Model-driven Development of an Automated Material Flow System: An Experience Report

Marco Konersmann
konersmann@uni-koblenz.de
University of Koblenz-Landau, Germany

Jens Holschbach
jens.holschbach@paluno.uni-due.de
University of Duisburg-Essen, Germany

Abstract

Automated Material Flow Systems (aMFS) are usually developed by copying the code for a similar system and adapting it for the new one. Copies of systems do usually not profit from the evolution of its original system. In this paper we present and discuss our experiences in implementing an aMFS with a component-oriented, model-driven development approach in an academic course.

Introduction Automated Material Flow Systems (aMFS) are a crucial and standard part of Cyber-Physical Production Systems (CPPS). They connect different machines or plant parts to transport material from one to the other. During the (re)engineering of aMFS, the initial (re)engineering stage, i.e., drawing up the material flow plan of the system, generally serves as requirements document for other engineering disciplines, including software engineering.

After that, reuse of control software is commonly achieved through copying, pasting, and modifying existing control code from a similar aMFS. The static structure and high variability of today's control software of aMFS prevent commonly required adaptions, e.g., adding a light barrier or exchanging an LED with a piezo buzzer. Often, the software has to be modified in different parts of the monolithic structure or existing code must be wrapped to adapt the interfaces and/or behavior. Both approaches are error-prone and reduce the software quality.

To overcome these problems, we envision a model-driven development approach for aMFS that comprises hierarchical components with unified interface descriptions and an integration of models, code, and runtime information [1]. For supporting efficient reuse and evolution, we define four requirements that have to be fulfilled by a model-driven approach for aMFS: (R1) An aMFS must be described by reusable components. (R2) Components must be composable. Composed components interconnect their children via interfaces. (R3) It must be possible to formally validate or (re)create the consistency between interconnected components. (R4) Bidirectional consistency must be ensured between models and the implemented program code. I.e., when the code or the model is changed, changes must be propagated to the other artifact to support (ad-hoc) evolution.

In this paper we present the experiences from an academic course in which four master's students developed an aMFS prototype, that produces sweets on demand, with the concepts of the envisioned approach.

System Development and Results The students first learned about the technologies used in the course for collaboration and for the development of languages and code generators, the development of cyber-physical systems using Arduino development tools on ESP8266 microcontrollers, and an introduction into different kinds of sensors and actuators. Afterwards the team planned and distributed their work in a guided self-organization.

Then they decided on the basic requirements for the aMFS to build. The system (called "FlowScale", see Figure 1) fills boxes with sweets on demand. Each box is identified via an RFID card. The system consists of *filling stations* which fill the boxes with a set of cuboid sweets. Each station has sweets with a specific color. The sweets are initially stacked in one *charger* per station and pushed out with a stepper motor, landing in the boxes. Each station controls its own conveyor *belt* with a stepper motor to move the boxes forward. *Ultrasonic sensors* ensure the right positioning of the box. An *RFID scanner* at each station identifies the individual box and asks a manufacturing execution software (MES) how many pieces of its specific sweet should be pushed into the box.

To describe an aMFS, the students defined a DSL using Xtext[1]. In the DSL, *components* define incoming and outgoing events, optional child components, a state machine, and the number of general purpose input-output (GPIO) pins they require on a microcontroller. *Devices* represent microcontrollers, that execute the software for the components. Components can be mapped to devices. Table 1 gives an overview of the components defined for the FlowScale system. The component composition is shown in Figure 2.

The students developed code generators for C++ with Xtend[2]. The generated code represents hierarchical components which can be interconnected locally or distributed over multiple microcontrollers. The

[1]https://www.eclipse.org/Xtext/
[2]https://www.eclipse.org/xtend/

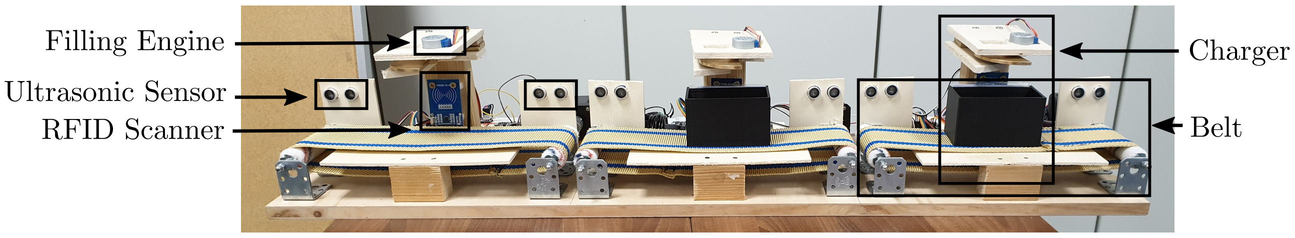

Figure 1: The prototype for ordering sweets

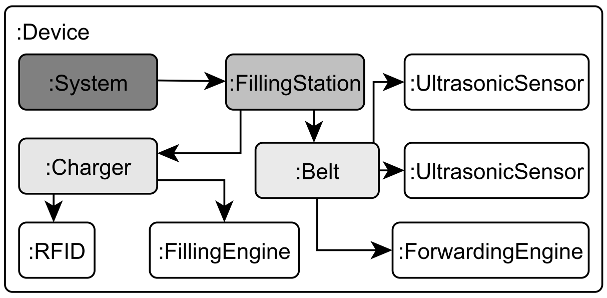

Figure 2: Composition of components in FlowScale

UltrasonicSensor	Used to detect containers
ForwardingEngine	Moves boxes using a roller conveyor with a motor
RFID Scanner	Used to identify boxes
Belt	Composes two ultrasonic sensors and an forwarding engine to move boxes
FillingEngine	Uses a motor to push exactly one sweet off the magazine, by rotating a disc with a hole by 360 degrees
Charger	Componses an RFID scanner and a filling engine to fill boxes with sweets
FillingStation	Composes one charger and one belt
System	Composes multiple filling stations

Table 1: The components of the FlowScale system

components can implement state machines and have no static dependencies to their parents or neighbours. The code has to be manually adapted with implementation details, for which the generated code explicitly provides areas. FlowScale uses ESP8266 microcontrollers for running the code.

To experimentally evaluate the evolvability of the approach, the students added a component for signaling the result of an order. In case of an error, e.g., a box has been taken out in between and not correctly inserted again, an LED blinks. For this scenario a new component was defined and the code generator was extended accordingly. The model was extended and the generated code can successfully excecute on the microcontrollers. In another evolution scenario, the LED was replaced with a piezo buzzer.

Discussion With the protoyipical implementation of an aMFS, we implemented reusable, composable components (requriements R1 and R2). The code generation provides one-way consistency from models to code. In the described course, the translations were developed for bidirectionality (R4), but only code generation was implemented due to the low number of participants. We are going to implement the code-to-model transformation in future work. The specification and validation of in- and output constraints between components (R3) were not part of the course. Ongoing work is examining this field.

The results have shown that the models created in the DSL are easy to understand and to evolve. The code generation enforces well-structured code. The semantic mapping between model elements and the resulting program code is encoded in these model-to-code transformations. For future developments, this knowledge should be made more explicit, e.g., via a comprehensive documentation. When aMFS are to be developed with our envisioned approach, new device types have to be added continuously. It is necessary that bidirectional model/code transformations can be easily defined. We are currently evaluating different alternatives for describing these transformations.

Conclusion Usually, aMFS are developed in a copy-paste-change style, which is a problem for evolution and maintenance. In this paper we described how students developed a DSL and code generators for a sweets production system, following our ideas for developing component-based aMFS [1]. Four master's students with an upfront training created well-structured code. Their models are easy to understand and evolve. As next step, we are going to implement the missing bidirectionality and consistency validation.

Acknowledgements We would like to thank Pedram Golkar-Hamzey-Yazd, Christian John, Sebastian Pobel, and Maximilian Staubach for their contributions to the project.

References

[1] Birgit Vogel-Heuser, Marco Konersmann, Thomas Aicher, Juliane Fischer, Felix Ocker, and Michael Goedicke. Supporting evolution of automated Material Flow Systems as part of CPPS by using coupled meta models. In *ICPS 2018*. IEEE, May 2018.