#include "Counter.h"
#include <Arduino.h>
void Counter::getInputs(){
  //
  //TODO place here your input
  //binding for function signal
  //
}
void Counter::setOutputs(){
  //
  //TODO place here your input
  //binding for function counter
  //
}
```

Code 12 HW config Counter .cpp

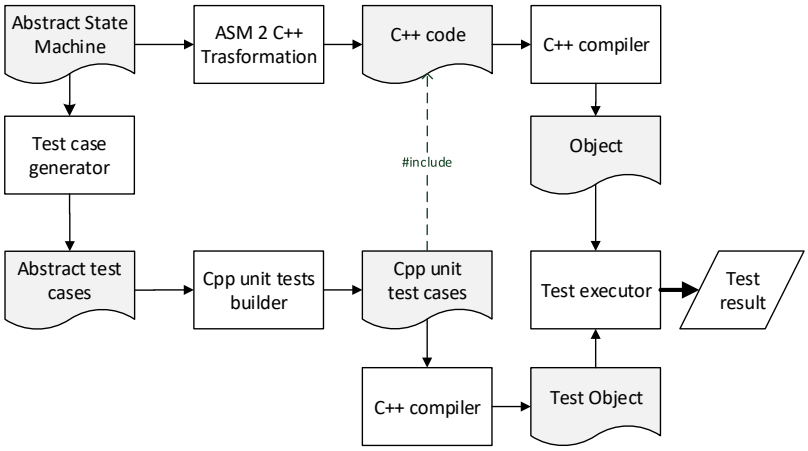

FIGURE 6 Generation of unit tests from ASMs

the execution of one step) and performs a given number of steps of the machine as requested by the tester. There is no guarantee at all that all the specification parts (like rules and conditions) will be covered by test cases. However, in this way, we are able to generate test cases from every ASM, even if it contains complex terms and infinite domains (like integers).

The second approach exploits the counterexample generation of the model checker NuSMV [12]. In this case, the test sequences are generated in order to cover the rules and the guards inside the conditional rules. The tool translates the testing requirements to suitable temporal properties and the counter examples generated by NuSMV are translated to abstract test sequences. In this case, the tool guarantees that the desired coverage is obtained, but only if the ASM is translatable to the language of the model checker and if no state explosion occurs. The use of a model checker generally requires more time, since it must perform the exploration of the whole state space. One could also use techniques for model decomposition [13].

4.4.2 | Translation of abstract tests to concrete tests

Once abstract test cases are generated, the next step is to translate them to concrete test cases in the C++ programming language. The test suite is composed of a set of abstract *test cases* and each test case, in turn, is composed of a sequence of states. The states contain the values of monitored and controlled functions. Each test suite is translated using the Boost Test C++ library (https://www.boost.org/). The boost library provides a set of interfaces to write test programs organized in test suites and test cases. The translation of abstract test cases into concrete test cases is reported in Table 7.

Abstract Test		Concrete Test

Test Suite		`BOOST_AUTO_TEST_SUITE(testSuiteName)`
		`...`
		`BOOST_AUTO_TEST_SUITE_END( )`

Test Case		`BOOST_AUTO_TEST_CASE(testCaseName) {`
		`SUTClass sut;`
		`...`
		`}`

State	Monitored function $m = val$	`sut.m = val;`
	Controlled function $c = val$	`BOOST_CHECK(sut.c[0] == val);`
ASM step		`sut.step();`

TABLE 7 Translation of abstract tests to concrete tests

A test suite is defined by using the *BOOST_AUTO_TEST_SUITE(testSuiteName)* macro, it automatically registers a test suite named *testSuiteName*. A test suite is ended using *BOOST_AUTO_TEST_END()*. Each test suite can contain one or more test cases. A test case is declared using the macro *BOOST_AUTO_TEST_CASE(testCaseName)*. The content of a test case is enclosed by the symbols *{}* and the name is unique.

Inside a test case, first we create an instance *sut* of the class which the ASM is translated to. Then, for each state in the abstract test, we check the value of controlled functions, we set the value of monitored functions and finally we perform an ASM step. The controlled functions are checked using the macro *BOOST_CHECK(controlledFunctionName[0] == value)*, while the *values* to *monitoredFunctionName* are assigned using the assignment operator. The ASM step is performed by calling the main function in C++ (that corresponds to the main rule in ASM) and after that the updateSet is applied in order to obtain the next state.

Test generation on the Counter example

Code 13 shows an example of an automatically generated test suite. This test suite includes four test cases which simulate different execution scenarios. Each test case instantiates a C++ object of the previously generated C++ class and checks whether all initialized functions have the values defined in the ASM initialization section. Once the values are verified, the monitored functions values are set. After that a step of ASM is executed by calling the main method and the update set is applied. Once the update set is performed, the test case starts again from the check of the controlled function. The translation process continues until the list of states belonging to the current test case is entirely translated.

4.5 | Generation of BDD scenarios

The behavior-driven development (BDD) is considered as the evolution and extension of the test-driven development (TDD)[14]. It aims at writing automated acceptance tests that represent complex system stories or *scenarios*. While classical unit tests focus more on checking internal functionalities of classes, BDD tests should be examples that anyone from the development team can read and understand. The use of scenarios is common not only at the code level but also at the level of (abstract) models. In the `Asmeta` framework, we have introduced the idea of using scenarios for validating ASMs and developed a language AVALLA (and a corresponding tool `AsmetaV`)[15] for writing scenarios. With AVALLA, the designer can describe a scenario, which is briefly a sequence of external actor actions and expected reactions of the system. Scenarios can be executed in order to check whether the actual behavior of the system conforms to the requirements. In a classical model-driven engineering approach, the designer writes the abstract specification and then through a process of systematic transformation, s/he can obtain the source code together with unit tests. In this way, the generated code comes with a set of unit tests that can also be used later for regression testing. In this section, we extend the `Asmeta` framework with a translator, called ASMETAVBDD, which translates an abstract scenario written in the AVALLA language to the BDD code.

Among all the testing framework, we have selected the Catch2 (https://github.com/catchorg/Catch2) which aims to be a modern, C++-native, header-only, test framework for unit-tests, TDD and BDD. Catch2 and AVALLA share several concepts that can be found in every BDD approach, so the translation from AVALLA to Catch2 is rather straightforward. Such translation complements the generation of C++ code[3] and the generation of C++ tests[4] already supported by `Asmeta`. It takes an AVALLA scenario and produces the C++ code. Table 8 summarizes the transformation rules we have defined, which are briefly described here:

```
BOOST_AUTO_TEST_SUITE(TestCounter)

  BOOST_AUTO_TEST_CASE( my_test_0 ){
    // instance of the SUT
    counter sut;
    // check state
    BOOST_CHECK(sut.counter[0]==0);
    // set monitored variables
    sut.signal=true;
    // call main rule
    sut.step();
    BOOST_CHECK(sut.counter[0]==1);
    sut.signal=true;
    sut.step();
    BOOST_CHECK(sut.counter[0]==2);
    sut.signal=true;
    sut.step();
    BOOST_CHECK(sut.counter[0]==3);
    sut.signal=false;
    sut.step();
    BOOST_CHECK(sut.counter[0]==3);
  }

  BOOST_AUTO_TEST_CASE( my_test_1 ){ ...
  }

  BOOST_AUTO_TEST_CASE( my_test_2 ){ ...
  }

  BOOST_AUTO_TEST_CASE( my_test_5 ){
    // instance of the SUT
      // instance of the SUT
    counter sut;
    // check state
    BOOST_CHECK(sut.counter[0]==0);
    // set monitored variables
    sut.signal=true;
    // call main rule
    sut.step();
    BOOST_CHECK(sut.counter[0]==1);
    sut.signal=true;
    sut.step();
    BOOST_CHECK(sut.counter[0]==2);
    sut.signal=true;
    sut.step();
    BOOST_CHECK(sut.counter[0]==3);
    ...
  }

BOOST_AUTO_TEST_SUITE_END()
```

Code 13 Counter test suite example

scenario is simply translated to a SCENARIO macro. The name is taken from the AVALLA scenario.

load is translated to a declaration of an instance of the class that is obtained by translating the ASM to C++. Let's call that instance X.

set all the set commands before a `step` command are grouped together and translated to a WHEN macro. Inside WHEN, every set is translated to a simple assignment to the field representing the monitored function.

check is translated to a REQUIRE macro. The argument of the check is translated to a C++ term, by reusing the translation already defined in `Asm2C++`.

step represents an abstract step of ASM. In C++, it is translated to a call of the function `r_main()` that computes the update set, and a call of the function `fireUpdateSet()` that applies the update set to the current state in order to apply the new values of controlled location computed by the main rule.

exec allows the user to execute an arbitrary `Asmeta` rule. The tool translates the rule to a C++ function that is called whenever `exec rule` is invoked.

By following the aforementioned rules, the ASMETAVBDD tool generates a C++ file that can be compiled and executed. If the scenario is validated for the ASM, and translations to C++ of the ASM and of the AVALLA scenario are correct, then the BDD scenario in C++ will be correct and, when executed, no REQUIRE instruction of the scenario will fail. However, there are two possible uses of the obtained BDD code. First, the user can manually inspect the BDD test and check whether the C++ code actually has the intended behavior. In this way, we can produce the C++ code with its tests also given in the BDD style. The use of the BDD style should increase the comprehension of the test by nontechnical stakeholders like customers or business experts. Second, the scenarios can be used for regression testing. Indeed, sometimes the C++ code is modified in order to add further details after its automatic generation. If one wants to check that the expected behaviors are still preserved after the modification, one can run the BDD tests again for confirmation.

AVALLA	Catch2

scenario name
load spec.asm
	SCENARIO(name){
	spec X; // create an instance of spec
	...
	}

set block
set $l_1 = v_1$
...
set $l_n = v_n$
	WHEN("set monitored variables"){
	X.l1=v1;
	...
	X.ln=vn;
	}

check expr	REQUIRE(X.C++expr);
step	THEN("n-th step occurs"){ ..... X.mainRule(); X.fireUpdateSet(); }
exec rule	add function definition `rule()` and call it

TABLE 8 Translation of AVALLA constructs to Catch2 macros

Generation of BDD scenarios on the Counter example

Code 14 shows an example of an automatic BDD scenario generated on the Counter example. The scenario simulate an execution where the counter starts from 0 to 3, then it interrupts the execution. The scenario instantiates a C++ object of the generated C++ class, then the initialized functions are checked to verify if the value is the same as in ASM initialization section. Once the values are verified the monitored functions are set and a step of ASM is executed. Once the step is performed, the scenario starts again from the check of the functions.

```
SCENARIO(count3){
  // instance of the SUT
  counter sut;
  REQUIRE (sut.counter[0]=0);
  WHEN (``set monitoredvar'')
    sut.signal=true;
  }
  THEN (``1-st step occurs'')
    sut.step();
  }
  REQUIRE (sut.counter[0]=1);
  WHEN (``set monitoredvar'')
    sut.signal=true;
  }
  THEN (``2-st step occurs'')
    sut.step();
  }
  REQUIRE (sut.counter[0]=2);
  WHEN (``set monitoredvar'')
    sut.signal=true;
  }
  THEN (``3-st step occurs'')
    sut.step();
  }
  REQUIRE (sut.counter[0]=3);
  WHEN (``set monitoredvar'')
    sut.signal=false;
  }
  THEN (``4-st step occurs'')
    sut.step();
  }
  REQUIRE (sut.counter[0]=3);
}
```

Code 14 Counter BDD scenario example

5 | VALIDATION OF THE TRANSFORMATION

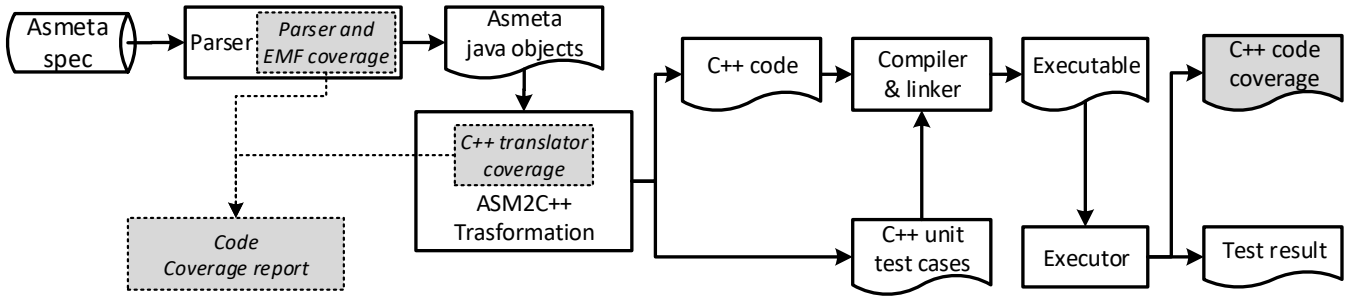

FIGURE 7 Validation process

In the previous sections, we have presented the automatic transformation of ASMs models to C++ code. However, no validation activities have been performed to guarantee the correctness of the transformation process. In this section, we define a mechanism to test the correctness of the M2T transformation with respect to two main criteria – syntactical correctness and semantic correctness – which is based on the definition of conformance between the specification and the code. Using this approach, we have devised a process able to test the generated code by reusing unit tests presented in Section 4.4.

In principle, to validate the transformation τ, we would need a set of inputs (a set of ASMs) or a way to generate inputs according to some criteria and an oracle that tells whether the output of τ (C++ code) is what is intended (for example, the user could write by hand the expected C++ code for each ASM in the test set). We follow a different path since we use the unit tests to validate the transformations. In our approach, to check whether the resulting code is what is intended, we first check the well-formedness of the code and then we test its behavior in order to check whether it conforms to the original ASM. This is consistent with our definition of correctness given in Definition 3 and is based on Theorem 1. This is a sort of *indirect* testing [16], in which we do not test directly the transformation rules but the results of such transformations. We exploit the fact that both the ASM and its translation to C++ are executable.

The validation process is depicted in Figure 7 and is explained as follows. Given an `Asmeta` textual specification A, A is parsed by the `Asmeta` parser that builds the corresponding Java objects. For the specification A, we apply our `Asm2C++` tool that implements the code transformation τ in order to obtain the C++ code. Besides, we apply the test generator component [4] and generate a set of abstract test cases that can be translated to C++ unit tests. Then, we perform the following validation activities: testing the transformation correctness and coverage computation.

5.1 | Code generation correctness

First, we want to introduce the notion of *conformity* of the target C++ code to the source ASM. Formally, we can define the M2T transformation as a function τ that takes an ASM A and returns a C++ class $\tau(A)$ with the corresponding fields and methods. Each location l of the ASM A is transformed to a member (field or function) of the class τ(A) (as explained by Bonfanti et al. [3]).

Definition 1 (State conformance)**.** Given an ASM A, we say that the state of an object O of the class $\tau(A)$ conforms to a state s of A if the value of every location l in s is equal to the value of $\tau(l)$ in the target object O.

Informally, to compare ASM states and C++ states we look at the values of the ASM functions that are translated to C++ members. To compare values, we use the equality but in the future we may extend the concept of conformity between locations in order to introduce some tolerance, e.g., by allowing a small difference between two values. We can refer to *controlled conformity*, if we restrict to only controlled locations.

Additionally, we want to introduce the notion of behavioral conformance. In our approach, we want that the target C++ class C preserves the behavior of the ASM. Since ASMs are executable, we require that every execution of the class C has a corresponding behavior in the abstract specification.

Definition 2 (Behavioral conformance)**.** We say that a class $C = \tau(A)$ behaviorally conforms to the ASM A, if starting from any reachable state r of any object O of C such that r is conforming to the state s of A, by executing `O.step()` we obtain a state r' that is controlled conforming to the next state s' of A.

Informally, our C++ code behaves like the original ASM, if starting from a conforming state (with the same monitored and controlled locations) and executing a `step`, then the code will arrive to a next state that has the same controlled locations.

We now introduce the concept of *correctness* of the M2T transformation. We deal with the correctness from two distinct points of view: first *syntactic* or *type-correctness* and second *semantic* or *behavioral* conformity.

Definition 3 (Transformation correctness)**.** We say that the transformation $\tau(A)$ is correct if the C++ class is syntactically correct and behaviorally conforms to A.

Verifying the correctness of the translation τ would require the use of formal techniques like model checking or theorem proving. As shown by Rahim et al.[16], several attempts already exist in this direction. In our case, this would require, at least, to formalize the target language C++ and this would be a great overhead. Moreover, proving the correctness of the transformation may still not be enough in case of critical systems. For such systems, the transformation should also be tested in any case (rephrasing what Ed Brinksma said in his 2009 keynote at the Testcom/FATES conference: "Who would want to fly in an airplane with software automatically generated with a code generator that has never been tested?"). As observed by Conrad et al.[17], a translation validation approach, that is based on testing, seems to be a better solution in an engineering context. Therefore, we have concentrated our efforts in validating the transformation by testing. This activity exploits the generated unit tests, as explained in Section 4.4, and is based on the following theorem.

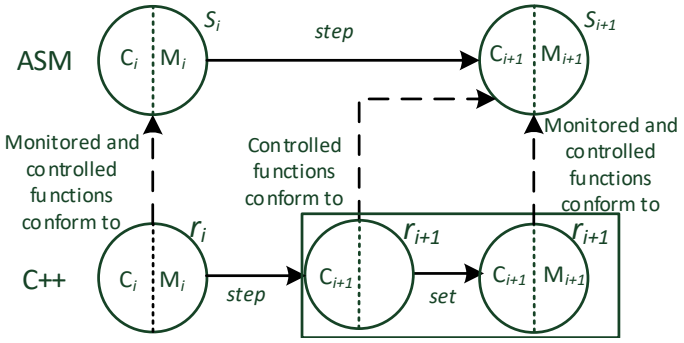

FIGURE 8 ASM/C++ conformance

Theorem 1 (Correctness by testing)**.** Given a C++ test `t` obtained by translating any run $s_1, \ldots, s_n$ of A, if $\tau(A)$ is correct, then, when executing `t`, each C++ state before the i-th `O.step()` will conform to s_i, all controlled locations will be checked by `t`, and `t` will pass with no errors.

Proof. The evolution and relations between `t` and the abstract states are depicted in Figure 8. If τ is correct, then the C++ code is correct and it can be executed. Let's consider the pair of states s_i and s_{i+1} and assume that r_i in C++ conforms to s_i in the abstract run both in the controlled part $\mathsf{C_i}$ and the monitored part $\mathsf{M_i}$. The controlled conformity of r_{i+1} is guaranteed, thanks to Definition 2, by executing immediately before each state the instruction `O.step()` (see Table 7). Then, the unit test sets the monitored variables in C++ to the values in s_{i+1} (see Table 7). At the end, the state in C++ immediately before the $(i + 1)$-th step conforms again to s_{i+1}. The test will check the controlled part, and due to the assumption that τ is correct, it will find the expected values for the controlled part. By induction on i, we can prove the theorem. □

Thanks to Theorem 1, we are sure that every test will check the conformance of the states in it with the original sequence of the ASM, and that if a test fails, then there is a fault in the translation. Of course, testing cannot prove the correctness of the transformation but can help us in gaining confidence in the translation correctness. In the following section, we explain the process we have devised to put in practice the proposed methodology.

5.2 | Testing the transformation correctness

Syntactic correctness

Using the C++ compiler, we first check the syntactical correctness of the generated code. We use the `-Wall` option to turn on all the possible warning flags and quit the process in case of an error. This first phase captures translation faults that produce invalid source code. Also the tests are compiled in order to obtain the corresponding obj files. The objs for the `Asmeta` specification and for the tests are linked together.

Semantic correctness

The obtained executable is executed in order to check that the behavior as specified by the tests corresponds to the behavior of the generated code. The tests will set the suitable monitored values and check the conformance of the controlled parts. In this way, we test the semantic correctness of the code according to Theorem 1. A failing test means that the C++ code does not conform to its specification and since the code has been obtained by applying the transformation, a fault in the transformation has been found.

5.3 | Tools used

To support the validation process, we have used several tools. `Ant`[**] is a tool that supports users while developing software across multiple platforms. The configuration files are written using XML where each file contains one project and one or more targets. A target is composed of one or more tasks - pieces of code that can be executed. Moreover, the configuration file contains properties to support the user in customizing the build process.

To compute the java code coverage we use `JaCoCo`[††], which is a free code coverage library for Java. `JaCoCo` requires `Ant` tasks to compile and run Java programs and to create the coverage report of the executed code. We have written a project using `Ant` to automatically compile and run JUnit tests to test the `Asm2C++` generator. Once the specifications are translated into C++ code, another task generates C++ unit tests and runs the tests on the generated C++ code. After C++ unit tests are executed, the `Ant` file invokes a task to run the `JaCoCo` tool, which provides the coverage of the selected code. To compute the coverage of the C++ code, we use the `gcov` utility that instruments the generated C++ source code and outputs coverage information when it is executed.

5.4 | Dealing with internal nondeterminism

In ASMs, the internal nondeterminism is represented by the following `choose` rule:

choose x **in** D **with** P **do** R

meaning to execute rule R with an arbitrary x chosen in D, which is a domain or a set of elements, and satisfying the property P. In C++, the choose rule is translated by randomly searching an element in D satisfying P and then executing the code obtained by the translation of R. In this way, however, the ASM and the C++ code may choose different values for x. The test obtained from the abstract test case may, therefore, fail only because of this reason. To tackle this problem, we have enabled the test case generator and the C++ translator to enforce a deterministic behavior that consists in taking the *first* element of D such that P is true and use that for the variable x. Substituting a known nondeterministic behavior with a deterministic alternative is adopted also in [18]. Although this approach cannot guarantee that the actual nondeterministic translation is correct, it allows us to test the translation of the choose rule and the specification containing it.

5.5 | Coverage

Although it suffers from well-known shortcomings, the measure of the coverage of software artifacts during testing can give a good feedback about the depth of the testing activity itself. For this reason, we propose to measure the coverage of the following aspects.

- First, the *coverage of the source language*, `AsmetaL` in our case, gives a good indication on how many constructs are tackled by the transformation under test τ. The more constructs τ is able to deal with during testing, the higher the

[**] https://ant.apache.org/
[††] http://www.eclemma.org/jacoco/

applicability of τ is. A request of a good level of coverage avoids the problem of transformations that are well tested but only on a limited set of source specifications. In our approach, we instrument the `Asmeta` parser in order to collect the information during parsing. This represents the coverage of the *inputs* of the transformation.

- Second, the *coverage of the transformation code*, the `Asm2C++` code that implements the transformation written in Xtend and Java in our case, gives a good indication on how much the transformation code itself is tested. If some parts of the transformation are never covered, there is the risk that some critical conditions are actually not tested, or that some code is useless and never used therefore. This represents the pure coverage of the transformation.
- Third, the *coverage of the produced code*, the C++ code including the unit tests in our case, gives an indication on how much the tests are able to exercise the generated C++ code. Although among the three coverage measures this is less significant as it depends also on the technique used to generate the tests, it is important to check whether there are parts of the produced code that are never covered and this may be a signal that the transformation produces some meaningless code. This represents the coverage of the *outputs* of the transformation.

Figure 9 shows the coverage of the source language in terms of number of `Asmeta` constructs covered during parsing. The coverage increases with the number of specifications, until most of the constructs are covered (80% of the total). We did not cover all of them, because there are some constructs that are not used in any `Asmeta` specification in the repository. We initially started to write ad hoc `Asmeta` specifications but then we realized that the language contains useless constructs and such language over-specification should be addressed before in order to simplify the language.

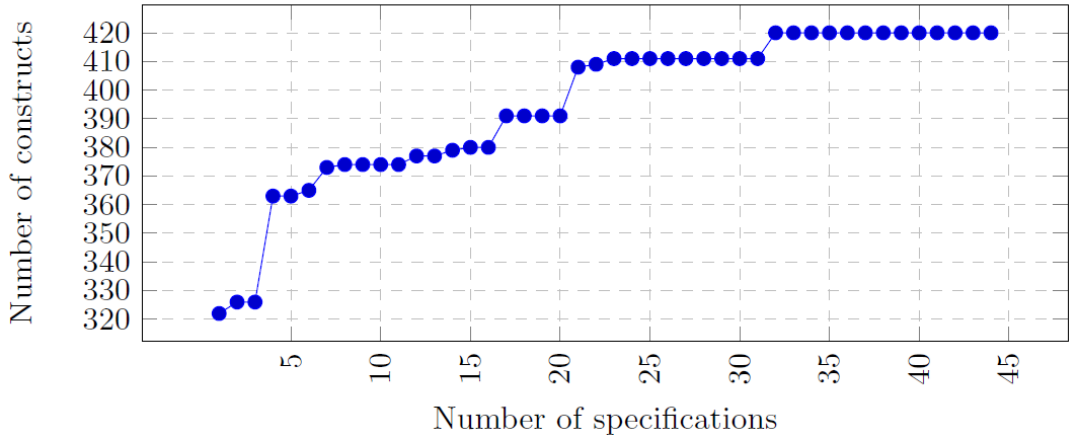

FIGURE 9 Coverage of the `Asmeta` parser

The coverage of the transformation code is shown in Figure 10. The result obtained is satisfactory because most of the code is covered, despite not all the classes are 100% covered. By analyzing the missing coverage, we found that it is due to some error checking that is never performed since it refers to some errors already caught by the parser, the code will never be executed, and consequently is not coverable.

Regarding the coverage of the produced code, initially, the value was low because the ASM rules were translated in two execution modes: the first was in the parallel mode (the standard ASM mode), while the second was in the sequential mode. The sequential mode is used rarely in ASM specifications and the unused code contributes to decrease the percentage of code coverage (the highest coverage was $\approx$70%). For this reason, we have improved the translator by producing the sequential version of rules only if they are actually called by `seqBlock` rules. The result of this improvement is a higher percentage of code coverage and in most cases it reaches 100% of the generated code.

Element	Missed Instructions	Cov.
TermToCpp		83%
RuleToCpp		88%
FunctionToCpp		79%
FunctionToH		88%
Util		71%
DomainToH		83%
ExpressionToCpp		88%
FindMonitoredInControlledFunct		71%
DomainToCpp		84%
ToString		100%
DomainContainerToH		100%

FIGURE 10 `Asm2C++` coverage

6 | RELATED WORK

Automatic code generation from formal specifications is available as a part of tool support for several formal methods. SCADE[‡‡] and MATLAB/Simulink[§§] provide this feature as a commercial off-the-shelf solution. The formal method B [19], on the other hand, provides this facility in the form of the Atelier B platform[¶¶], that comes with code generators for different target languages, including C, C++, Java, and Ada, and its Community Edition is freely available without any restriction. EventB2Java [20] is another tool that generates executable code implemented as a plug-in of the Rodin platform.

As best of our knowledge, there is no state of the art, reusable and publicly available tool for the ASM method that is capable of automatically generating programming language code from formal specifications written in the ASM method. In the past, Schmid [10] introduced a compilation scheme to transform an ASM specification (written in ASM-SL) into C++ code, but this work was done within a company setting. Although some of the key results of the proposed compilation scheme were useful for our work too as shown in Section 4.2.

The challenging nature of the model transformation process also creates the need for validation. This need and the associated challenges have been documented in several publications [21,22,23]. A comprehensive survey on the related state of the art are presented by Rahim et al. [16] and by Calegari and Szasz [24].

Wimmer et al. [25] present a language-agnostic approach for testing model-to-text and text-to-model transformations. They extend the object constraint language (OCL) with additional string operations to specify contracts for practical examples and to evaluate the correctness of current UML-to-Java code generators offered by some UML tools. As compared to this work, our input models are verified and validated by both users and tools, i.e., they are well-formed, implement the specified requirements, and do not contain unintended behaviors.

Conrad [17] proposes a translation validation workflow for the generated code in the context of the IEC 61508 standard. The translation validation process is comprised of (a) numeric equivalence testing between the generated code and the corresponding model, and (b) additional measures to demonstrate that unintended functionality has not been introduced during the translation process. In a similar work, Sampath et al. [26] present a technique for verifying and validating Stateflow (a hierarchical state-machine modeling language that is part of the Simulink/Stateflow tool-suite from The MathWorks Inc.) model translation to C code. However, both these works are based on the proprietary tool Simulink. Stateflow (a graphical language) is mostly used in combination with Simulink, where the former is used to model the discrete controllers and the latter is used to model the continuous dynamics of a system [27]. Our work, on the other hand, uses the `Asmeta` framework, which is an open-source project, freely available, based on the rigorous method ASM, and allows to model a variety of systems.

Küster et al. [28] present their initial experiences with a white box model-based approach for testing model transformations within the context of business process modeling. They propose multiple techniques for constructing test cases and show how to use them to locate errors in model transformations. As aforementioned, this work is performed within the context of business process modeling and uses a supported notation. Our work, in comparison, is generally applicable across multiple domains and uses ASMs, which is a scientifically well-founded method for systems engineering.

[‡‡] http://www.esterel-technologies.com/products/scade-suite/
[§§] https://www.mathworks.com/products/simulink/
[¶¶] http://www.atelierb.eu/en/

Stümer et al.[29] present a general and systematic test approach for model-based code generation. This approach undertakes formal descriptions of the optimizations under test by using graph transformation rules. The proposed tool automatically creates test models (first-order test cases) from the classification tree, which is used to derive a formal description of the input space of an optimization rule. In a further step, test vectors (second-order test cases) are generated, which ensure structural coverage of the test model and the corresponding code. Model and generated code then undergo a back-to-back test using these test vectors. A signal comparison of the test outputs is used to determine functional equivalence between the model and the code. The main difference between this work and our approach is that, although many of their observations are general, they target Simulink and Stateflow programs. Moreover, we extend their use of coverage information to measure the quality of the testing activity by explicitly distinguishing between several types of coverage.

In another work, Satpathy et al.[30] generate test cases by performing symbolic execution over a B model, and from those test cases they obtain a Java program. The resulted Java program acts as a test driver. When it runs in conjunction with the implementation, testing is performed in an automatic manner. A similar work is reported by Ambert et al.[31]. In this paper, authors present the BZ-Testing Tool (BZ-TT), which is capable of generating functional test cases from B as well as Z[32] specifications using constraint logic programming.

Engel et al.[33] presented a method for automatic generation of self-contained unit tests in the JUnit format[##]. The implementation is based on the verification system KeY[34] and supports the JAVA CARD programming language[35]. The approach exploits the implementation of a system and does not necessarily require its detailed formal specification.

Cheon et al.[36] present an approach to implement unit test oracles from formal behavioral interface specifications. Instead of writing testing code, a programmer writes formal specifications (e.g., pre- and postconditions) that are later used by runtime assertion checkers as the decision procedure for test oracles. The authors have implemented the proposed approach using the Java Modeling Language (JML)[37] and the JUnit testing framework.

Lamancha et al.[38] present an approach to automatically generate test cases through model transformations. Their work takes as input UML 2.0 sequence diagrams[39] and automatically derive test cases scenarios that conform the UML Testing Profile[||||]. In this work, these test case scenarios are automatically transformed using the model to text transformation. The models used for test case generation in this work are not necessarily amenable to verification and validation activities.

One of the main differences between the work presented in this paper regarding testing and other works is that our approach is grounded in the Asmeta framework that supports the complete model-driven software engineering paradigm. Starting from the specification, the models are rigorously specified and analyzed for their correctness through validation and verification tools. After that, the `Asm2C++` tool translates the specification in C++ code and generates C++ unit tests to verify the correct behavior of the generated C++ code.

7 | CONCLUSION

In this paper, we have presented the translation of ASM models into C++ code to automatically obtain the system implementation. The generation of unit tests starting from the ASM models and the generation of BDD scenarios from `Asmeta` scenarios have been implemented to automatically obtain tests for the implementation. Finally, we have presented the validation of the transformation to guarantee the correctness of the process.

The generation process of the C++ code starts from the ASM specification and using `Asm2C++` the C++ code of the model is automatically generated. The translation from `Asmeta` to C++ code is shown in Section 4.2. We firstly show how a generic ASM model is translated and then for each ASM constructor we display the translation adopted. We have dealt with two characteristics of ASM models – the parallelism and the nondeterminism. We have found the solution to maintaining in the C++ code the ASM behavior. Furthermore, starting from the C++ code generated by `Asm2C++` tool, we have applied some steps to generate a runnable Arduino project as shown in Figure 5. The unit tests are generated automatically starting from the ASM model by following the process shown in Figure 6. Starting from the ASM specification, the abstract tests are generated by a test case generator and then they are translated into C++ unit tests using `Asm2C++`. We have adopted two approaches: the first approach is based on coverage-driven generation and covers the C++ code better than the second approach which generates test sequences randomly. However, random generation can be applied to a greater set of specifications and it requires much less resources. BDD tests are generated starting from AVALLA scenarios using the ASMETAVBDD tool. Most of the textual information in the

[##] http://junit.org
[||||] http://utp.omg.org/

BDD scenario is generated automatically from the corresponding AVALLA scenario. The process to automatically validate the transformation correctness from `Asmeta` specifications to the C++ code is shown in Section 5. The process consists in parsing an `Asmeta` specification, and generating the C++ code and unit test cases. The source code is compiled, linked, and executed. During tests execution, possible faults can be found and the code coverage information is collected.

The application of the activities presented before guarantee that starting from a validated and verified `Asmeta` specification, we can automatically generate the code implementation with unit tests executable on the code.

ACKNOWLEDGMENT

This work has been partially supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the State of Upper Austria in the frame of the COMET center SCCH, and the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

1. Börger E, Stark RF. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc. . 2003.

2. Arcaini P, Gargantini A, Riccobene E, Scandurra P. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience* 2011; 41: 155–166.

3. Bonfanti S, Carissoni M, Gargantini A, Mashkoor A. Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino. In: Barrett C, Davies M, Kahsai T., eds. *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*Springer International Publishing; 2017: 295–301.

4. Bonfanti S, Gargantini A, Mashkoor A. Generation of C++ Unit Tests from Abstract State Machines Specifications. In: Society IC., ed. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*; 2018: 185-193

5. Bonfanti S, Gargantini A, Mashkoor A. Generation of Behavior-Driven Development C++ Tests from Abstract State Machine Scenarios. In: Springer International Publishing; 2018: 146–152

6. Bonfanti S, Gargantini A, Mashkoor A. Validation of Transformation from Abstract State Machine Models to C++ Code. In: Springer International Publishing; 2018: 17–32.

7. Schmidt DC. Model-Driven Engineering. *IEEE Computer* 2006; 39(2): 25–31.

8. Kossak F, Mashkoor A, Geist V, Illibauer C. Improving the Understandability of Formal Specifications: An Experience Report. In: Salinesi C, Weerd v. dI., eds. *Requirements Engineering: Foundation for Software Quality - 20th International Working Conference, REFSQ 2014, Essen, Germany, April 7-10, 2014. Proceedings*Springer International Publishing; 2014: 184–199

9. Czarnecki K, Helsen S. Classification of model transformation approaches. In: Bettin J, Emde Boas vG, Agrawal A, Willink E, Bezivin J., eds. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. 45. USA. ; 2003: 1–17.

10. Schmid J. Compiling abstract state machines to C++. *Journal of Universal Computer Science* 2001; 7(11): 1068–1087.

11. Dalvandi M, Butler M, Rezazadeh A, Fathabadi AS. Verifiable Code Generation from Scheduled Event-B Models. In: Springer International Publishing. 2018 (pp. 234–248)

12. Gargantini A, Riccobene E. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *JUCS - Journal of Universal Computer Science* 2001; 7(11): 1050–1067.

13. Arcaini P, Gargantini A, Riccobene E. Decomposition-Based Approach for Model-Based Test Generation. *IEEE Transactions on Software Engineering* 2017; PP(99).

14. Solis C, Wang X. A Study of the Characteristics of Behaviour Driven Development. In: IEEE ., ed. *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*; 2011: 383-387

15. Carioni A, Gargantini A, Riccobene E, Scandurra P. A scenario-based validation language for ASMs. In: Börger E, Butler M, Bowen JP, Boca P., eds. *ABZ Conference, September 16-18, 2008, London, UK*. 5238 of *Lecture Notes in Computer Science*. Springer; 2008: 71-84.

16. Ab. Rahim L, Whittle J. A survey of approaches for verifying model transformations. *Software and Systems Modeling* 2015; 14(2): 1003–1028.

17. Conrad M. Testing-based Translation Validation of Generated Code in the Context of IEC 61508. *Form. Methods Syst. Des.* 2009; 35(3): 389–401.

18. Tillmann N, Halleux dJ. Pex–White Box Test Generation for .NET. In: Beckert B, Hähnle R., eds. *Tests and Proofs*Springer Berlin Heidelberg; 2008; Berlin, Heidelberg: 134–153.

19. Abrial JR. *The B-book: assigning programs to meanings*. Cambridge University Press . 2005.

20. Rivera V, Cataño N, Wahls T, Rueda C. Code generation for Event-B. *STTT* 2017; 19(1): 31–52. doi: 10.1007/s10009-015-0381-2

21. Baudry B, Ghosh S, Fleurey F, France R, Le Traon Y, Mottu JM. Barriers to Systematic Model Transformation Testing. *Commun. ACM* 2010; 53(6): 139–143.

22. France R, Rumpe B. Model-driven Development of Complex Software: A Research Roadmap. In: Society IC., ed. *2007 Future of Software Engineering*FOSE'07. ; 2007; Washington, DC, USA: 37–54.

23. Van Der Straeten R, Mens T, Van Baelen S. Challenges in Model-Driven Software Engineering. In: Chaudron MRV., ed. *Models in Software Engineering*Springer; 2009; Berlin, Heidelberg: 35–47.

24. Calegari D, Szasz N. Verification of Model Transformations: A Survey of the State-of-the-Art. *Electronic Notes in Theoretical Computer Science* 2013; 292: 5 - 25. Proceedings of the XXXVIII Latin American Conference in Informatics (CLEI).

25. Wimmer M, Burgueño L. Testing M2T/T2M Transformations. In: Moreira A, Schätz B, Gray J, Vallecillo A, Clarke P., eds. *Model-Driven Engineering Languages and Systems*Springer; 2013; Berlin, Heidelberg: 203–219.

26. Sampath P, Rajeev AC, Ramesh S. Translation validation for stateflow to C. In: ACM Press . , ed. *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*; 2014: 1-6

27. Hamon G, Rushby J. An Operational Semantics for Stateflow. In: Wermelinger M, Margaria-Steffen T., eds. *Fundamental Approaches to Software Engineering*Springer Berlin Heidelberg; 2004; Berlin, Heidelberg: 229–243.

28. Küster JM, Abd-El-Razik M. Validation of Model Transformations – First Experiences Using a White Box Approach. In: Kühne T., ed. *Models in Software Engineering*Springer; 2007; Berlin, Heidelberg: 193–204.

29. Stuermer I, Conrad M, Doerr H, Pepper P. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering* 2007; 33(9): 622–634.

30. Satpathy M, Butler M, Leuschel M, Ramesh S. Automatic Testing from Formal Specifications. In: Gurevich Y, Meyer B. , eds. *Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*Springer Berlin Heidelberg; 2007; Berlin, Heidelberg: 95–113

31. Ambert F, Bouquet F, Chemin S, et al. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In: Hierons R, Jeron T., eds. *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR*. 2. ; 2002: 105–120.

32. Spivey JM. *Understanding Z: a specification language and its formal semantics*. 3. Cambridge University Press . 1988.

33. Engel C, Hähnle R. Generating Unit Tests from Formal Proofs. In: Gurevich Y, Meyer B., eds. *Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*Springer Berlin Heidelberg; 2007; Berlin, Heidelberg: 169–188

34. Beckert B, Hähnle R, Schmitt PH. *Verification of object-oriented software: The KeY approach*. Springer-Verlag . 2007.

35. Chen Z. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional . 2000.

36. Cheon Y, Leavens GT. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: Magnusson B. , ed. *ECOOP 2002 — Object-Oriented Programming: 16th European Conference Málaga, Spain, June 10–14, 2002 Proceedings*Springer; 2002; Berlin, Heidelberg: 231–255

37. Leavens GT, Baker AL, Ruby C. *Formal Underpinnings of Java Workshop (at OOPSLA 98)*ch. JML: a Java modeling language: 404–420; 1998.

38. Lamancha BP, Reales P, Polo M, Caivano D. Model-Driven Test Code Generation. In: Maciaszek LA, Zhang K. , eds. *Evaluation of Novel Approaches to Software Engineering: 6th International Conference, ENASE 2011, Beijing, China, June 8-11, 2011. Revised Selected Papers*Springer Berlin Heidelberg; 2013; Berlin, Heidelberg: 155–168

39. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language reference manual*. Pearson Higher Education . 2004.