Generation of C++ Unit Tests from Abstract State Machines Specifications

Silvia Bonfanti, Angelo Gargantini
Department of Management, Information and Production Engineering
University of Bergamo
Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it

Atif Mashkoor
Software Competence Center Hagenberg GmbH
&
Johannes Kepler University Linz
Austria
atif.mashkoor@{scch|jku}.at

***Abstract*—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. Following the same approach, model-based testing suggests deriving also (unit) tests from abstract models. Previously, we have presented `Asm2C++` [1] – a tool that translates Abstract State Machines (ASMs) to C++ code. In this paper, we extend the `Asm2C++` tool such that it can now 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. We also present some experiments that prove the feasibility of the proposed approach.**

I. Introduction

The Abstract State Machines (ASM) method [2] is a formal method that is used to guide the rigorous development of software and embedded systems. The ASM-inspired development starts from an abstract specification of a system and then it goes on to capture the complete details of the system through a sequence of refinements. During this process, a designer can apply classical verification and validation techniques on models, e.g., simulation, walking through a scenario, and model checking, in order to assess the correctness of the development. The last step of the development process is the automatic generation of code artifacts.

The generation of code directly from specifications is one of the main cornerstones of the model-driven engineering paradigm [3] and also a common practice in industry. For example, Airbus uses automatic code synthesis from SCADE [4] models to generate the code for embedded controllers in the Airbus A380 [5]. The Event-B [6] method has been used in the past to generate code artifacts for hemodialysis machines [7]. The Matlab/Simulink[1] method is also a popular method in the automotive industry [8].

The code generated through a rigorous refinement process is, in principle, *correct-by-construction,* i.e., already verified and validated. However, this is not enough in case of critical systems. For such systems, the generated code should also be tested. In principle, testing an automatically generated code seems useless if both specification and the code generation process can be proven correct. In practice, however, designers want to test even the automatically generated code, in order to gain confidence that errors are not introduced inadvertently in any step (including code compiling, for example)[2]. Testing is actually a valuable and complementary adjunct to the use of formal methods in software development [9]. Moreover, tests are extremely useful for regression testing, i.e., if the code is later manually modified in order to introduce implementation details, the designer can use tests in order to check that no faults are introduced. Inspection of unit tests can reveal potential errors in specifications and eventually increases the confidence that both specification and its implementation are correct. Moreover, unit tests can be used to validate the model-to-code transformation [10]. Unit tests can also detect possible faults introduced by the model-to-code transformation process.

The automatic test generation from models has been advocated by researchers in the area of Model-Based Testing (MBT) for several years. A good introduction of this topic can be found in [11], [12]. In the MBT process, the specification is reused also for testing purposes. The main aim of this process is to substitute or at least complement other testing processes, like manual, capture/replay, or script-based ones. Instead of writing test cases, the test designer writes (or reuses) the abstract model of the system under test and then, through a sequence of steps, a MBT tool generates the test cases from the model. This approach can provide advantages in terms of cost and test effectiveness [11].

In this paper, we present the extension of our code generation framework by introducing a technique that generates unit test cases for the code generated from a formal model given as an ASM. First, some abstract tests are generated from the ASM of the system. To do that, we can use either test cases generation tools like ATGT [13] which exploits a model checker, or use the ASM simulator [14] in order to obtain

This work is partially 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.

[1]https://www.mathworks.com/products/simulink.html

[2]Ed Brinksma said in his 2009 keynote at the Testcom/FATES conference: ”Who would want to fly in an airplane with software formally verified but never tested?”. We can rephrase his statement in this way: ”Would you fly in an airplane with automatically generated code that has never been tested?”

traces of abstract states. Then, we translate the sequences of abstract states into C++ unit tests. This translation leverages the model-to-code translation we implemented in the Asm2C++ tool. The resulting unit tests are C++ tests in the BOOST framework[3].

The paper is organized as follows: In Section II, the ASM method is briefly introduced along with its associated toolset - the Asmeta framework. In Section III, we introduce the example that is subsequently used in the paper to explain the code and the test case generation process. The code generation process is explained in Section IV along with an analysis of the generated C++ code. Section V describes the unit tests generation process using the illustrative example. In Section VI, we compare two test generators tested on some ASM specifications. Section VII reports on the related work. The limitations of our approach are listed in Section VIII and the future works to overcome these limits are discussed in Section IX.

II. Abstract State Machines

A. The ASM language

Abstract State Machines (ASMs) [2] are an extension of Finite State Machines (FSMs), where unstructured control states are replaced by states with arbitrarily complex data. ASM *states* are algebraic 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 couple (*location*, *value*) represents a machine memory unit. Therefore, ASM states can be viewed as abstract memories.

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 rules 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* is, therefore, defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which in turn could fire other transitions rules. An ASM can have more than one *initial state*.

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. It is possible to specify state *invariants*.

An ASM can be *nondeterministic* due to the presence of monitored functions (*external* nondeterminism) and of choose rules (*internal* nondeterminism).

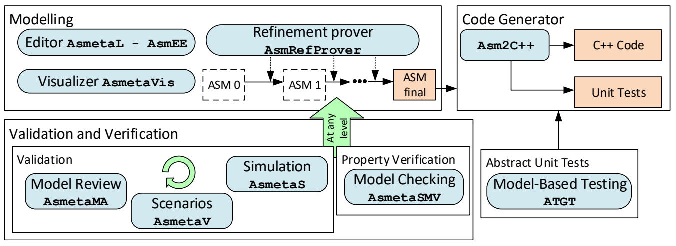

Fig. 1. The ASM development process powered by the Asmeta framework

B. The ASM methodology

The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Fig. 1 shows the development process based on ASMs. The process is supported by the Asmeta (ASM mETAmodeling) framework[4] [15] which provides a set of tools to help the developer in various activities. The first step is modeling the system using the language AsmetaL. Starting from the first model (also called the ground model), the developer refines it until the final ASM is reached. During the refinement steps, the tool AsmRefProver ensures whether the current ASM model is the correct refinement of the previous ASM model. During the modeling, the user is supported by the editor AsmEE and the visualizer AsmetaVis which transforms the textual model into a graphical notation. The graphical notation helps the developer to better understand the behavior of the textual model. At any level of refinement validation and property verification activities can be applied. The validation of the model can be achieved either through the model simulator AsmetaS, through scenarios AsmetaV, or through the model reviewer AsmetaMA. The simulator AsmetaS allows to perform two types of simulation: interactive simulation (the user inserts the values of functions that depends on the environment) or random simulation (the tool randomly chooses the values of functions that depend on the environment). Scenarios can be written using the Avalla language and they contain the expected system behaviors. The scenarios are executed using the AsmetaV tool to check whether the machine runs correctly. The model reviewer AsmetaMA performs static analysis – it determines whether a model has sufficient quality attributes (e.g., minimality, completeness, consistency). The verification tool AsmetaSMV checks whether the properties derived from the requirements document comply with the behavior of the model.

When the final ASM model is reached, the Asm2C++ tool automatically translates the ASM specification into C++ code

[3]http://www.boost.org/

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

(see Section IV). In this paper, we have extended the Asm2C++ tool such that now it translates the abstract tests – generated, for example, by using ATGT tool – to concrete tests (see Section V).

III. Illustrative Example

A. The ACVM example

The process from modeling to unit test generation, going through code generation is explained in the paper using the Automatic Coffee Vending Machine (ACVM) example. The example chosen is not too complicated because we want to show all the code including the ASM model, the code generated by the Asm2C++ tool, and the unit tests.

The ACVM distributes coffee, tea, and milk and it accepts only 50 cents and 1 Euro coins. If the user inserts 50 cents the machine distributes milk (if it is available); if the user inserts 1 Euro the machine distributes coffee or tea (if they are available) based on user choice. If a drink is distributed then its availability is decremented and the money is preserved into the machine. At the beginning, the machine has 10 coffees, 10 teas and 10 milks. The machine can contain 25 coins at maximum, when this limit is reached the machine does not distribute products any longer.

B. The ASM model for the ACVM example

The ACVM example is modeled using a refinement-based approach and each refinement step is amenable to validation and verification activities. Once the final ASM model (see Code 1) is reached, the Asm2C++ tool is applied.

For brevity, we skip here the refinement steps and only explain the final ASM model. Four domains are defined in the ACVM ASM model:

- CoinType: enumerative domain to represent the coins accepted by the machine (HALF for 50 cents, ONE for 1 Euro).
- Product: enumerative domain to represent the products available at the coffee machine (COFFEE, TEA, MILK).
- QuantityDomain: static domain to represent the quantity of products available: from 0 (the product is not available) to 10 (the maximum quantity of the product available in the machine).
- CoinDomain: static domain to represent the number of coins inside the machine: from 0 (no coins are inside the machine) to 25 (the maximum number of coins contained by the machine).

Two monitored functions are defined: insertedCoins as CoinType to read the coin inserted by the user and chosenProduct as Product to read the product selected by the user. The number of coins inside the machine and the availability of the product are represented by two controlled functions: coins as CoinDomain and available as QuantitativeDomain subjected to its input (Product type). The domains declared in the signature are static and their definition is performed inside the definition section. The behavior of the machine is modeled in the main rule, while the activity of serving the product is modeled in the rule (r_serveProduct). The init section contains the initialization of coins and available functions. The initial number of coins inside the machine is 0, while the availability of each product is 10.

```
asm coffeeVendingMachine
import STDL/StandardLibrary

signature:
  enum domain CoinType = {HALF | ONE}
  enum domain Product = {COFFEE | TEA | MILK}
  domain QuantityDomain subsetof Integer
  domain CoinDomain subsetof Integer
  controlled available: Product −> QuantityDomain
  controlled coins: CoinDomain
  monitored insertedCoin: CoinType
  monitored chosenProduct: Product

definitions:
  domain QuantityDomain = {0 .. 10}
  domain CoinDomain = {0 .. 25}

  rule r_serveProduct($p in Product) =
    par
      available($p) := available($p) − 1
      coins := coins + 1
    endpar

main rule r_Main =
  if(coins < 25) then
    if(insertedCoin = HALF) then
      if(available(MILK) > 0) then
        r_serveProduct[MILK]
      endif
    else if chosenProduct != MILK then
        if (chosenProduct = TEA and available(TEA)>0) then
          r_serveProduct[TEA]
        else
        if (chosenProduct = COFFEE and
             available(COFFEE)>0) then
          r_serveProduct[COFFEE]
        endif
      endif
      endif
    endif
  endif

default init s0:
function coins = 0
function available($p in Product) = 10
```

Code 1. CoffeeVendingMachine ASM model

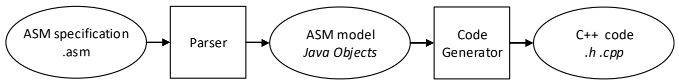

Fig. 2. Code Generation Process: from ASM to C++

IV. Code generation

In [1], we presented the tool Asm2C++, which translates ASMs to C++ code. Asm2C++ is based on Xtext[5]– a framework for the development of domain-specific languages –, which provides facilities to parse and translate specifications.

The process in Fig. 2 shows how an ASM specification is translated into C++ code. The generated code is composed of a

[5]https://www.eclipse.org/Xtext/

header (*.h*) and a source (*.cpp*) file. The header file contains the translation of the ASM signature while the source file defines how the ASM evolves by translating each ASM rule to a C++ method.

The header and the source files of the ACVM example – automatically generated by the Asm2C++ tool – are shown in Code 2 and Code 3. In the header file (see Code 2), we can identify several parts: *domain declaration*, *domain container declaration*, *function declaration*, and *rule declaration*. In the domain declaration section, the domains defined in the ASM file are translated (CoinType, Product, QuantityDomain and CoinDomain). Enum, static and dynamic domains are defined into the *namespace* using the keyword *enum* for enum domains and *typedef* for static and dynamic domains; furthermore, enum domains are initialized. Inside the class, domain containers are defined (one for each domain defined in the domain declaration section) which contain domains elements. The keyword *const* identifies static domains (no elements are added or removed from the domain during ASM execution), if it is omitted the container is relative to dynamic domain (during an ASM execution elements are added or removed from the domain). In the example, there are four containers and they are represented using the set container which stores unique elements (inside an ASM domain duplicate values are not admitted). The function declaration section contains the declaration of all the functions of the ASM specification. Each function declared in the ASM file is translated as a variable in C++ except for controlled functions. Each controlled function is translated to an array of two elements, the first is the value of the function at state S_n, while the second element contains the value assumed at state S_{n+1}. Rules are translated as methods in C++ and prototypes are defined in the rule declaration section.

The source file (see Code 3) is divided in four parts: *method implementation*, *function definition*, *function and domain initialization*, and *updateSet*. Each method defined in the header file is implemented in the method implementation section based on the behavior defined by the ASM specification. Functions defined in the ASM definition section are implemented in the function definition section (which is empty in the example). The functions and the domains (dynamic domains) initialized in the init section of the specification are added to the function and domain initialization section of the source file. These elements are included into the C++ constructor to guarantee that the initialization is performed at the beginning of the program execution. An ASM execution corresponds to a sequence of steps, during each step the functions do not change their values until the update set is applied (at the end of each step). To guarantee the same behavior in C++, we introduce an array of two elements for each controlled function (as previously explained), the first element is the value of function at state S_n, while the second element contains the value assumed by the function in state S_{n+1}. At the end of the step, the value of the next state is assigned to the current state using the *fireUpdateSet()* method. The method updates the value of the controlled function to the next state value. After that, the next state can be computed.

```
#ifndef coffeeVendingMachine_H
#define coffeeVendingMachine_H

#include<string>
typedef std::string String;
#include<iostream>
using namespace std;
#include <set>
#include <map>
#define ANY String

/* Domain declaration */
namespace coffeeVendingMachinenamespace{
 enum CoinType {HALF, ONE};
 enum Product {COFFEE, TEA, MILK};
 typedef int QuantityDomain;
 typedef int CoinDomain;
}
using namespace coffeeVendingMachinenamespace;
class coffeeVendingMachine{
 /* Domain containers declaration */
  const std::set<CoinType> CoinType_elems;
  const std::set<Product> Product_elems;
  const std::set<QuantityDomain> QuantityDomain_elems;
  const std::set<CoinDomain> CoinDomain_elems;
 public:
 /* Function declaration */
  std::map<Product, QuantityDomain> available[2];
  CoinDomain coins[2];
  CoinType insertedCoin;
  Product chosenProduct;
 /* Rule declaration */
  void r_serveProduct (Product _p);
  void r_Main();
  coffeeVendingMachine();
  void fireUpdateSet();
};

#endif
```

Code 2. coffeeVendingMachine.h code

V. Generation of unit tests from ASMs

The complete process for MBT applied to ASMs we devise in this paper is presented in Fig. 3. The user starts with an ASM specification that is already validated and verified. By applying the Asm2C++ transformation presented in Section IV, 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. We currently support two main ways: random generation by using the Asmeta simulator and coverage-driven test generation by model checking [13]. The translation of abstract tests to concrete tests is done by the C++ unit tests builder, which follows transformation rules we will present in Section V-B. 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.

```
//coffeeVendingMachine.cpp automatically generated
#include "coffeeVendingMachine.h"
using namespace coffeeVendingMachinenamespace;
// Conversion of ASM rules in C++ methods
void coffeeVendingMachine::r_serveProduct (Product _p){
  {
   available[1][_p] = (available[0][_p] − 1);
   coins[1] = (coins[0] + 1);
  }
}
void coffeeVendingMachine::r_Main(){
  if ((coins[0] < 25)){
   if ((insertedCoin == HALF)){
    if ((available[0][MILK] > 0)){
   r_serveProduct(MILK);
    }
   }else if ((chosenProduct != MILK)){
   if ((chosenProduct == TEA) & (available[0][TEA] > 0)){
    r_serveProduct(TEA);
   }else if ((chosenProduct == COFFEE)
                              & (available[0][COFFEE] > 0)){
     r_serveProduct(COFFEE);
   }
    }
  }
}
// Function and domain initialization
coffeeVendingMachine::coffeeVendingMachine():
// Static domain initialization
QuantityDomain_elems(std::set<int>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}),
CoinDomain_elems(std::set<int>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}),
CoinType_elems({HALF,ONE}),
Product_elems({COFFEE,TEA,MILK})
{
  //Function initialization
  coins[0] = coins[1] = 0;
  for(auto const& _p : Product_elems){
   available[0].insert({_p,10});
   available[1].insert({_p,10});
  }
}
// Apply the update set
void coffeeVendingMachine::fireUpdateSet(){
  available[0] = available[1];
  coins[0] = coins[1];
}
```

Code 3. coffeeVendingMachine.cpp code

To be more precise, the complete generation of C++ unit tests from ASM specification, requires at least two steps:

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

A. 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 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,

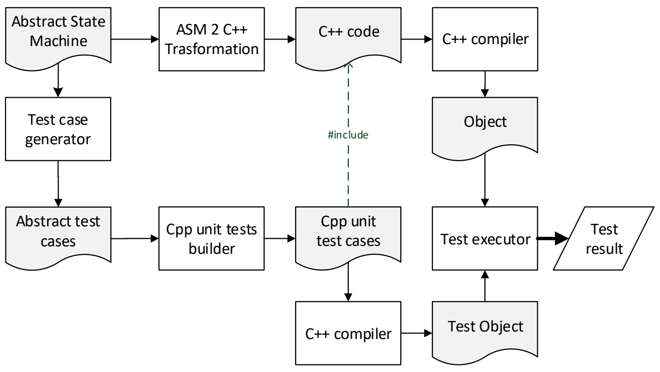

Fig. 3. Generation of unit tests from ASMs

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 [13]. 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 model checker generally requires more time, since it must perform the exploration of the whole state space. One could use techniques for model decomposition [16], but still the model checker we use, has some limits, like it cannot accept infinite domains.

B. 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. 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 I.

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

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.mainRule();` `sut.updateSet();`

TABLE I
TRANSLATION OF ABSTRACT TESTS TO CONCRETE TESTS

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 method in C++ (that corresponds to the main rule in ASM) and after that the updateSet is applied in order to obtain the next state.

C. Test generation of the ACVM example

Code 4 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.

```
BOOST_AUTO_TEST_SUITE(TestcoffeeVendingMachine)

 BOOST_AUTO_TEST_CASE( my_test_0 ){
  // instance of the SUT
  coffeeVendingMachine sut;
  // check state
  BOOST_CHECK(sut.available[0][COFFEE]==10);
  BOOST_CHECK(sut.available[0][TEA]==10);
  BOOST_CHECK(sut.available[0][MILK]==10);
  BOOST_CHECK(sut.coins[0]==0);
  // set monitored variables
  sut.chosenProduct=COFFEE;
  sut.insertedCoin=HALF;
  // call main rule
  sut.r_Main();
  sut.fireUpdateSet();
 }

 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
  coffeeVendingMachine sut;
  // check state
  BOOST_CHECK(sut.available[0][COFFEE]==10);
  BOOST_CHECK(sut.available[0][TEA]==10);
  BOOST_CHECK(sut.coins[0]==0);
  BOOST_CHECK(sut.available[0][MILK]==10);
  // set monitored variables
  sut.chosenProduct=COFFEE;
  sut.insertedCoin=HALF;
  // call main rule
  sut.r_Main();
  sut.fireUpdateSet();
  // check state
  BOOST_CHECK(sut.available[0][TEA]==10);
  BOOST_CHECK(sut.available[0][MILK]==9);
  BOOST_CHECK(sut.coins[0]==1);
  BOOST_CHECK(sut.available[0][COFFEE]==10);
  ...
 }

BOOST_AUTO_TEST_SUITE_END()
```

Code 4. CoffeeVendingMachine test suite example

VI. EXPERIMENTS

In order to test our approach, we have selected 8 ASM specifications from the Asmeta repository[6]. These benchmarks include 4 small examples (AdvancedClock, coffeVendingMachine, ferryman, and Safety Injection System - SIS) and 4 case studies taken from our previous projects: the specification for a hemodialysis machine (Hemodialysis_ref4) [17], the ground model (LGS_GM) and refined specification (LGS_3L) of a landing gear system [18], and the specification of a medical software component (StereoAcuityCertifier) [19].

For every specification, we have generated the test cases by using different test generators and options, translated the specification and the tests to C++, and executed the tests in order to check that our process was able to successfully test the system implementation under test. During testing, we performed the coverage evaluation of the C++ code of the main C++ file by using the `gcov` tool. We use the coverage as a measure of the quality of the produced tests. For each run, we recorded the total number of test cases, the total number of steps and the maximum length of the tests. We measured also the total time required for test generation and execution. The results are shown in Table II.

Regarding the choice of the abstract test generator, in order to compare the two proposed techniques – NuSMV and the simulator – we proceeded in the following way. We first executed NuSMV and if it completed successfully, then we asked the simulator to generate two test suites both with the same number of tests as generated by NuSMV: the first one with around the same total number of tests and the second one with the same max number of steps. If NuSMV was not able to generate the test suite, then we asked the simulator to generate 10 tests with 1000 steps each.

For 3 specifications (ferryman, Hemodialyis, and LGS_3L),

[6]https://sourceforge.net/p/asmeta/code/HEAD/tree/asm_examples/

ASM specification	Test Generator	Tot. steps	#tests	max length	time (millisec)	coverage	errors
AdvancedClock	NuSMV	3661	7	3600	48693	100.0	
	Simulator	3661	7	523	20229	95.24	
	Simulator	25200	7	3600	134916	100.0	
coffeeVendingMachine	NuSMV	56	7	26	7331	100.0	
	Simulator	63	7	9	4950	100.0	
	Simulator	182	7	26	5802	100.0	
ferryman	NuSMV	0	0	0	3658	N/A	no test generated
	Simulator	10000	10	1000	111864	100.0	
SIS	NuSMV	88	32	6	15914	94.44	
	Simulator	128	32	4	4624	90.28	
	Simulator	192	32	6	5152	87.5	
Hemodialysis_ref4	NuSMV	0	0	0	11227	N/A	no test generated
	Simulator	1000	10	100	160984	68.3	
	Simulator	10000	10	1000	31929	N/A	out of memory C++ compiler
LGS_GM	NuSMV	223	8	56	23255	100.0	
	Simulator	448	8	56	7232	100.0	
	Simulator	232	8	29	4426	100.0	
LGS_3L	NuSMV	0	0	0	1893	N/A	no test generated
	Simulator	10000	10	1000	476615	N/A	out of memory C++ compiler
	Simulator	11	1	11	2066	N/A	out of memory C++ compiler
StereoAcuityCertifier	NuSMV	364	80	9	42712	97.6	
	Simulator	720	80	9	15286	90.4	
	Simulator	480	80	6	8437	90.4	

TABLE II
EXPERIMENTAL RESULTS

NuSMV was not able to generate tests as all the specifications contain some constructs not supported by the translation feature of the model checker. For the largest specification (LGS_3L) of these, the simulator was able to generate tests that were translated to unit tests, but then the compiler was unable to build and execute those tests. We plan to adopt some techniques able to reduce the size of unit tests, like limiting the CHECK commands only to some controlled variables.

For 3 simple specifications (AdvancedClock, coffeVendingMachine, and LGS_GM), both test generator policies obtained 100% coverage of the generated code. However, as expected, NuSMV was able to obtain the same level of coverage with much shorter test sequences but requiring more time.

For 2 specifications (SIS and StereoAcuityCertifier), both test generators produced unit tests, which, however, were not able to cover all the implementation code. In both cases, NuSMV achieved a good level of coverage with a fewer total number of steps.

Although we applied our technique only to a limited set of ASMs, the experiments confirm the feasibility of our approach. As expected, a coverage-driven generation by model checking produces better test cases than random test sequences. However, it pays the price in terms of applicability and requires more time resources.

VII. RELATED WORK

The paper by Dick et al. [20] is one of the pioneering works that promoted the use of formal methods in software testing. The authors used the VDM method [21] for the generation of test cases, their sequencing, and the test oracle.

In another work [22], authors generate test cases by performing symbolic execution over a B model [23], 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 in [24]. 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 [25] specifications using Constraint Logic Programming.

Engel et al. [26] presented a method for automatic generation of self-contained unit tests in the JUnit format[7]. The implementation is based on the verification system KeY [27] and supports the JAVA CARD programming language [28]. The approach exploits the implementation of a system and does not necessarily require its detailed formal specification.

In [29], authors 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)[8] and the JUnit testing framework.

In [30], authors present an approach to automatically generate test cases through model transformations. Their work takes as input UML 2.0 sequence diagrams [31] and automatically derive test cases scenarios that conforms the UML Testing

[7]http://junit.org
[8]http://www.eecs.ucf.edu/~leavens/JML//index.shtml

Profile[9]. 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 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++` translates the specification in C++ code and generates C++ unit tests to verify the correct behaviour of the generated C++ code.

VIII. Limitations

Despite the effort to cover all the ASM constructs, we still have some limitations in test cases generation due to the following factors:

- For specifications with infinite domains, only the random approach provided by the simulator can be used. Moreover, the current translation to NuSMV does not support some specific domains (like Strings) and complex terms (like lists).
- The current translation to C++ [1] does not support some ASM constructs (like abstract domains).
- ASMs can express *internal nondeterminism* by means of the `choose` rule which randomly chooses an element in a domain and then some actions are performed. Both the test generation and the `Asm2C++` translation support this kind of rule. In C++, the rule is translated using a random method that automatically selects a value from the limited domain. Despite that, our technique is not suitable to be used in the presence of internal nondeterminism. This is because if the value chosen by the test generator is different as compared to the value chosen by C++ compiler, the behavior of ASM and C++ will not be the same and the test will fail. We plan to work on this issue in two directions: either by combining runtime monitoring with testing (as suggested in [32]) or by forcing the choice of the same elements during test generation and test execution. Currently, the only solution is to transform a `choose` rule into a monitored variable, converting the nondeterminism from internal to *external*.

IX. Conclusion and Future Work

In this paper, we have presented an extension of the `Asm2C++` tool that enhances the capability of the tool by supporting the production of concrete unit tests. The unit tests are generated automatically starting from the ASM model by following the process shown in Fig. 3. 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++`. In parallel, the tool also translates the ASM specification into C++ code. Once the test cases and the C++ code are available, the test executor runs the test cases and verifies the test results. The process has been validated using an illustrative example based on an automatic coffee vending machine (see Section III). Furthermore, in Section VI, we have performed some experiments to check the applicability of our approach and we have compared two techniques adopted to generate abstract tests. 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.

As listed in Section VIII, our approach suffers from some limitations. In order to overcome those limits, we plan the following future work. We intend to manage the internal nondeterminism by finding a solution to assign the same value to the test generator and to the C++ compiler. Furthermore, we will work on the translation of further ASM constructs to include a larger number of ASM specifications.

[9]http://utp.omg.org/

References

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

[2] E. Börger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., 2003.

[3] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MC.2006.58

[4] F.-X. Dormoy, "Scade 6: a model based solution for safety critical software development," in *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS08)*, 2008, pp. 1–9.

[5] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[6] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[7] A. Mashkoor and M. Biro, "Towards the trustworthy development of active medical devices: A hemodialysis case study," *IEEE Embedded Systems Letters*, vol. 8, no. 1, pp. 14–17, 2016.

[8] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from Matlab/Simulink for critical applications," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2014, pp. 1–6.

[9] J. Rushby, "Automated test generation and verified software," in *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, B. Meyer and J. Woodcock, Eds. Berlin, Heidelberg: Springer, 2008, pp. 161–172. [Online]. Available: https://doi.org/10.1007/978-3-540-69149-5_18

[10] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper, "Systematic Testing of Model-Based Code Generators," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 622–634, Sep. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4288195/

[11] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[12] P. C. Jorgensen, *The Craft of Model-Based Testing*. Auerbach Publications, 2017.

[13] A. Gargantini and E. Riccobene, "ASM-based testing: Coverage criteria and automatic test sequence generation," *JUCS - Journal of Universal Computer Science*, vol. 7, no. 11, pp. 1050–1067, nov 2001.

[14] A. Gargantini, E. Riccobene, and P. Scandurra, "A Metamodel-based Language and a Simulation Engine for Abstract State Machines," *J. UCS*, vol. 14, no. 12, pp. 1949–1983, 2008.

[15] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra, "A model-driven process for engineering a toolset for a formal method," *Software: Practice and Experience*, vol. 41, pp. 155–166, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1019

[16] P. Arcaini, A. Gargantini, and E. Riccobene, "Decomposition-based approach for model-based test generation," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2017.

[17] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene, "Integrating formal methods into medical software development: The ASM approach," *Science of Computer Programming*, jul 2017. [Online]. Available: https://doi.org/10.1016/j.scico.2017.07.003

[18] P. Arcaini, A. Gargantini, and E. Riccobene, "Rigorous development process of a safety-critical system: from ASM models to java code," *International Journal on Software Tools for Technology Transfer*, pp. 247–269, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10009-015-0394-x

[19] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene, "Formal validation and verification of a medical software critical component," in *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, Sept 2015, pp. 80–89.

[20] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *FME '93: Industrial-Strength Formal Methods: First International Symposium of Formal Methods Europe Odense, Denmark, April 19–23, 1993 Proceedings*, J. C. P. Woodcock and P. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 268–284. [Online]. Available: https://doi.org/10.1007/BFb0024651

[21] C. B. Jones, *Systematic software development using VDM*. Prentice Hall Englewood Cliffs, 1990, vol. 2.

[22] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh, "Automatic testing from formal specifications," in *Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, Y. Gurevich and B. Meyer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 95–113. [Online]. Available: https://doi.org/10.1007/978-3-540-73770-4_6

[23] J.-R. Abrial, *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[24] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "BZ-TT: A tool-set for test generation from Z and B using constraint logic programming," in *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR*, vol. 2, 2002, pp. 105–120.

[25] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988, vol. 3.

[26] C. Engel and R. Hähnle, "Generating unit tests from formal proofs," in *Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, Y. Gurevich and B. Meyer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 169–188. [Online]. Available: https://doi.org/10.1007/978-3-540-73770-4_10

[27] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

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

[29] Y. Cheon and G. T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," in *ECOOP 2002 — Object-Oriented Programming: 16th European Conference Málaga, Spain, June 10–14, 2002 Proceedings*, B. Magnusson, Ed. Berlin, Heidelberg: Springer, 2002, pp. 231–255. [Online]. Available: https://doi.org/10.1007/3-540-47993-7_10

[30] B. P. Lamancha, P. Reales, M. Polo, and D. Caivano, "Model-driven test code generation," in *Evaluation of Novel Approaches to Software Engineering: 6th International Conference, ENASE 2011, Beijing, China, June 8-11, 2011. Revised Selected Papers*, L. A. Maciaszek and K. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 155–168. [Online]. Available: https://doi.org/10.1007/978-3-642-32341-6_11

[31] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Pearson Higher Education, 2004.

[32] P. Arcaini, A. Gargantini, and E. Riccobene, "Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism," in *The 9th Workshop on Advances in Model Based Testing (A-MOST 2013) - IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg*, ser. ICSTW'13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 178–187. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2013.29