Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino

Silvia Bonfanti[1,2](✉), Marco Carissoni[1], Angelo Gargantini[1], and Atif Mashkoor[2]

[1] Università degli Studi di Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it, m.carissoni1@studenti.unibg.it
[2] Software Competence Center Hagenberg GmbH, Hagenberg im Mühlkreis, Austria
atif.mashkoor@scch.at

Abstract. This paper presents Asm2C++, a tool that automatically generates executable C++ code for Arduino from a formal specification given as Abstract State Machines (ASMs). The code generation process follows the model-driven engineering approach, where the code is obtained from a formal abstract model by applying certain transformation rules. The translation process is highly configurable in order to correctly integrate the underlying hardware. The advantage of the Asm2C++ tool is that it is part of the Asmeta framework that allows to analyze, verify, and validate the correctness of a formal model.

1 Introduction

The Abstract State Machines (ASM) method [4] is a formal method that is used to guide the rigorous development of software and embedded systems seamlessly from their informal requirements. The ASM method follows a design process based on the refinement principle that allows to capture all details of the system design by a sequence of refined models till the desired level of detail. It combines validation (by simulation and testing) and verification methods at any desired level of detail. The final step of this refinement process consists in realizing the implementation, generally code that is compiled and deployed on the real system. Performing this last step manually increases costs, limits the reuse of a formal specification, is error prone as some faults can be introduced in the code writing process, and can be a barrier for a wider adoption of ASMs. For these reasons, we have devised a methodology supported by the Asm2C++ tool that is able to generate the desired source code from ASMs. In this paper, we target Arduino[1] that is a widespread platform for rapid prototyping of embedded systems and

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.arduino.cc/.

© Springer International Publishing AG 2017
C. Barrett et al. (Eds.): NFM 2017, LNCS 10227, pp. 295–301, 2017.
DOI: 10.1007/978-3-319-57288-8_21

supports C++. It is also suitable for learning the design of embedded systems due to its low cost.

The ultimate aim of the paper is to show the implementation of the model-driven engineering (MDE) paradigm through ASMs: requirements models are platform independent, there is a clear distinction between platform-specific details and original user and system requirements, the code generation process is seamless and automatic, and last but not least, the rigorous quality and correctness assurance is embedded in the development process. As an additional goal, we aim at producing a code which is *readable* such that the code instructions can be easily *traced back* to the specification concepts and constructs. Although this may decrease the code efficiency, we believe that it increases the maintainability and the usability of the `Asm2C++` tool.

The paper is organized as follows: In Sect. 2, we present the ASM methodology. The process of code generation is presented in Sect. 3 and by means of a simple example, we illustrate some basic concepts of the proposed translation in Sect. 4. Section 5 presents some related work and Sect. 6 concludes the paper with some future work.

2 Abstract State Machine Methodology

The ASM method guides the development of software from requirements capture to code generation through several steps. Figure 1 shows the process of the ASM-based development. This method is supported by the `Asmeta` (ASM mETAmodeling) framework[2] [3] which provides a set of tools to help a developer in various development activities. The modelling process is based on refinement, i.e., it starts from an abstract model and adds further details to capture the complete system behaviour described in the requirements document. The correct refinement between two models is automatically proved using the `ASMRefProver` tool. If a model becomes complex, it is difficult to understand the behaviour only by the textual specification. For this reason, the visualizer `AsmetaVis` provides a visual notation that helps in the navigation of the model.

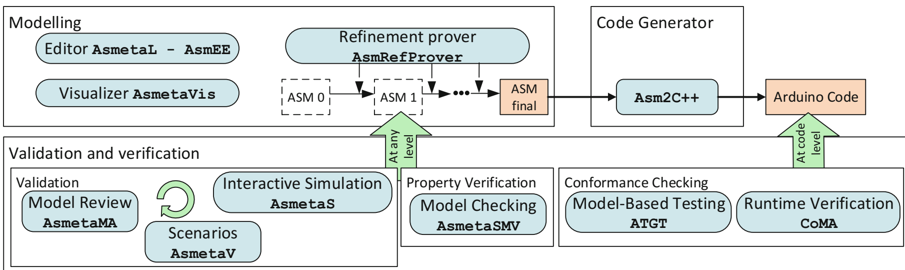

Fig. 1. ASM process: from requirements to code

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

The validation and verification (V&V) activities are well-integrated in the process, as shown in Fig. 1, and can be applied to any refined machine. The validation of a model can be achieved in multiple ways: either through the model simulator `AsmetaS`, through the model validator `AsmetaV` or through the model reviewer `AsmetaMA`. The simulator `AsmetaS` allows to perform two type of simulations: *interactive simulation* (the user inserts the values of parameters by choice) and *random simulation* (the tool randomly chooses the values that depends on the environment). The model validator `AsmetaV` takes *scenarios* as input files that contain the expected system behaviours. The scenarios are executed 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` verifies whether the properties, derived from the requirements document, comply with the behaviour of the model. When the final model is available, the Arduino code is automatically generated using the `Asm2C++` tool (see Sect. 3). When an actual code of the system implementation is available, *conformance checking* is possible. It is divided in model-based testing (to check the conformance *offline*) and runtime verification (to check the conformance *online*). The former uses the `ATGT` tool that automatically generates from ASM models tests cases which can be used to test any programming language. The latter, using the `CoMA` tool, can be used to perform runtime verification: the machine code is checked during the execution.

The language used by `Asm2C++` is UASM (Unified Syntax for Abstract State Machine) [2], the new ASM syntax developed by the ASM community to unify various ASMs dialects.

3 Code Generation Process

The translation process shown in Fig. 2 generates the runnable C++ code for Arduino starting from a UASM specification that we assume verified and validated. The first step of the transformation process consists in parsing the textual

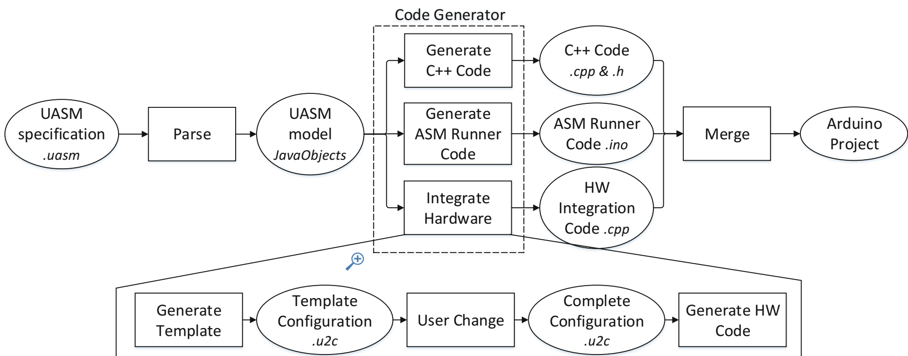

Fig. 2. Transformation process: from specification to code

specification and producing the UASM model, which is given to the code generator. The code generator performs three activities: (1) Generate C++ Code (2) Generate ASM Runner Code (3) Integrate Hardware. The result is merged as an Arduino project.

The first activity translates the ASM model into C++ code. The code is composed of a header (.h) that contains the translation of the ASM signature and a source (.cpp) file that defines how the ASM evolves by translating each ASM rule to a C++ method.

The second activity generates the Arduino code that defines the running policy according to the ASM execution divided in four iterative steps: acquire inputs, perform the main rule, update state, and release outputs. The output, the ASM Runner, is an .ino file that is the default extension for the Arduino C++ code.

The third activity integrates all HW-related aspects into the project: Arduino board version, I/O devices connections, Arduino-specific libraries that must be included, and any other HW-dependent information. The tool automatically generates a template configuration file (with .u2c extension). According to the HW configuration, the user edits this file which is used to generate the HW integration file. This is a C++ source file that works as an adapter between the generated code and the hardware. The output files are finally merged together to compose the Arduino project.

`Asm2C++` is built on top of 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 was realized by means of Xtend, a Java dialect provided by the Xtext framework with features for code generation. The listing below shows the translation scheme for the SeqBlock rule of the ASM method. A SeqBlock is a list of rules which are executed sequentially and is translated as a list of C++ instructions enclosed by curly brackets. In Xtend syntax, the content within ''' ''' symbols is a template string, while the code inside ≪ ≫ brackets is a variable part of the template expression that will be translated according to the rules parameter.

```
override String caseSeqBlock(SeqBlock rules) {
    return ''' {  ≪translateRules(rules.getRules())≫  } '''
}
```

The detailed information about the `Asm2C++` tool can be found at http://asmeta.sourceforge.net/download/asm2c++.html.

4 Illustrative Example

`Asm2C++` has been used to implement a small case study. The system is a control panel to be placed on the car dashboard that enables the driver to interact with

various car functionalities. The panel is responsible for controlling the following functionalities: 1. Switching on/off the system 2. Climate control 3. Smart headlights activation 4. Radio system. Code examples 1 to 4 in Fig. 4 focus on functionality 1 to show some translation rules. The ASM is translated in the CarPanel class, where domains, functions and rules become respectively data types, properties and methods. As shown in Code 3, the runner cyclically calls four CarPanel methods: 1. Acquire inputs from sensors (getInputs) 2. Perform the main rule (r_Main) 3. Update the ASM state (updateState) 4. Set outputs to actuators (setOutputs). Parallel execution is translated as described in [7], where controlled functions are duplicated and the state is updated only after the main rule.

The implementation process followed the methodology described in Sect. 2. We first defined a ground model that was progressively refined. When the model reached the last refinement step, we generated the runnable Arduino code. Along this process, we proved liveness properties with the model checker and executed some scenarios with the AsmetaV tool. In order to check the compliance between the specification and the code, we ran the same scenarios on the Arduino code, obtaining the same behavior as for the ASM simulation. The real system is shown in Fig. 3.

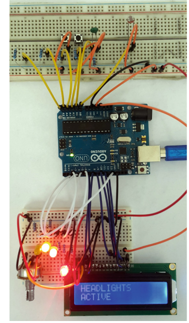

Fig. 3. CarPanel

```
asm CarPanelFinal
  enum Switch = {OFF, ON}
  controlled carState -> Switch
  initially OFF
  monitored carButton -> Switch
  ...
  rule r_Main =
    if carState = OFF then
      r_SwitchOnCar
    else if carButton = ON then
      carState := OFF
    else
      par
      r_Menu
      r_Headlights
      r_SetTemperature
      ...
      endpar
    endif
    endif
  ...
```

Code (1) UASM

Code (2) CarPanel.h

```
class CarPanel{
  enum Switch {OFF, ON};
  Switch carState[2],carButton;
  public:
  void getInputs();
  void r_Main();
  void updateState();
  void setOutputs();
  ...
};
```

```
#include "CarPanel.h"
CarPanel carPanel;
...
void loop(){
  carPanel.getInputs();
  carPanel.r_Main();
  carPanel.updateState();
  carPanel.setOutputs();
}
```

Code (3) ASM runner

```
#include "CarPanel.h"

// main rule
void CarPanel::r_Main(){
  if (carState[0] == OFF)
    r_SwitchOnCar();
  else if (carButton == ON)
    carState[1] = OFF;
  else{
    r_Menu();
    r_Headlights();
    r_SetTemperature();
    ...
  }
}
// apply the update set
// to the current state
void CarPanel::updateState(){
  carState[0]=carState[1];
}
...
```

Code (4) CarPanel.cpp

Fig. 4. Snippets from model and code

5 Related Work

Automatic code generation from formal specifications is available as a part of tool support for several formal methods. SCADE[3] and MATLAB/Simulink[4] provide this feature as a commercial off-the-shelf solution. The formal method B [1], on the other hand, provides this facility in the form of the Atelier B platform[5], that comes with code generators for different target languages, including C, C++, Java, and Ada, and its Community Edition is freely available without any restriction. EventB2Java is another tool that generates executable code implemented as a plug-in of the Rodin platform [5].

As best of our knowledge, there is no state of the art, reusable and publicly available tool for the ASM method that is capable of automatically generating programming language code from formal specifications written in the ASM method. In the past, [7] introduced a compilation scheme to transform an ASM specification (written in ASM-SL) into C++ code, but this work was done within a company setting. Although some of the key results of the proposed compilation scheme were useful for our work as mentioned in Sect. 4.

6 Conclusions and Future Work

We have presented `Asm2C++`, a tool that is able to generate C++ from formal specifications written as ASMs. This work follows the MDE paradigm: source code is obtained from requirements models by applying a set of M2T transformations. We have already successfully tried the tool with students of advanced programming courses to teach them rapid prototyping and designing of embedded devices.

In the future, we plan to extend the tool with an automatic test cases generator. From the ASM specification, a series of tests could be automatically generated which would be executed on the Arduino board. This would test both the system and the translation from the specification to the code. As, currently, the conformance relation between the specification and the code is coarsely defined, we also intend to formally specify and prove the correctness of the code transformation process.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Arcaini, P., Bonfanti, S., Dausend, M., Gargantini, A., Mashkoor, A., Raschke, A., Riccobene, E., Scandurra, P., Stegmaier, M.: Unified syntax for abstract state machines. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 231–236. Springer, Cham (2016). doi:10.1007/978-3-319-33600-8_14

[3] http://www.esterel-technologies.com/products/scade-suite/.
[4] https://www.mathworks.com/products/simulink/.
[5] http://www.atelierb.eu/en/.

3. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw.: Pract. Exp. **41**, 155–166 (2011)
4. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, New York (2003)
5. Cataño, N., Rivera, V.: EventB2Java: a code generator for event-B. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 166–171. Springer, Cham (2016). doi:10.1007/978-3-319-40648-0_13
6. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on OOPSLA, pp. 307–309. ACM (2010)
7. Schmid, J.: Compiling abstract state machines to C++. JUCS **7**(11), 1068–1087 (2001)