Available online at www.sciencedirect.com

ScienceDirect

Procedia Manufacturing 24 (2018) 153–158

Procedia MANUFACTURING

www.elsevier.com/locate/procedia

4th International Conference on System-Integrated Intelligence

Generating Correct, Compact, and Efficient PLC Code from Scenario-based Assume-Guarantee Specifications

Daniel Gritzner[a,*], Joel Greenyer[a]

[a]*Leibniz Universität Hannover, Fachgebiet Software Engineering, Welfengarten 1, D-30167 Hannover, Germany*

Abstract

Cyber-physical systems can be found in many areas, e.g., manufacturing, health care or smart cities. They consist of many distributed components cooperating to provide increasingly complex functionality. The design and development of such a system is difficult and error-prone. To help engineers overcome these challenges we created a formal, scenario-based specification language. Short scenarios, i.e., event sequences, specify requirements and the desired behaviors by describing how components may, must, or must not behave. Scenarios provide an intuitive way for creating formal assume-guarantee (GR(1)) specifications, giving engineers easy access to simulation, for validating the specified behavior, and controller synthesis, for creating controller software which is correct by construction. In this paper we present an approach for generating Programmable Logic Controller (PLC) code from a scenario-based specification. Previous code generation efforts, including our own, created large, verbose source files causing some tools, e.g., compilers or editors, to perform slowly or even become unresponsive. Our new approach creates compact files, shifting significant amounts of code from executable instructions to data, to reduce the burden on the compiler and other tools. The generated code is efficient and introduces minimal to no latency between the occurrence of an event and the system's reaction to it.

Keywords: code generation; assume-guarantee specification; scenarios; controller synthesis; programmable logic controller

1. Introduction

Cyber-physical systems can be found in many areas, e.g., manufacturing, health care or smart cities. In a cyber-physical system many components, such as robots for example, need to cooperate to provide some desired functionality. Each component may even be involved in multiple concurrent cooperative processes. As users demand increasingly complex functionality, the required behavior of each component becomes increasingly complex as well. This complexity makes designing and developing cyber-physical systems a difficult and error-prone task. At all times engineers need to account for the multitude of scenarios each component may be involved in concurrently.

* Corresponding author. Tel.: +49-511-762-19675
E-mail address: daniel.gritzner@inf.uni-hannover.de

10.1016/j.promfg.2018.06.046

To help with the software development for such systems we created a formal, scenario-based specification language, called Scenario Modeling Language (SML) [7]. SML allows engineers to express, in an intuitive way [1], how components may, must or must not behave. Short scenarios, i.e., event sequences, specify requirements and desired behavior as well as assumptions about the environment's behavior. SML specifications enable simulation for manual validation of the specified system early in the design process. Additionally, the specifications can be automatically checked for realizability, i.e., can a system even behave in the specified way or are there any requirements which contradict each other. This automatic checking is based on a General Reactivity of rank 1 (GR(1)) condition, thus accounts for environment assumptions: can the desired system behavior be achieved given that the environment behaves as assumed? These features, simulation and automatic checking, help engineers to create good specifications.

However, a good specification does not guarantee that implementations derived from it work as intended. We therefore developed a code generator which generates Structured Text that can be executed on common Programmable Logic Controllers (PLCs). The generated code is correct by construction and takes care of the complex interleaving of individual component's behavior into one coherent system behavior. Only hardware vendor-specific details have to be implemented manually. If, for example, the specification requires a robot to move into a specific position under certain circumstances, the generated code will automatically issue an instruction to the robot in the proper situation. Only how the robot carries out this atomic task, performing the required movement, needs to be implemented manually.

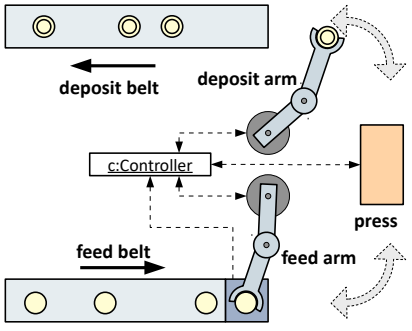
Code generation often creates large, verbose source files causing some tools, e.g., compilers, editors, or version control software, to perform slowly or even become entirely unresponsive and thus causing unnecessary downtime for engineers. In this paper we present a code generator which produces compact source files which carry significant amounts of the generated code as data, by representing transitions between states of a state machine via mapping functions which themselves are encoded by integer arrays, as opposed to executable instructions. This reduces the burden on all tools processing the generated code, especially the compiler. Furthermore, the generated code is efficient and introduces minimal to no latency between the occurrence of an event and the system's reaction to it, especially compared to our previous efforts in the area of code generation [8].

The remainder of this paper is structured as follows. Section 2 introduces a running example, while Sect. 3 explains the core concepts of SML and software controller synthesis from SML. Section 4 then builds on these foundations to present our code generation approach. The paper concludes with related work in Sect. 5 and a summary in Sect. 6.

2. Example

We use a production system example of a typical manufacturing process as shown in Fig. 1 to explain our approach. In the example, blank work items arrive via a conveyor belt, are transported to a press by a robot arm, and are then pressed into useful items. These items are transported to another conveyor belt by a second robot. The figure also shows the guarantees, i.e., requirements and desired behavior, the system must fulfill as well as the assumptions under which it must fulfill these guarantees. The assumptions model the environment the system will be in when it is in use.

Guarantees **G1-G5** not only define the system's desired behavior and requirements, they also include additional conditions, e.g., "[...] the feed arm must pick it up *when possible*." in **G1**. A new blank may arrive while the feed arm



guarantees (requirements and desired behavior):
(G1) When a new blank arrives, the feed arm must pick it up when possible.
(G2) After picking up an item, the feed arm must move to the press, release the item into the press (when the press is ready), and finally move back to the feed belt.
(G3) When an item is put into the press, the press must start pressing.
(G4) When the press finishes, the deposit arm must pick the pressed item up when possible.
(G5) After picking up an item, the deposit arm must move to the deposit belt, release the item onto the deposit belt, and finally move back to the press.

assumptions:
(A1) The feed arm is able to pick up every blank before the next one arrives.
(A2) After being instructed to press an item, the press will eventually finish.
(A3) After a robot arm is instructed to move to a new location, it will eventually arrive at the new location.
(A4) After a robot arm is instructed to pick up an item, it will eventually pick up that item.
(A5) After a robot arm is instructed to release an item, it will eventually release that item.

Fig. 1. A production system (left) consisting of two robot arms, each adjacent to a conveyor belt, a press, and a software-based controller sending instructions to other components as well as processing their sensor inputs. An informal specification for that system is shown on the right.

```
1  guarantee scenario BlankArrives { // G1
2      feedBelt -> controller.blankArrived()
3      wait [feedArm.location == feedBelt && !feedArm.carriesItem]
4      urgent controller -> feedArm.pickUp()
5  }
6  guarantee scenario ArmDeliversItemToPress { // G2
7      feedArm -> controller.pickedUpItem()
8      urgent controller -> feedArm.moveTo(press)
9      feedArm -> controller.arrivedAt(press)
10     wait [!press.hasItem]
11     urgent controller -> feedArm.releaseItem()
12     feedArm -> controller.releasedItem()
13     urgent controller -> feedArm.moveTo(feedBelt)
14 }
15 guarantee scenario PressStartsPressing { // G3
16     feedArm -> controller.releasedItem()
17     urgent controller -> press.startPressing()
18 }
19 [...]
```

Listing 1. Excerpt of a specification for our production system example; comments indicate which guarantee each scenario models

is still delivering the previous blank or is still on its way back to the feed belt. In these cases the feed arm must only be instructed to pick up the newly arrived blank when it is at the feed belt while not already carrying an item.

A1 specifies that no queue of unprocessed work items forms at the feed belt as the feed arm is able to pick up any new blanks more frequently than they arrive. Assumptions **A2-A5**, which engineers usually make implicitly, specify that all components are operating normally and eventually finish their tasks. Making them explicit is necessary for formal methods such as simulation or controller synthesis. They also improve the maintainability of a specification.

3. Scenario-based Modeling

To help engineers with the development of reactive systems, including cyber-physical systems, we developed a scenario-based modeling approach centered around a DSL we call Scenario Modeling Language (SML) [7]. SML, which is a text-based variant of Life Sequence Charts [11], makes it easy to write formal, scenario-based specifications by writing short scenarios. Each scenario describes how components may, must, or must not behave. Listing 1 shows an excerpt of an SML specification, a more detailed version of which can be found in [8].

Scenarios are automatically interwoven into a coherent system behavior based on common events (e.g., lines 12 and 16) instead of requiring explicit function calls. Most events are *message events* representing sensor events (e.g., line 2), components reporting that they finished a task (e.g., lines 7 and 9), or the software controller sending instructions to another components (e.g., lines 4 and 8). But events may also be based on the system's state (e.g., line 10). Comparing the informal guarantees and assumption from Fig. 1 to their formal scenario counterparts in Listing 1 shows that the informal and formal version are similar, making it easy to formalize requirements and assumptions as scenarios.

The play-out algorithm [11] defines how multiple concurrently active scenarios are interwoven into valid event sequences. In SML, components are divided into *controllable system components* (the controller in the center of the left side of Fig. 1) and *uncontrollable environment components* (every other component). The play-out algorithm waits for the environment to choose an event, activates and progress scenarios accordingly, and then picks a valid reaction. The play-out algorithm induces a *non-deterministic state space*. Fig. 2 shows an excerpt of the state space induced by our example specification. Each state in this space represents the states of all components and a set of active scenarios. The play-out algorithm determines which events are possible based on these states. Each state is either a *system*

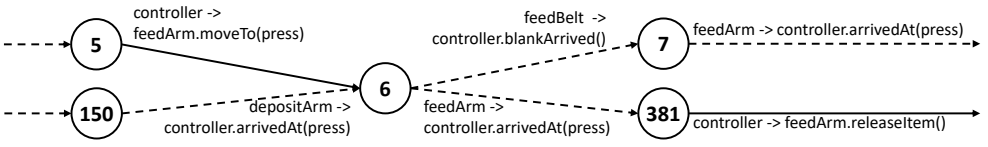


Fig. 2. Excerpt of a state graph induced by our example specification. The nodes represent system states and the labeled edges represent events. Solid arrows indicate that an event is a controllable system event, whereas dashed arrows indicate than an event is an environment event.

state or an *environment state* depending on whether the outgoing events are *system events* (the sending component is controllable) or *environment events* (the sending component is uncontrollable). We call a sequence of system events which is delimited by environment events but not interrupted by any such events a *super step* of the system.

To generate executable code from an SML specification compute a deterministic strategy which fulfills a General Reactivity of rank 1 (GR(1)) condition, i.e., which fulfills all guarantees or ensures that at least one assumption cannot be fulfilled. This process is called *controller synthesis*. A scenario is fulfilled iff on any infinite path through the induced state space all of the scenario's obligations (e.g., *urgent* message events) are fulfilled and it is never violated (e.g., by violating a *strict* order of events). We use Chatterjee's attractor-based GR(1) game solving algorithm [5] to compute a set of *winning states*, i.e., the set of states in which the system can fulfill the GR(1) condition regardless of the environment's behavior (i.e., its choice of events). From the winning states we extract a strategy for the system, which, in that case, is a state space graph in which all system states have exactly one outgoing event. In previous work we even explained how to synthesize a controller which makes the final system behave in an energy-efficient way [9].

4. PLC Code Generation

In our previous work we already presented an approach for generating Structured Text for PLCs from a scenario-based specification via controller synthesis [8]. In this approach we suggested generating a single primary state machine representing the software controller itself (the controller in Fig. 1) and one secondary state machine for every environment component (the two robot arms and the press in Fig. 1). The primary state machine instructs the secondary state machines when to perform which action. The secondary state machines implement these actions and report back when done. The communication between state machines happens via integer state variables and boolean event variables. The primary state machine, which handles the complex interleaving of actions into a coherent system behavior and thus contains a lot of the complexity of the software, is fully generated. Additionally, templates for every secondary state machine are generated. Manual implementation effort is only required for completing the templates of the secondary state machines with hardware-specific instructions, any required hardware-specific initialization, and code for setting sensor variables to TRUE when the appropriate event occurs.

This approach already has desirable properties: a lot of the complex code (the primary state machine) is generated and manual implementation effort is only required for isolated and easier to understand fragments. If the process changes in the sense that the same components perform the same actions but in a different combination, only the fully generated primary state machine needs to be replaced. However, the primary state machine, which is a naive translation of the state graph outlined in Fig. 2 into a state machine using a large CASE ... OF construct, can easily reach thousands or ten thousands of states with each state consisting of multiple statements. This creates very large files which may put a significant burden on tools such as compilers and code editors as they may not expect such large files since handwritten code is usually subdivided into smaller files. In our experience, some tools PLC code editors may even become completely unresponsive when trying to copy and paste the generated primary state machine into one of their code editor windows. Additionally, this naive pattern of code generation relies on the compiler to generate optimized code which quickly jumps to and executes the code for the current state. Furthermore, latencies between the occurrence of an event and its processing are introduced as only one event can be processed in each PLC cycle.

We therefore developed a new approach for generating the primary state machine while keeping the rest of our architecture. In this new approach, an example of which is shown in Listing 2, we generate mapping functions for events. For each environment event appearing in the strategy we create a source and a target integer array (e.g., lines 9 and 10) to represent the mapping function. The lengths of these arrays is equal to number of occurrences of the event within the strategy graph. As an example, to encode the transition from state 6 to state 7 in Fig. 2 there would be an index i s.t. feedBelt_controller_blankArrived_source[i] = 6 and feedBelt_controller_blankArrived_target[i] = 7. The indices i are chosen s.t. that the source array is sorted in ascending order.

If, at runtime, an event occurs, i.e., its respective boolean variable is set to TRUE, we perform a binary search on its associated source array to find the index of the entry with the current value of the global variable controllerState (e.g., lines 15-18). We then update the controllerState to the value at the same index in the associated target array. In a loop (lines 13-23), we update the controllerState until all environment events have been processed or no additional environment event source array contains the current controllerState. The later means we are in a system super step. The code for this loop and the binary search is fully generated. On platforms offering pointers in Structured Text,

```
1  VAR // encoding of mapping functions representing state graph - BEGIN
2      changed : BOOL;
3      index : INT;
4      source : ARRAY [0..1000] OF INT := [...]
5      target : ARRAY [0..1000] OF INT := [...]
6      feedRobotTarget : ARRAY [0..1000] OF INT := [...]
7      depositRobotTarget : ARRAY [0..1000] OF INT := [...]
8      workerRobotTarget : ARRAY [0..1000] OF INT := [...]
9      feedBelt_controller_blankArrived_source : ARRAY [0..652] OF INT := [...]
10     feedBelt_controller_blankArrived_target : ARRAY [0..652] OF INT := [...]
11     [...]
12 END_VAR // encoding of mapping functions representing state graph - END
13 REPEAT // handling of environment events - BEGIN
14     changed := FALSE;
15     changed := (BinarySearch(ADR(feedBelt_controller_blankArrived),
16                              ADR(feedBelt_controller_blankArrived_source),
17                              ADR(feedBelt_controller_blankArrived_target),
18                              (SizeOf(feedBelt_controller_blankArrived_source)/2)-1)
19                     <> -1)
20                OR changed;
21     [...]
22     UNTIL NOT changed
23 END_REPEAT // handling of environment events - END
24 changed := TRUE; // handling of system super step - BEGIN
25 index := BinarySearch(ADR(changed), ADR(source), ADR(target), (SizeOf(source)/2)-1);
26 IF index <> -1 THEN
27     IF feedRobotTarget[index] <> 0 THEN
28         feedRobotState := feedRobotTarget[index];
29     END_IF
30     IF depositRobotTarget[index] <> 0 THEN
31         depositRobotState := depositRobotTarget[index];
32     END_IF
33     IF workerRobotTarget[index] <> 0 THEN
34         workerRobotState := workerRobotTarget[index];
35     END_IF
36 END_IF // handling of system super step - END
```

Listing 2. Example of a generated primary state machine with CODESYS-specific enhancements such as the use of pointers

e.g., CODESYS, the code for the binary search needs to be generated only once and can be reused for every event. For standard-compliant Structured Text, duplicates of the binary search implementation have to be generated. The generated loop and binary search can efficiently handle multiple events in a single PLC cycle, thus reducing the latency compared to our previous approach. Additionally, fewer lines of code are needed.

System events are handled similarly to environment events in the generated code. There is a source and a target array for every super step of the system which encodes controllerState updates (lines 4 and 5). Additionally, there is a single array for every secondary state machine (lines 6-8). The length of these arrays is equal to the number of super steps. The index found in the binary search to update the controllerState is also used to look up the values in all of the secondary state machine arrays (lines 25-36). If any of the looked up values is non-zero, that particular secondary state machine's state variable is set to the looked up value to represent the issuing of an instruction to perform an action. Since every action performed by an environment component is characterized by two events, a system event marking its beginning and an environment event marking its end, no super step will contain two system events which represent instructions issued to the same secondary state machine. Thus, this approach can efficiently encode all super steps found in the strategy.

We generated code from a specification for the example in Fig. 1 and successfully tested this code in a simulation to ensure that it is working as intended. The strategy, from which the code was generated, consisted of 3017 states and 5958 transitions (after merging system transitions into super steps). Compared to our previous code generation approach, the size of the primary state machine went from 21978 lines of code and a 792128 bytes file size down to 789 lines of code and 97669 bytes for standard-compliant Structured Text and 135 lines of code and 87627 bytes for CODESYS-specific Structured Text. In this case, the new approach required less than 5% of the lines of code compared to the old approach and the file size was reduced by more than 85% for the primary state machine.

5. Related Work

There exists previous work on synthesizing controllers from LSC/SML-style scenarios [4, 10], and other forms of scenarios [13]. Most of these approaches produce finite state controllers or state machines as output, from which

code can be generated. Some consider code generation in particular for robotics/embedded applications [2, 12]. The novelty of our synthesis procedure w.r.t. to the above is that it supports scenario-based specifications with a greater expressive power—assume/guarantee specifications.

There is work on generating PLC code from state machines [15] or Petri nets [16], and using formal methods to verify PLC code [3]. Other work considers synthesis and code generation, some specifically for robotics applications, based on temporal logic specifications such as LTL and its GR(1) fragment [6, 14]. In contrast to temporal logics based approaches, LSCs/SML aim to provide a more intuitive language that is easier to use.

6. Conclusion

In this paper we presented an approach for generating compact and efficient Structured Text executable on PLCs. We generate this code from scenario-based specifications written in SML. Using SML engineers can easily define requirements, desired behavior, and environment assumptions of a system to create a formal assume-guarantee specification. The generated code uses multiple state machines to separate the decision "when to perform which atomic action" from the implementation of each atomic action. After code generation, engineers only need to implement the atomic actions, with their complex interleaving into the desired system behavior having already been generated.

Acknowledgment

This research is funded by the DFG project EffiSynth.

References

[1] Alexandron, G., Armoni, M., Gordon, M., Harel, D., 2014. Scenario-based programming: Reducing the cognitive load, fostering abstract thinking, in: Proc. 36th Int. Conf. on Software Engineering (ICSE), pp. 311–320.

[2] Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schäfer, W., Meyer, M., Pohlmann, U., Priesterjahn, C., Tichy, M., 2014. The MechatronicUML design method – process and language for platform-independent modeling .

[3] Biallas, S., Brauer, J., Kowalewski, S., 2012. Arcade.PLC: A Verification Platform for Programmable Logic Controllers, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 338–341. doi:10.1145/2351676.2351741.

[4] Brenner, C., Greenyer, J., Schäfer, W., 2015. On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications, in: Egyed, A., Schaefer, I. (Eds.), Fundamental Approaches to Software Engineering (FASE 2015). Springer. volume 9033 of *Lecture Notes in Computer Science*, pp. 51–65.

[5] Chatterjee, K., Dvorák, W., Henzinger, M., Loitzenbauer, V., 2016. Conditionally Optimal Algorithms for Generalized Büchi Games, in: Faliszewski, P., Muscholl, A., Niedermeier, R. (Eds.), 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 25:1–25:15.

[6] Ehlers, R., Raman, V., 2016. Slugs: Extensible GR(1) Synthesis. Springer International Publishing, Cham. pp. 333–339.

[7] Greenyer, J., Gritzner, D., Katz, G., Marron, A., 2016. Scenario-based modeling and synthesis for reactive systems with dynamic system structure in scenariotools, in: de Lara, J., Clarke, P.J., Sabetzadeh, M. (Eds.), Proceedings of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2016), CEUR. pp. 16–32.

[8] Gritzner, D., Greenyer, J., 2017. Synthesizing executable PLC code for robots from scenario-based GR (1) specifications, in: Proc. of the 4th Int. Workshop on Model-driven Robot Software Engineering (MORSE 2017).

[9] Gritzner, D., Greenyer, J., 2018. Synthesis of Cost-optimized Controllers from Scenario-based GR(1) Specifications, in: Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D., Seidl, C. (Eds.), Modellierung 2018, Gesellschaft für Informatik e.V., Bonn. pp. 167–182.

[10] Harel, D., Kugler, H., 2002. Synthesizing state-based object systems from LSC specifications. Foundations of Computer Science 13:1, 5–51.

[11] Harel, D., Marelly, R., 2003. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer.

[12] La Manna, V.P., Greenyer, J., Clun, D., Ghezzi, C., 2015. Towards executing dynamically updating finite-state controllers on a robot system, in: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 42–47.

[13] Liang, H., Dingel, J., Diskin, Z., 2006. A comparative survey of scenario-based to state-based model synthesis approaches, in: Proc. Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ACM, New York, NY, USA. pp. 5–12.

[14] Maoz, S., Ringert, J.O., 2015. Synthesizing a lego forklift controller in GR(1): A case study, in: Proceedings of the 4th Workshop on Synthesis (SYNT), co-located with CAV. 2015.

[15] Sacha, K., 2005. Automatic Code Generation for PLC Controllers. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 303–316.

[16] Thapa, D., Dangol, S., Wang, G.N., 2005. Transformation from petri nets model to programmable logic controller using one-to-one mapping technique, in: International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06), pp. 228–233.