Generation of Behavior-Driven Development C++ Tests from Abstract State Machine Scenarios*

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. In this paper, we present the AsmetaVBDD tool that automatically translates the scenarios written in the AValLa language (used by the asmeta validator (AsmetaV)) into Behavior-Driven Development scenarios for C++.

1 Introduction

The Behavior-Driven Development (BDD) is considered as the evolution and extension of the Test-Driven Development (TDD) [12]. It is increasingly being used to improve the code quality and reducing error rates in software. It aims at writing automated acceptance tests that represent complex system stories or *scenarios*. BDD builds upon TDD by requiring testers to write acceptance tests describing the behavior of the system from customers' point of view. While classical unit tests focus more on checking internal functionalities of classes, BDD testers take care to write tests as examples that anyone from the development team can read and understand [13]. BDD is currently supported at the level of code by several tools like Cucumber [13] for Java, PHP and C#, or Catch2 for C++[4].

The use of scenarios is common not only at the code level but also at the level of (abstract) models. The scenario-based techniques have been applied in different research areas and a variety of definitions, modes of use, and interaction mechanisms with users are given. In particular, scenarios have been used in the area of software engineering [1,10], business process reengineering [2], and user interface design [9]. The author in [8] classifies scenarios according to their use in the systems development ranging from requirements analysis, user-designer communication, examples to motivate design rationale, envisioning (imagined use of a future design), software design (examples of behavior thereof), to implementation, training, and documentation.

* 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.

[4] https://github.com/catchorg/Catch2

In the past, we have introduced the idea of using scenarios for validating Abstract State Machines [6] and developed a language AVALLA (and a corresponding tool) [7] for writing scenarios, which is integrated into the ASMETA framework [3]. 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.

Although most ASMETA tools work at the abstract specification level, ASMETA also supports the automatic generation of C++ code [4] and of unit tests [5]. In a classical model-driven engineering approach [11], 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 paper, we extend the ASMETA framework with a translator, called ASMETAVBDD, which translates an abstract scenario written in the AVALLA language to the BDD code. The paper is organized as follows. Sect. 2 presents some background about BDD, AVALLA, and the translation process form ASM specifications to C++. The translation from AVALLA to BDD code is presented in Sect. 3. The paper is concluded in Sect. 4 with the proposed future work.

2 Background

In this section, we present the framework we use for BDD at the level of code along with the AVALLA language and its use. However, we first introduce a simple example to show the output obtained by translating the AVALLA scenario to BDD.

The Lift example As a case study, we take part of a simple example of lift from [7]. The lift has for each floor one button, which, if pressed, causes the lift to visit (i.e., move to and stop at) that floor. A lift without requests should remain at its final destination and await further requests. The call of the lift is modeled in the ASM specification as a monitored function calledAtFloor: Integer -> Boolean, while the state of the lift is modeled by three controlled functions: floor that contains the floor number where the lift cabin is, state that represent whether the cabin is moving, and direction that shows the direction of a moving cabin.

2.1 BDD for C++

The are several frameworks for BDD in C++. One of the most powerful is Catch2. Catch2 is a testing framework for C++ that supports unit testing by means of macros. Moreover, it allows to write tests as a nested series of Given-When-Then statements in the style of BDD. In addition to the classic style for writing test cases, Catch2 supports an alternative syntax that allows to write tests as *executable specifications* in a classical BDD style. This set of macros include:

```
SCENARIO( scenario name )
```

that signals the start of a scenario/test case. Other macros include:

```
#include "catch.hpp"

SCENARIO("lift is called") {
 GIVEN("A lift at ground level") {
  Lift lift;
  REQUIRE(lift.floor == 0);
  REQUIRE(lift.state == STOP);

  WHEN("the lift is called at floor 4") {
   lift.calledAtFloor(4);
   THEN( "the lift start moving" ) {
    REQUIRE( v.state == MOVING );
    REQUIRE( v.direction == UP );}}
    ....
}}
```

Fig. 1: A simple Catch2 BDD test

```
scenario liftstarts
// load ASMETA specification
load Lift.asm
// check initial state
check floor = 0;
check state = STOP;
// lift is called at floor 4
set calledAtFloor(4) := true;
// perfrom a step
step
// lift is moving upward
check state = MOVING;
check direction = UP;
check floor = 1;
// ...
```

Fig. 2: A simple AVALLA scenario

```
GIVEN( something )
WHEN( something )
THEN( something )
```

Fig. 1 shows an example of a scenario written in Catch: when the lift is called to the fourth floor from the ground floor, then it starts moving upwards.

2.2 AVALLA

In [7], we have introduced a domain specific language, called AVALLA, to be used by the designer to manually describe scenarios (see Tab. 1). A Scenario represents a scenario of a provided ASM specification. Basically, a scenario has a name, a *spec* denoting the ASM specification to validate, and a list of target commands of type Command. A Command and its concrete sub-classes provide a classification of scenario commands. The Set command updates monitored or shared function values that are supplied by the user actor as input signals to the system. Command Step represents the reaction of the system, which executes one single ASM step. The Check class represents commands supplied by the user actor to inspect external property values and/or by the observer actor to further inspect internal property values in the current state of the underlying ASM. Finally, an Exec command executes an ASM transition rule when required by the observer actor. AVALLA supports also invariants of scenarios and the semantics of the language is given in terms of an ASM itself, so to execute a scenario ASMETA uses the ASM simulator. An example of an AVALLA scenario representing the same behavior of the C++ code is reported in Fig. 2.

Abstract syntax	Concrete syntax
Scenario	**scenario** name **load** spec_name $[C_1 \dots C_m]$
where C_j are commands: Set, Exec, Step, or Check	
Set	**set** loc := value;
where loc is a location term for a monitored function, and value is a term denoting a possible value for the underlying location	
Step	**step**
perform a machine step (compute update set and apply it to the current state)	
Check	**check** expr;
where expr is a boolean-valued term made of function and domain symbols of the ASM	
Exec	**exec** rule;
where rule is a rule of the underlying ASM	

Table 1: The AVALLA concepts and their textual notation

2.3 ASMETA to C++

The translation from ASMs to C++, performed by the tool `Asm2C++`, has been presented in [4]. We recollect here some notions that will be used in the next section. Every ASM X is translated to a class C_X in which the monitored and controlled functions are translated to C++ fields of C_X. A step in the ASM is translated to C++ as a call of two functions: one representing the main rule, and the other one, called `updateState()`, applies the update set to the controlled part of the state.

3 Generation of BDD tests from AVALLA

The Catch2 testing framework 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 [4] and the generation of C++ tests [5] already supported by ASMETA. Our translator is defined as a Model-To-Text transformation (we use Xtend[5] to define it). It takes an AVALLA scenario and produces the C++ code. Table 2 summarizes the transformation rules we have defined, which are briefly described here:

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++`.

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

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.r_main();`
	`  X.updateState();`
	`}`

exec rule	add method definition `rule()` and call it

Table 2: Translation of AVALLA constructs to Catch2 macros

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 `updateState()` 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 rules above, 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` check 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.

4 Conclusions and future work

In this paper, we have presented an approach in which BDD tests are automatically built from AVALLA scenarios. The approach is also augmented by a prototype tool ASMETAVBDD. However, not all of the AVALLA constructs are currently supported by the tool. For instance, we do not currently take into ac-

count blocks. In the future, we plan to extend our tool in order to be able to translate any AValLa scenario. Moreover, most of the textual information in the BDD scenario is generated automatically from the corresponding AValLa scenario but it may not be very informative. To add specific information, we plan to extend the translator such that it can also read the comments in the AValLa scenario, understand the commands they refer to, and translate them into BDD scenario. Currently, the comments are simply skipped since the AValLa parser just ignores them. In this way, we loose some valuable information we already have in the abstract scenario. Furthermore, we plan to develop a feature that automatically translates a BDD scenario to an AValLa scenario. This is useful for stakeholders involved in the validation process who do not know the AValLa language. They can write scenarios using their preferred BDD tool and AsmetaVBDD automatically translates them into AValLa scenarios.

Acknowledgments We would like to thank Andrea Spalluzzi who has developed the first version of the translator during his master thesis.

References

1. Anderson, J.S., Durney, B.: Using scenarios in deficiency-driven requirements engineering. In: Proceedings of the International Symposium on Requirements Engineering. pp. 134–141. IEEE (1993)
2. Antón, A.I., McCracken, W.M., Potts, C.: Goal decomposition and scenario analysis in business process reengineering. In: Advanced Information Systems Engineering. pp. 94–104. Springer Berlin Heidelberg (1994)
3. 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)
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) @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. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ Conference, September 16-18, 2008, London, UK. Lecture Notes in Computer Science, vol. 5238, pp. 71–84. Springer (2008)
8. Carroll, J.M.: Five reasons for scenario-based design. Interacting with Computers **13**(1), 43–60 (2000)
9. Carroll, J.M., Rosson, M.B.: Getting around the task-artifact cycle: How to make claims and design by scenario. ACM Transactions on Information Systems **10**(2), 181–212 (Apr 1992)
10. Potts, C., Takahashi, K., Antón, A.I.: Inquiry-based requirements analysis. IEEE Software **11**(2), 21–32 (Mar 1994)
11. Schmidt, D.C.: Model-driven engineering. Computer **39**(2), 25–31 (Feb 2006). https://doi.org/10.1109/MC.2006.58

12. Solis, C., Wang, X.: A study of the characteristics of behaviour driven development. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 383–387. IEEE (aug 2011)
13. Wynne, M., Hellesøy, A.: The Cucumber Book Behaviour-Driven Development for Testers and Developers. The Pragmatic Programmers, LLC (2012)