Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications

Daniel Gritzner(✉) and Joel Greenyer

Fachgebiet Software Engineering, Leibniz Universität Hannover,
Welfengarten 1, 30167 Hannover, Germany
{daniel.gritzner,greenyer}@inf.uni-hannover.de

Abstract. Robots are found in most, if not all, modern production facilities and they increasingly enter other domains, e.g., health care. Robots participate in complex processes and often need to cooperate with other robots to fulfill their goals. They must react to a variety of events, both external, e.g., user inputs, and internal, i.e., actions of other components or robots in the system. Designing such a system, in particular developing the software for the robots contained in it, is a difficult and error-prone task. We developed a formal scenario-based modeling method which supports engineers in this task. Using short, intuitive scenarios engineers can express requirements, desired behavior, and assumptions made about the system's environment. These models can be created early in the design process and enable simulation as well as an automated formal analysis of the system and its components. Scenario-based models can drive the execution at runtime or can be used to generate executable code, e.g., programmable logic controller code. In this paper we describe how to use our scenario-based approach to not only improve the quality of a system through formal methods, but also how to reduce the manual implementation effort by generating executable PLC code.

Keywords: Code generation · Robot · Scenario · GR(1) specification

1 Introduction

Robots are found in many domains, e.g., manufacturing, transportation, or health care. Especially in manufacturing they are ubiquitous. Modern production systems implement complex processes, often requiring the cooperation of many robots to achieve their desired goals. Each robot may even be involved in several concurrent processes, making the design of its behavior a difficult and error-prone task. The robot has to react to a multitude of events, both external events, e.g., sensor inputs, and internal events, i.e., actions of other robots in the system. The inherent complexities of modern manufacturing processes make it difficult to develop robot software which is free of defects, that is, which makes the robot act or react properly under all possible circumstances. The specification, from which an implementation is derived, may be inconsistent and the

© Springer International Publishing AG 2018
M. Seidl and S. Zschaler (Eds.): STAF 2017 Workshops, LNCS 10748, pp. 247–262, 2018.
https://doi.org/10.1007/978-3-319-74730-9_23

manual implementation thereof itself may introduce further defects. The task of designing such systems becomes even more difficult when considering non-functional requirements such as reducing the system's energy consumption.

We developed a formal, yet still intuitive scenario-based specification approach to support engineers with the difficult design of such systems. Our approach uses short scenarios to model *guarantees* (goals, requirements, or desired behavior) and *assumptions* made about the environment. Scenarios are sequences of events, similar to how engineers describe requirements to each other, e.g., *"When A and B happen, then component C_1 must do D, followed by C_2 doing E."* These sequences are used to intuitively describe when events or actions may, must, or must not occur [1,14]. The formal nature of scenario-based specifications allows applying powerful analysis techniques early in the design process. Through simulation and controller synthesis, which, if successful, can prove that the requirements defined in the specification are consistent, defects can be found and fixed early during development. The same techniques used for simulation can be used to directly execute a specification at runtime [15] and the techniques used for controller synthesis can be used to automatically generate executable code. This reduces manual implementation effort significantly, thus mitigating some of the cost of writing a formal specification. With mature enough tool support, an overall reduction in development costs could even be achieved.

The contribution of this paper is an approach for generating executable code for Programmable Logic Controllers (PLCs) from aforementioned scenario-based specifications. This enables engineers to use formal methods such as checking if all requirements are consistent to ensure the correctness of the specification and to generate code which is correct by construction. A PLC program must handle two concerns: (1) it must correctly decide when to perform which atomic action, e.g., when to move which robot arm to which location, and (2) it must implement each atomic action, e.g., moving a specific robot arm to a specific location. Our approach generates code handling the first concern, leaving only the manual implementation of atomic actions to engineers. From the point of view of Model Driven Architecture [20], a scenario-based specification would be a Platform Independent Model of a system and the generated PLC code, after an implementation of each atomic action has been added, would be a Platform Specific Model of the same system. The latter can then be used directly as the software for an actual physical version of the specified system.

The remainder of this paper is structured as follows. Section 2 introduces an example used for explanation and discussion throughout the paper. Sections 3 and 4 introduce scenario-based modeling and controller synthesis. Section 5 builds on these foundations to describe how to generate PLC code from such a controller. The paper finishes with related work and a conclusion in Sects. 6 and 7.

2 Example

To explain and discuss our approach we use a production system example, shown in Fig. 1. It models a typical manufacturing process. Blank work items arrive via

a feed belt, which has a sensor telling a controller about the arrival of new work items. These blanks are then picked up by a robot arm and put into a press, which will press the blanks into useful items. These pressed items are then picked up by another robot arm which will put the items on a deposit belt which will transport the items to their next destination.

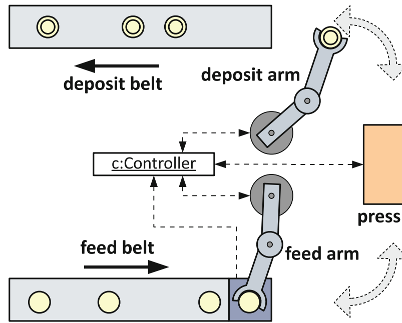

Fig. 1. A production system consisting of two robot arms, each adjacent to a conveyor belt, a press, and a software-based controller sending instructions to other components as well as processing their sensor inputs.

The specification we use for this example models the following guarantees **G** and assumptions **A**:

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.
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.

Guarantees **G1**–**G5** define the system's desired behavior and requirements it must fulfill as described at the beginning of this chapter. They also include additional conditions, e.g., "[...] the feed arm must pick it up *when possible*." in

G1. These conditions express additional structural conditions required to fulfill certain goals. In the same example, **G1**, a new blank may arrive while the feed arm 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 back at the press.

The assumptions specify what the engineers assume is true about the environment the system will operate in. As an example, **A1** specifies that the feed arm is able to pick up arriving blanks more frequently than the frequency of arrival of new blanks. This assumption implies that no queue of unprocessed blanks forms at the feed belt. Assumptions **A2**–**A5** specify that robot arms and the press will eventually finish their tasks after being instructed to perform a certain action. These assumptions are actually important to ensure that the specification is realizable, since they basically specify that system is operating normally, i.e., the components are working as intended. A correctly working system only needs to fulfill its guarantees as long as all assumptions hold.

3 Scenario-Based Modeling

In this section we introduce our scenario-based modeling approach, which we use to write formal specifications. It is based on a DSL we developed for modeling scenarios, called the Scenario Modeling Language (SML) [16].

SML offers engineers an easy to use way to write formal, scenario-based specifications. It is a text-based variant of Life Sequence Charts [10,18], offering a similar feature set with a few extensions. Listing 1 shows the specification of our production system example. Comments next to each scenario indicate which guarantee or assumption they represent. A scenario-based specification also consist of a class model, called domain model, describing the different components of the system and an instance thereof, an object model. The object model contains a concrete instance for every physical component of the specified system.

A specification references a domain model (line 1) and has a name (line 2). In our example the domain model contains classes such as *RobotArm* and *Press*. These classes model each component type's attributes and possible events it can receive. Events can be either actions it should perform or sensor events it may be notified of. Our production system example includes events such as a *RobotArm* being told to pick up an item or the *Controller* being notified of the arrival of a new blank. The specification defines which components are software-*controllable* (line 4) with all other classes automatically being interpreted as *uncontrollable*, also called environment-controllable. Non-spontaneous events (lines 5–13) are events which cannot occur unless enabled, e.g., the event *pressingFinished* cannot occur unless assumption **A2** is active (lines 51–54) and is in a state in which the second line is expected next. Other events, sent by uncontrollable objects and being the initializing event of a scenario (e.g., *blankArrived*) can occur spontaneously. This then triggers the creation of an instance of a scenario called an *active scenario*. Active scenarios have one or more references to events they expected next, called *enabled events*. When *PressEventuallyFinishes* (lines

```
1  import "../model/productioncell.ecore"
2  specification ProductioncellSpecification {
3      domain productioncell
4      controllable { Controller }
5      non-spontaneous events {
6          Controller.pickedUpItem
7          Controller.arrivedAt
8          Controller.releasedItem
9          Controller.pressingFinished
10         RobotArm.setCarriesItem
11         RobotArm.setLocation
12         Press.setHasItem
13     }
14     collaboration FeedBeltBehavior {
15         static role Controller controller
16         static role ConveyorBelt feedBelt
17         static role RobotArm feedArm
18         static role Press press
19
20         guarantee scenario BlankArrives { // G1
21             feedBelt -> controller.blankArrived()
22             wait [feedArm.location == feedBelt && !feedArm.carriesItem]
23             urgent controller -> feedArm.pickUp()
24         }
25         guarantee scenario ArmDeliversItemToPress { // G2
26             feedArm -> controller.pickedUpItem()
27             urgent controller -> feedArm.moveTo(press)
28             feedArm -> controller.arrivedAt(press)
29             wait [!press.hasItem]
30             urgent controller -> feedArm.releaseItem()
31             feedArm -> controller.releasedItem()
32             urgent controller -> feedArm.moveTo(feedBelt)
33         }
34         ... // new blanks are picked up before next one arrives (A1)
35     }
36     collaboration PressBehavior {
37         static role Controller controller
38         static role RobotArm feedArm
39         static role RobotArm depositArm
40         static role Press press
41
42         guarantee scenario PressStartsPressing { // G3
43             feedArm -> controller.releasedItem()
44             urgent controller -> press.startPressing()
45         }
46         guarantee scenario PickUpPressedItem { // G4
47             press -> controller.pressingFinished()
48             wait [depositArm.location == press && !depositArm.carriesItem]
49             urgent controller -> depositArm.pickUp()
50         }
51         assumption scenario PressEventuallyFinishes { // A2
52             controller -> press.startPressing()
53             strict eventually press -> controller.pressingFinished()
54         }
55     }
56     collaboration DepositBeltBehavior {
57         ... // deposit arm transports pressed items (G5); similar to G2
58     }
59     collaboration RobotArmBehavior {
60         dynamic role Controller controller
61         dynamic role RobotArm arm
62         dynamic role Location targetLocation
63         static role Press press
64
65         assumption scenario ArmMovesToLocation { // A3
66             controller -> arm.moveTo(bind targetLocation)
67             strict eventually arm -> controller.arrivedAt(targetLocation)
68             strict committed arm -> arm.setLocation(targetLocation)
69         }
70         ... /* arm picks up item (A4) and arm releases item (A5); both similar
71         to A3 */
72     }
73 }
```

Listing 1. Excerpt of a specification for our production system example; some scenarios have been omitted for brevity

51–54) is activated by a *startPressing* event, it will point to line 53, indicating that this scenario waits for a *pressingFinished* event. When an event enabled in an active scenario occurs, the reference to this enabled event advances to the next event. When all references advance past the last event in a scenario, it terminates.

Roles (e.g., lines 15–18) are used similarly to lifelines in sequence diagrams. Static roles are bound when the system is initialized and dynamic roles are bound when an active scenario is created. Binding a role means assigning an object from the object model to this role. The abstraction through roles allows reusing the same specification for different object models modeling different configurations of the same type of system, e.g., production systems with varying numbers of robots. In lines 60–69 the use of dynamic roles is shown. Any object of the proper class from the object model can be bound to these roles. As an example, when an object of class *Controller* sends the event *moveTo* to an object of class *RobotArm*, an active instance of the scenario *ArmMovesToLocation* (lines 65–69) is created. In this active scenario, the role *controller* is played by the object which sent the initial event and the role *arm* is played by the object which received the event. Dynamic roles can even be bound to parameters (line 66) or to an object referenced by an object already bound to a role (not shown). Multiple copies of the same scenario with different role bindings can be active concurrently.

Events use different keywords to enforce *liveness* and *safety conditions*. Events flagged as *committed*, *urgent*, or *eventually* must not be enabled forever. Committed and urgent events must occur immediately, allowing only other committed or urgent events to occur beforehand. Committed events take priority over urgent events. An event which must occur eventually can occur at an arbitrary time in the future, i.e., the system can choose to wait. *Strict* events enforce a strict order. Events which occur out of order generally terminate a scenario early by *interrupting* it. If line 22 in an active scenario is enabled and *blankArrived* occurs (line 21; same active scenario), this active scenario is interrupted. However, if at least one enabled event is strict, an interruption causes a safety violation instead. Safety violations must never occur.

Additional keywords offer flow control. *Wait* is used to wait for a certain condition to be satisfied before the next message is enabled. The keywords *interrupt* and *violation* can be used to specify conditions, which are checked when the event becomes enabled and may cause an interruption or a safety violation. If the condition is not satisfied, the next event is immediately enabled instead. Furthermore, there are *while* (repeat an event sequence while a condition holds), *alternative* (branching within a scenario), and *parallel* (concurrent event sequences). *Collaborations* are used to group scenarios together and do not have any semantic implications beyond providing a scope for roles.

We implemented SML and algorithms for simulating and analyzing SML specifications as a collection of Eclipse plug-ins called ScenarioTools. We use the Eclipse Modeling Framework (EMF) [27] and leverages this to integrate other powerful tools such as OCL [30] and Henshin [3]. This enables engineers to enhance SML specification with tools they already are familiar with while

still being able to use ScenarioTools' simulation and analysis features, e.g., checking if a specification is realizable.

4 Controller Synthesis

In this section we give an overview of how controller synthesis works. We briefly explain the play-out algorithm, which gives our specifications execution semantics used for simulation, analysis, and controller synthesis, and how it induces a state space. Furthermore, we briefly explain controller synthesis, that is, generating a strategy for the system to behave such that it fulfills a given specification.

4.1 Play-Out

The play-out algorithm [18,19] defines how scenarios can be interwoven into valid event sequences. Basically, the algorithm waits for the environment to choose an event, activates and progress scenarios accordingly, and then picks a reaction which is valid according to all active scenarios. Environment events can either be spontaneous events or enabled non-spontaneous events. They are events sent by uncontrollable objects. When at least one system event with a liveness condition, e.g., *urgent*, sent by a controllable object is enabled, play-out will pick one of these events. It honors particular priorities such as picking committed events first. In case all such events are flagged as *eventually*, the algorithm may also choose to wait for further environment events. Events are considered to be blocked if they would directly lead to a safety violation due to the strictness of an enabled event. The play-out algorithm never picks blocked events. A sequence of events sent by system objects enclosed by one environment event on either end is called a *super step*.

For any given set of scenarios and a given object model the play-out algorithm generally has multiple valid events to choose from at any point. It is non-deterministic. This property induces a state space or graph as shown in Fig. 2, an excerpt of the graph of our production system example (cf. Sects. 2 and 3). Each node represents a *state*, characterized by its active scenarios and the attribute values of all objects. Every edge/transition represents an event. While the edge labels in Fig. 2 seem to reference roles from the SML specification, they actually reference objects from the object model. The events in such a state graph are always concrete events sent from one object to another object using concrete parameter values (if applicable).

Such a state space is actually a game graph. Each state is either controllable by the system (= has only controllable outgoing transitions) or by the environment (= has only uncontrollable outgoing transitions). These two players play against each other. The system tries to fulfill its guarantees infinitely often given that the assumptions hold. More precisely, for every guarantee, it always tries to reach states in which no liveness condition must be fulfilled and to reach them via a sequence of events which do not cause a safety violation. The environment aims for the opposite. It tries to fulfill all assumptions the same way the system

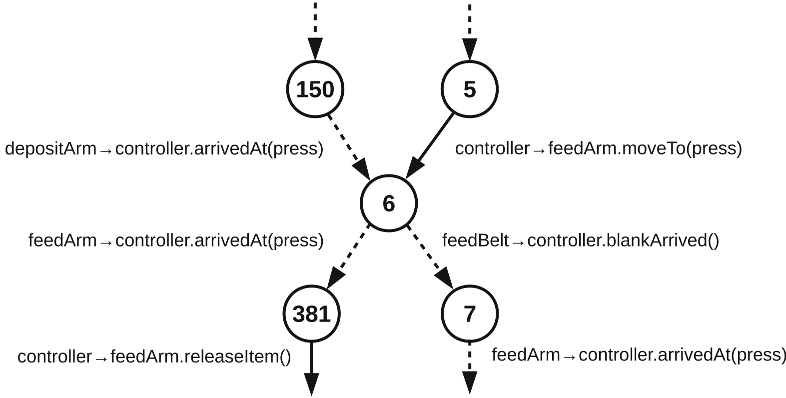

Fig. 2. Excerpt of a game graph induced by our example specification. Controllable/system events are represented by solid arrows. Uncontrollable events are represented by dashed arrows. *Set*-events, e.g., *setLocation*, have been omitted for brevity

fulfills the guarantees, but at the same time it tries to force the system to violate at least one of the guarantees. This type of game is called a *GR(1) game*. We impose an additional goal on the system, in particular we enforce the condition that each super step must be finite to ensure that the system will eventually be able to react to external events from the environment again.

4.2 Synthesis

Our controller synthesis is an implementation of Chatterjee's attractor-based General Reactivity of rank 1 (GR(1)) game solving algorithm [9]. A GR(1) condition is based on assumptions and guarantees. Formally, as Linear Temporal Logic [25] formula, it is

$$\left(\bigwedge_i \Box\Diamond a_i\right) \implies \left(\bigwedge_j \Box\Diamond g_i\right) \tag{1}$$

with a_i = "assumption i is satisfied" and g_j = "guarantee j is satisfied". Informally, this formula is true iff at least one assumption can only be fulfilled finitely often (i.e., goal states of this assumption are only visited a finite number of times in any infinite execution of the system) or all guarantees can be fulfilled infinitely often.

We map our specifications to a GR(1) condition by mapping active assumption scenarios to assumptions a_i and by mapping active guarantee scenarios to guarantees g_j. The goal states a_i of an active assumption scenario Sc are all those states in which Sc has no liveness condition to fulfill and has never been violated (tracked via a Boolean flag). Guarantee scenarios are mapped analogously. Additionally, we introduce an extra guarantee whose goal states are all

environment controlled states to ensure that all super steps are finite for well-separated specifications. In a well-separated specification [24], the system cannot force the environment into a violation of the assumptions by any action it takes. Well-separation is a desirable property of a good specification.

Chatterjee's aforementioned game solving algorithm uses the assumptions' and guarantees' goal states to calculate *attractors*. Attractors of a condition are all states from which a player can guarantee reaching a goal state of this condition. A system attractor of g_j is a state from which the system can ensure to visit a goal state of g_j regardless of the environment's behavior. Chatterjee's algorithm iteratively removes environment dominions from the game graph. Environment dominions are subsets of the game graph in which the environment can fulfill all assumptions but the system is unable to fulfill at least one of the guarantees. Environment dominions are identified by finding states which are not system attractors for at least one g_j. Using the environment attractors of all a_i, Chatterjee's algorithm determines if the environment can fulfill all assumptions in the subgraph defined by the non-attractor states of aforementioned g_j. These iterations are performed until the game graph cannot be reduced further.

The states retained after the algorithm finishes are called *winning states*. They contain a strategy in which the system can guarantee to fulfill the GR(1) condition defined by all assumptions and guarantees. If the initial state of the game graph is a winning state, the specification is *realizable*, i.e., the requirements and behavior defined by the scenarios are consistent. Using the same attractor approach, we can extract a *strategy* (also: *controller*) from the winning states. A strategy is similar to a game graph but contains exactly one outgoing transition for each controllable state (Fig. 2 happens to be a strategy). It deterministically specifies what the system must do for any valid environment. These strategies serve as the basis for generating Structured Text to execute on a PLC.

5 Generating Executable Code

In this section we describe how to generate Structured Text from a synthesized controller which is correct by construction. A synthesized controller contains some events which are only necessary for defining and checking a GR(1) condition but which serve no purpose in the generated PLC code. Thus, we explain a pre-processing step of the controller to reduce it to events of interest for code generation. After that, we describe how to generate executable PLC and finish the section with a discussion of possible extensions to our approach.

5.1 Pre-processing the Controller

Figure 3 shows an excerpt of a synthesized controller including a *setLocation* event which is required to be able to express conditions such as the *wait* condition in line 48 of Listing 1. However, this event is not useful for code generation and should be removed, as shown in Fig. 2. In general, expert knowledge of the domain is necessary to identify events to remove and thus an engineer should be

able to provide a list of such events. A tool can still provide helpful suggestions for removal based on heuristics, though. We propose two heuristics, (1) events sent by uncontrollable objects to other uncontrollable objects, and (2) *set*-events. Either of these two heuristics would be sufficient to propose the proper list of events to remove to the engineer in our example specification. When removing events, transitions have to be updated, such as the outgoing transition of state 150 in Fig. 3 which must point directly to state 6 after the removal of 151, which is no longer necessary after removing *setLocation(*)*.

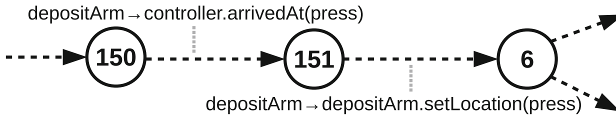

Fig. 3. Variant of Fig. 2 including a *setLocation* event previously omitted.

In Structured Text, components are controlled by setting the appropriate input attributes of function blocks, e.g., a block representing a specific robot arm of the system, and waiting for the output attributes to be set to values signaling that the desired action has been performed. The paradigm is: a component is instructed to do something (setting of input attributes) and it signals when it is done (setting of output attributes). In our approach, we adopt this paradigm by having the engineer define event pairs which correspond to "do X" and "X is done". Such a pair is shown in Fig. 2: *moveTo(press)* (transition from 5 to 6) and *arrivedAt(press)* (transition from 6 to 381; also outgoing transition of 7). These event pairs are characterized by a controllable object S (here: *controller*) sending an event to an uncontrollable object E (here: *feedArm*), instructing E to perform an action (here: *moveTo(press)*). Later, E signals back to S that it is now done performing this action. Again, heuristics can be used to support the engineer in defining these pairs. We observed that these pairs often occur in adjacent lines in scenarios, e.g., lines 27–28 and 30–31 in Listing 1. These event pairs are necessary in the next step, the actual code generation.

5.2 Generating Structured Text

We use the pre-processed controller and event pair definitions provided by the engineer to generate Structured Text which is executable on PLCs. For simplicity, we assume that there is exactly one controllable object in the system, e.g., the controller shown in the center of Fig. 1. Our generated code consists of multiple state machines. We translate the pre-processed controller to one state machine representing the controllable object. We call this the *primary* state machine, as it governs the whole process: it tells each component, via the other state machines, when to perform which action. We furthermore generate one state machine for each uncontrollable object which receives events, i.e., represent components having to perform an action. These state machines, called *secondary* state machines,

```
1  CASE controllerState OF // primary state machine
2      0:
3          // idle
4      1:
5          ... // omitted for brevity
6      5:
7          feedArmState := 1;
8          controllerState := 6;
9      6:
10         IF feedBelt_controller_blankArrived THEN
11             feedBelt_controller_blankArrived := FALSE;
12             controllerState := 7;
13         ELSIF feedArm_controller_arrivedAt_press THEN
14             feedArm_controller_arrivedAt_press := FALSE;
15             controllerState := 381;
16         END_IF
17     7:
18         ... // omitted for brevity
19 END_CASE
```

Listing 2. Generated PLC code (Structured Text) of the primary state machine

are much simpler. They consist of an idle state, which is their initial state, and one additional state for each action that must be performed. Listings 2 and 3 show examples of the generated code.

Events sent by uncontrollable objects are mapped to Boolean variables, e.g., *feedBelt_controller_blankArrived* which corresponds to the sensor event triggered by the arrival of a new blank item. These variables are used by the primary state machine to decide when to switch to which state (lines 10–16 in Listing 2). This state machine instructs the secondary state machines to perform actions as called for by the synthesized controller, e.g., lines 7–8 correspond to the transition from state 5 to 6 in Fig. 2. The previously defined event pairs are used to generate this code. Based on the knowledge that *controller→feedArm.moveTo(press)* and *feedArm→controller.arrivedAt(press)* are a pair, line 7 can be generated to instruct the feed arm's state machine (Listing 3) to switch to the proper state to perform this action. The same pair definition is used to generate line 9 in the secondary state machine, in which the feed arm informs the controller via a Boolean variable that is done performing the desired action. This separation into primary and secondary state machines allows any arbitrary combination of actions to be performed concurrently by different components.

Separating the generated state machines into different code files has proven to be a good practice when regeneration of the PLC code is a concern. By keeping state machines separate and the order of states in the secondary state machines deterministic and consistent, only the (fully generated) primary state machine has to be replaced after regenerating the PLC code. More elaborate changes

```
1  CASE feedArmState OF // secondary state machine for feed
       arm
2      0:
3          // idle
4      1:
5          // controller->feedArm.moveTo(press)
6          feedArmFB.xMoveRelExecute := TRUE; // perform
               example action
7          IF feedArmFB.xFunDone THEN // is example action
               done?
8              feedArmFB.xMoveRelExecute := FALSE; // clean-
                   up after example action
9              feedArm_controller_arrivedAt_press := TRUE;
10             feedArmState := 0;
11         END_IF
12     2:
13         // controller->feedArm.moveTo(feedBelt)
14         ... // omitted for brevity
15 END_CASE
```

Listing 3. Generated PLC code (Structured Text) of a secondary state machine

to the model, such as adding or removing actions components must perform, require some manual migration effort when regenerating code.

The primary state machine is fully generated and does not need to be modified. The secondary state machines are however actually only stubs after generation. Listing 3 shows an example after an engineer manually added the code in lines 6 and 8 and the condition in line 7. In general, after generating the Structured Text from a synthesized controller, each state in the secondary state machines contains some boiler plate code, in particular the if-statement with an empty condition but a body that already sets the appropriate Boolean and state variables (lines 9–10), and some comments telling the engineer which atomic action should be performed in this state. These stubs can easily be extended by an engineer by setting and checking the inputs and outputs of the appropriate function block. The proper function block definitions, as well as any initializations, have to be added manually as they are platform-specific. Additionally, code for checking sensor events which are not part of an event pair, e.g., when to set *feedBelt_controller_blankArrived* to *TRUE*, has to be added manually.

When generating PLC code from Listing 1 and using rotations at varying directions and speeds of a single axis (one for each component, i.e., both robot arms and the press) to represent actions such as movement or picking up a work item, 59 lines of code had to be written manually, 9 lines of code had to be modified (conditions which check whether an action has been performed successfully), and 1355 lines of code were generated automatically. While this is not an exhaustive evaluation, these numbers already point towards a significant

reduction in the required manual implementation effort. In particular, the complex interleaving of concurrent actions and events is fully generated.

5.3 Extensions

We assumed that there is only one controllable object in the system. As an extension to support multiple controllable objects, i.e., multiple software controllers, we are looking into algorithms to create multiple distributed controllers which automatically synchronize with each other when necessary.

Event pairs are defined during pre-processing. This implies that only the success case, i.e., the action can actually be performed, can be modeled. Instead, defining a mapping from controllable events (instructions) to sets of uncontrollable events (outcomes of the instructions) can easily rectify this. Different outcomes for each action can be defined and the specification can include appropriate reactions for each possible outcome.

By including checks of the Boolean variables of environment events, which are not expected to occur in a given state of the primary state machine, violations of the assumptions can be detected. These could be used to put the system into an emergency state which performs a shut down procedure.

6 Related Work

There exists previous work on synthesizing controllers from LSC/SML-style scenarios [6,8,17,29], and other forms of scenarios [22,31]. Most of these approaches produce finite state controllers or state machines as output, from which code can be generated. Some consider code generation from such synthesized controllers in particular for robotics/embedded applications [4,21].

The novelty of our synthesis procedure w.r.t. to the above is, first, that it supports scenario-based specifications with a greater expressive power—assume/guarantee specifications with multiple liveness objectives (GR(1)). Second, we describe a scenario-based modeling and code generation methodology that specifically targets the typical structure and nature of PLC software.

There is work on generating PLC code from state machines [26] or Petri nets [12,28], and formal methods are used also for verifying PLC code [5,13].

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 [2,7,11,23]. In contrast to temporal logics based approaches, LSCs/SML aim to provide a more intuitive language that is easier to use.

In previous own work, we considered the direct execution of SML specifications as *scenarios@run.time* [15]. Here, the scenarios are executed without the prior synthesis of a finite-state controller. Such an approach has advantages and disadvantages. For example, the prior synthesis does not only detect specification flaws, but a synthesized controller can also contain the solution for resolving issues related with under-specification. On the other hand, controller synthesis, due to its computational complexity, may not be possible for larger specifications, in which case direct execution is a valuable option.

7 Conclusion

In this paper we presented an approach for generating Structured Text executable on PLCs commonly found in the industry. We generate this code from scenario-based specifications written in an intuitive DSL we developed. Using this DSL, called Scenario Modeling Language (SML), engineers can easily define requirements, desired behavior, and environment assumptions of a system. These are defined in the form of assumption and guarantee scenarios, which have to be fulfilled infinitely often, i.e., SML specifications express GR(1) conditions, giving engineers a powerful class of conditions to express their goals in. The generated code, which is correct by construction, 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 an implementation of the desired process having already been generated.

Acknowledgment. This research is funded by the DFG project EffiSynth.

References

1. Alexandron, G., Armoni, M., Gordon, M., Harel, D.: Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 311–320 (2014)
2. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
4. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schäfer, W., Meyer, M., Pohlmann, U., Priesterjahn, C., Tichy, M.: The MechatronicUML design method - process and language for platform-independent modeling (2014)
5. Biallas, S., Brauer, J., Kowalewski, S.: 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, September 2012
6. Bontemps, Y., Heymans, P.: From live sequence charts to state machines and back: a guided tour. IEEE Trans. Softw. Eng. **31**(12), 999–1014 (2005)
7. Braberman, V., D'Ippolito, N., Piterman, N., Sykes, D., Uchitel, S.: Controller synthesis: from modelling to enactment. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, Piscataway, NJ, USA, pp. 1347–1350. IEEE Press (2013)
8. Brenner, C., Greenyer, J., Schäfer, W.: On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 51–65. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_4

9. Chatterjee, K., Dvorák, W., Henzinger, M., Loitzenbauer, V.: 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). Leibniz International Proceedings in Informatics (LIPIcs), vol. 58, pp. 25:1–25:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016)
10. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Formal Methods Syst. Des. **19**, 45–80 (2001)
11. Ehlers, R., Raman, V.: `Slugs`: extensible GR(1) synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 333–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_18
12. Frey, G.: Automatic implementation of Petri net based control algorithms on PLC. In: Proceedings of the 2000 American Control Conference, ACC (IEEE Cat. No.00CH36334), vol. 4, pp. 2819–2823 (2000)
13. Frey, G., Litz, L.: Formal methods in PLC programming. In: 2000 IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2431–2436 (2000)
14. Gordon, M., Marron, A., Meerbaum-Salant, O.: Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In: Proceedings of the 17th Conference on Innovation and Technology in Computer Science Education (ITICSE), pp. 198–203 (2012)
15. Greenyer, J., Gritzner, D., Gutjahr, T., Duente, T., Dulle, S., Deppe, F.D., Glade, N., Hilbich, M., Koenig, F., Luennemann, J., Prenner, N., Raetz, K., Schnelle, T., Singer, M., Tempelmeier, N., Voges, R.: Scenarios@run.time - distributed execution of specifications on IoT-connected robots. In: Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015. CEUR Workshop Proceedings (2015)
16. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: 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 International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), vol. 1725, pp. 16–32. CEUR (2016)
17. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Found. Comput. Sci. **13**(1), 5–51 (2002)
18. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-19029-2
19. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. SoSyM **2**, 82–107 (2003)
20. Kleppe, A.G., Warmer, J.B., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, Boston (2003)
21. La Manna, V.P., Greenyer, J., Clun, D., Ghezzi, C.: Towards executing dynamically updating finite-state controllers on a robot system. In: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, MiSE 2015, Piscataway, NJ, USA, pp. 42–47. IEEE Press (2015)
22. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, SCESM 2006, pp. 5–12. ACM, New York (2006)

23. Maoz, S., Ringert, J.O.: 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 (2015)
24. Maoz, S., Ringert, J.O.: On well-separation of GR(1) specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 362–372. ACM (2016)
25. Pnueli, A.: The temporal logic of programs. In: 1977 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
26. Sacha, K.: Automatic code generation for PLC controllers. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 303–316. Springer, Heidelberg (2005). https://doi.org/10.1007/11563228_23
27. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)
28. Thapa, D., Dangol, S., Wang, G.N.: 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 2006), vol. 2, pp. 228–233, November 2005
29. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Trans. Softw. Eng. **35**(3), 384–406 (2009)
30. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, Boston (2003)
31. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, pp. 314–323 (2000)