Validation of transformation from Abstract State Machine models to C++ code*

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

[1] Università degli Studi di Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it
[2] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
atif.mashkoor@scch.at
[3] Johannes Kepler University, Linz, Austria
atif.mashkoor@jku.at

Abstract. The automatic transformation of models to code is one of the most important cornerstones in the model-driven engineering paradigm. Starting from system models, users are able to automatically generate machine code in a seamless manner with an assurance of potential bug freeness of the generated code. `Asm2C++` [4] is the tool that transforms Abstract State Machine models to C++ code. However, no validation activities have been performed in the past to guarantee the correctness of the transformation process. In this paper, 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. Coverage measures give a user the confidence that the generated code has the same behavior as specified by the ASM model.

1 Introduction

The Abstract State Machines (ASM) method [6] is a formalism that is used to guide the rigorous development of software and systems. The ASM-inspired development starts with an abstract specification of a system and then continues until all details of the system have been captured through a sequence of refinements. During this process, the specifier can apply classical validation and verification (V&V) techniques like simulation, scenarios validation, and model checking. The last step of the development process is the transformation of models into code. If not performed carefully, this step can be critical and error-prone.

The automatic transformation of models into code is an important cornerstone of model-driven engineering [12]. This is also a common practice in industry. For example, Airbus uses automatic code synthesis from SCADE models

* The writing of this article is supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

to generate the code for embedded controllers in the Airbus A380 [3]. Following this paradigm, we have built the tool `Asm2C++` [4], which is able to generate C++ code from formal specifications given in terms of ASMs. Furthermore, it is able to produce unit test cases [5], which can be used, for example, for regression testing.

The code generation activity, however, may introduce new issues in the development process, e.g., an error in model transformation may introduce faults in the code that jeopardize all the V&V activities performed during the modeling phase. Therefore, it is very critical that the generated code is syntactically well formed and, mostly, it faithfully transforms the specification into code. This also means that the code transformation process must also be analyzed, validated, and verified, which at times can become a difficult task [3].

There exist several techniques for the validation of a transformation, including the use of theorem proving or model checking [1]. In this paper, we propose an approach based on testing. In principle, testing a generated code could be a useless activity if the transformation could be formally proven correct. In practice, however, specifiers want to test code transformations in order to gain confidence that errors are not inadvertently introduced at any step (including code compiling, for example)[4]. In order to address such issues, in this paper, we tackle the problem of validation of model-to-code transformation τ by contributing in the following directions:

1. we formally define when the generated code is correct both syntactically and semantically w.r.t. the original specification,
2. we show that generated tests can detect possible errors in τ and help the designer to fix them in the implementation of τ,
3. we setup a methodology that uses a combination of code compiling and execution in order to validate τ, and
4. we provide a user with a measure (coverage) that helps in building the confidence that τ is correct.

The rest of the paper is organized as follows. We present ASMs in Sect. 2. In Sect. 3, we present the process applied to transform ASMs into C++ code. The validation of the transformation is presented in Sect. 4 and corresponding results are presented in Sect. 5. The related work is presented in Sect. 6. The paper is concluded in Sect. 7.

2 Abstract State Machines and `Asmeta` framework

Abstract State Machines (ASMs) [6] are an extension of Finite State Machines (FSMs), where unstructured control states are replaced by states with arbitrarily complex data. ASM *states* are mathematical structures, i.e., domains of objects

[4] 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?"

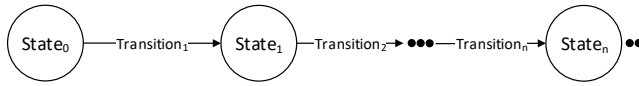

Fig. 1. An ASM run with a sequences of states and state-transitions (steps)

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 transitions rules (see Fig. 1).

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 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) and of 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 Sect. 4.4.

`Asmeta` framework. The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Fig. 2 shows the development process based on ASMs. The process is supported by the `Asmeta` (ASM mETAmodeling) framework[5] [2] which provides a set of tools to help the developer in various activities:

- **modeling**: the system is modeled using the language `AsmetaL`. The user is supported by the editor `AsmEE` and by `AsmetaVis`, the ASMs visualizer which

[5] http://asmeta.sourceforge.net/

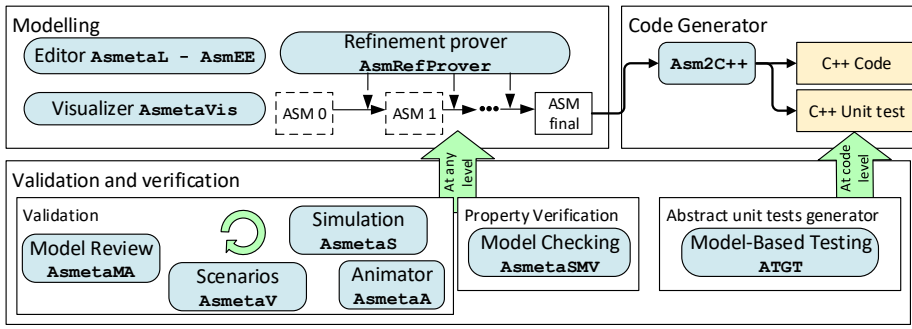

Fig. 2. The ASM development process powered by the `Asmeta` framework

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.

- **validation**: the process is supported by the model simulator `AsmetaS`, the scenarios `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. `AsmetaS` 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 behavior. 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 to C++ unit tests.

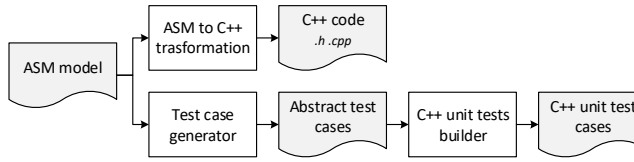

Fig. 3. `Asm2C++` tool

3 Code generation

The `Asm2C++` tool implements the model-to-code transformation. The tool is divided into two activities: transformation of the ASMs specifications into C++ code and generation of C++ unit tests starting from the ASMs specifications (see Fig. 3). `Asm2C++` is based on Xtext[6], a framework for the development of domain-specific languages, which provides facilities for parsing and code generation and is fully compatible with the Eclipse Modeling Framework. The code generator has been developed as a model-to-text (M2T) transformation. The transformation code is written mainly in Xtend - a Java dialect provided by the Xtext framework with features for code generation and text transformation.

3.1 C++ code generation

The `Asm2C++` tool transforms an ASM to a C++ class [4]. The generated C++ class is split into a header (*.h*) and a source (*.cpp*) file. The header file (see Code 1) contains 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 2) . The simulation of an ASM step, has been implemented using the step() method which calls sequentially the main method and the fireUpdateSet() method. The main method corresponds to the translation of the main rule into C++ code, while the fireUpdateSet method updates the locations to the next state values.

3.2 C++ unit tests generation

The `Asm2C++` tool is also able to produce unit tests from ASM specifications [5]. It generates abstract tests starting from the ASM specification and then the tests are translated into C++ unit tests using the Boost Test C++ library.[7] The abstract tests are generated in two different ways. The first is based on the `Asmeta` simulator while the second exploits the `ATGT` tool. Once the unit tests and the C++ code of the ASM specification are compiled, they are linked together and the tests are run on the code.

[6] https://www.eclipse.org/Xtext/
[7] https://www.boost.org/

```
#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();
void step();
};

#endif
```

Code 1. .h code

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

// Conversion of ASM rules in C++ methods
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];
}

void  asmSpecification::step(){
        mainRule();
        fireUpdateSet();
}
```

Code 2. .cpp code

The translation from abstract tests to concrete tests is done by following the rules reported in Tab. 1. A test suite TS is defined by using the macro *BOOST_AUTO_TEST_SUITE(testSuiteName)*, 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. Each test case contains an instance *sut* of the class which the ASM is translated to. Then, for each state transition in the abstract test, the test performs in order three tasks:

1. It sets the values of monitored functions (using the assignment operator).
2. It checks the value of controlled functions by using the macro `BOOST_CHECK`.
3. It performs an ASM step by calling the `step` method in the C++ class.

After each step, the monitored locations will be changed and the controlled location will be checked again, till the end of the abstract test sequence.

3.3 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 model-to-code transformation

Abstract Test		Concrete Test

Test suite TS		`BOOST_AUTO_TEST_SUITE(testSuiteName)`
		translation of each test case in TS
		`BOOST_AUTO_TEST_SUITE_END( )`

Test case t: $s_0, s_1 \dots s_n$		`BOOST_AUTO_TEST_CASE(testCaseName) {`
		`SUTClass sut;`
		translation of each state transition in t
		`}`

ASM step $s_i \rightarrow s_{i+1}$		*set monitored locations in* s_i *check controlled locations in* s_i `sut.step();`
State	Monitored location $m = val$	`sut.m = val;`
	Controlled location $c = val$	`BOOST_CHECK(sut.c[0] == val);`

Table 1. Translation of abstract tests to concrete tests

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 method) of the class τ(A) (as explained in [4]).

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 model-to-code 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.*

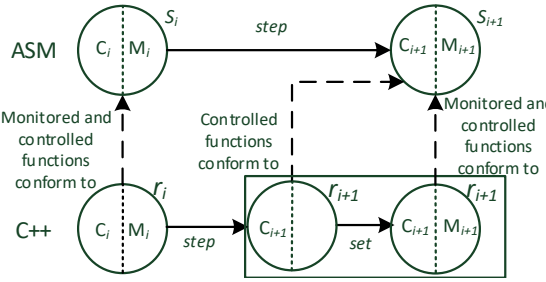

Fig. 4. ASM/C++ Conformance

Verifying the correctness of the translation τ would require the use of formal techniques like model checking or theorem proving. As shown in [1], 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 (recall the statement of Ed Brinksma). As observed in [8], 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 Sect. 3.2, and is based on the following theorem.

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 the relations between `t` and the abstract states are depicted in Fig. 4. 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 Def. 2, by executing immediately before each state the instruction `O.step()` (see Tab. 1). Then, the unit test sets the monitored variables in C++ to the values in s_{i+1} (see Tab. 1). 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 Thm. 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.

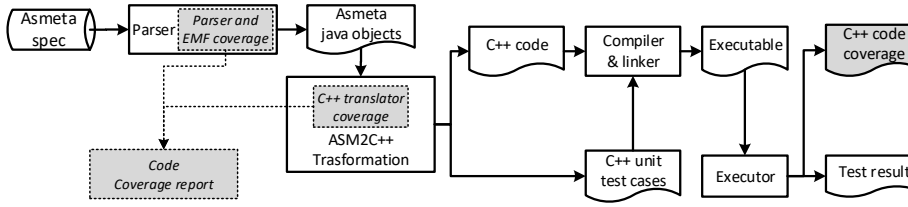

Fig. 5. Validation process

4 Validation of the transformation

In this section, we explain how we have devised a process able to validate our transformation by testing. 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 Def. 3 and is based on Thm. 1. This is a sort of *indirect* testing [1], 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 Fig. 5 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 [5] 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.

4.1 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 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 Thm. 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.

4.2 Coverage computation

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.

1. 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 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.
2. 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.
3. 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.

4.3 Tools used

To support the validation process, we have used several tools. `Ant`[8] 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.

[8] https://ant.apache.org/

To compute the java code coverage we use JaCoCo[9], 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 C++ code, we use gcov that instruments the generated C++ source code and outputs coverage information when it is executed.

4.4 Dealing with internal nondeterminism

In ASMs, 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 [14]. 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 Results

We have taken 44 ASM models taken from the public repository of the Asmeta framework[10] and we apply the validation process to each of them. The validation activity has allowed us to find and fix several faults and the coverage has given us a good indication on how to extend and improve the Asm2C++ tool.

5.1 Discovered faults

The validation process has allowed us to find faults in the transformation that have been classified into four categories:

1. *missing translation*: the translation of an ASM construct to C++ is missing;

[9] http://www.eclemma.org/jacoco/
[10] Source code and examples are available at http://asmeta.sourceforge.net/

2. *syntactically incorrect translation*: the translation to C++ is syntactically incorrect and the compiler finds the error;
3. *semantically incorrect translation*: the code is compiled by the C++ compiler, but the test cases fail because the behavior of the code does not conform to the behavior of the specification; and
4. *incorrect test case generation*: the generation of test cases is not correct.

The identification between the first two categories of errors is easy because in the first case the `Asm2C++` generator throws an exception and in the second case the compiler fails to compile the C++ code and prints an error message. The classification of errors between the third and the fourth category requires a deep analysis, because in both cases the tests fail when a conformance fault is found without providing any other information.

Missing translation. These errors are caused probably by forgetfulness or distraction of the programmer. In our case, for example, the first error reported by the `Asm2C++` tool concerned the translation of natural numbers which was missing. We have now translated natural numbers as unsigned integers.

Another fault we have discovered is the missing translation of abstract domains. We have now added a translation rule that for each abstract domain A produces a C++ class C_A, while constants in A are translated as objects. A set that contains all the objects of type C_A is also added to C_A to keep track of the static constants of A.

Syntactically incorrect translation. These errors are due to a misunderstanding of the semantics of the source notation (ASMs) and how it is translated to the target notation (C++). It can be also caused by an incomplete knowledge of the target language, C++ in our case. For instance, the compiler has found an error in the translation of *case terms*. Each case is translated as nested if-else and the otherwise clause was translated using the "otherwise" keyword which does not exist in C++, and that caused an error during compilation. This error has been resolved by replacing "otherwise" with the "else" keyword.

Semantically incorrect translation. These errors are caused because ASMs and C++ are executed differently. ASMs runs are sequences of states (rules are executed and then the functions are updated) while C++ programs execute instructions sequentially. For example, we found an error regarding the semantics of `seqRule`. In ASM specifications, rules are executed in parallel, but sometimes it is allowed to model sequential execution by means of `seqRule`. In case of a sequential block, the value of controlled functions must be updated immediately in both current and next states. This behavior had not been taken into consideration and some test cases failed.

Incorrect test case generation. These errors are caused when the test generation produces wrong test cases or when the translation from abstract test cases to concrete tests is incorrect. For example, we found an error that concerns invariants, which are constraints that must be satisfied during the ASMs

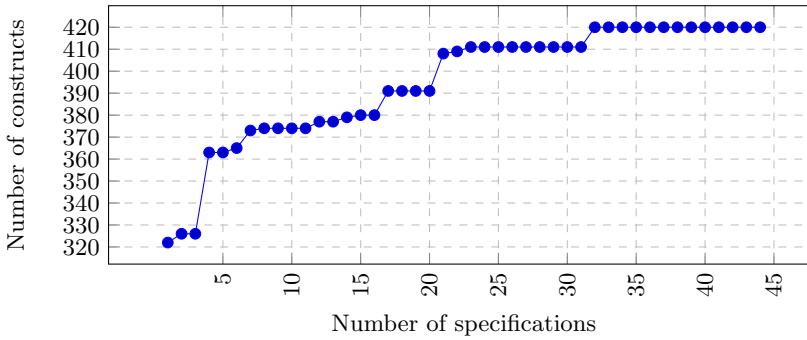

Fig. 6. Coverage of `Asmeta` parser

execution. Sometimes the expression of an invariant contains monitored functions which are chosen automatically by the test generator in order to build test cases. Initially, they were chosen randomly, but some test cases failed because the corresponding invariants could not be satisfied. To overcome this problem, we have forced the test generator to continue choosing values for monitored functions until they satisfy the corresponding invariants.

5.2 Coverage

In Sect. 4.2, we have listed the measures of code coverage one should perform during the testing of the transformation: coverage of the source language, coverage of the transformation code, and coverage of the produced code. In this section, we present the results obtained for each coverage criteria.

Fig. 6 shows the coverage of 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 overspecification should be addressed before in order to simplify the language.

Coverage of the transformation code is shown in Fig. 7. The result obtained is satisfactory because most of the code is covered, despite not all the classes are 100% covered. This is because the code contains many redundant checks in case of critical situations that should never happen.

The third coverage is about 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 ≈70%). For this reason, we have improved the translator

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%

Fig. 7. `Asm2C++` coverage

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.

6 Related work

The challenging nature of model transformation creates the need for validation of this systematic process. This need and the associated challenges have been documented by several researchers, e.g., [3,9,15]. A comprehensive survey on the related state of the art can be found in [1] and [7].

Wimmer et al. [16] present a language-agnostic approach for testing model-to-text and text-to-model transformations. They extend the Object Constraint Language 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 [8] 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 [11], Sampath et al. present a technique for verifying and validating Stateflow[11] model translation to C code. However, both these works, i.e., [8] and [11], are based on the proprietary tool Simulink[12]. Our work, on the other hand, is based on the Asmeta framework, which is an open-source project and freely available.

[11] Stateflow is a hierarchical state-machine modeling language that is part of the Simulink/Stateflow tool-suite from The MathWorks Inc.

[12] www.mathworks.com/products/simulink

In [10], Küster et al. 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.

In [13], Stümer et al. 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.

7 Conclusion

In this paper, we have presented a process to automatically validate the transformation correctness from `Asmeta` specifications to C++. The process is based on the notion of conformity between C++ code and `Asmeta` specifications, and on the definition of correctness (see Def. 3). It 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 code coverage information is collected. The coverage regards several artifacts involved in the transformation, namely the inputs (ASM specs), the transformation itself (Xtext code), and the outputs (the generated C++ code).

We have applied this process to a set of ASMs and we were able to discover several faults both within the transformation code and the test generator component. Such faults were sometimes due to the subtle misunderstanding of ASM semantics (like the `SeqRule`) that requires a peculiar translation to C++. Further, this activity has allowed us to increase the applicability of the transformation by extension to missing ASM constructs, and identification of parts of the `Asmeta` language, which are not used in practice.

References

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

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 **41**, 155–166 (2011)
3. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM **53**(6), 139–143 (Jun 2010)
4. 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. pp. 295–301. Springer International Publishing (2017)
5. Bonfanti, S., Gargantini, A., Mashkoor, A.: Generation of C++ unit tests from Abstract State Machines Specifications. In: 14th Workshop on Advances in Model Based Testing (A-MOST'18) @ICST 2018, Västerås, Sweden (2018)
6. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York, Inc. (2003)
7. Calegari, D., Szasz, N.: Verification of model transformations: A survey of the state-of-the-art. Electronic Notes in Theoretical Computer Science **292**, 5 – 25 (2013), proceedings of the XXXVIII Latin American Conference in Informatics (CLEI)
8. Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Form. Methods Syst. Des. **35**(3), 389–401 (Dec 2009)
9. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. pp. 37–54. FOSE'07, IEEE Computer Society, Washington, DC, USA (2007)
10. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations – first experiences using a white box approach. In: Kühne, T. (ed.) Models in Software Engineering. pp. 193–204. Springer, Berlin, Heidelberg (2007)
11. Sampath, P., Rajeev, A.C., Ramesh, S.: Translation validation for stateflow to c. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2014)
12. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2), 25–31 (2006)
13. Stuermer, I., Conrad, M., Doerr, H., Pepper, P.: Systematic Testing of Model-Based Code Generators. IEEE Transactions on Software Engineering **33**(9), 622–634 (Sep 2007)
14. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Tests and Proofs, pp. 134–153. Springer (2008)
15. Van Der Straeten, R., Mens, T., Van Baelen, S.: Challenges in model-driven software engineering. In: Chaudron, M.R.V. (ed.) Models in Software Engineering. pp. 35–47. Springer, Berlin, Heidelberg (2009)
16. 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. pp. 203–219. Springer, Berlin, Heidelberg (2013)