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

Daniel Gritzner and Joel Greenyer

Leibniz Universität Hannover, Fachgebiet Software Engineering,
Welfengarten 1, D-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. In 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 makes it difficult to develop robot software which is free of defects, i.e., which, under all possible circumstances, makes the robot act or react properly. The specification, from which an implementation is derived, may be inconsistent and the 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 total energy consumption of a system, e.g., by synchronizing the robots' movement such that their braking energy can be leveraged instead of being wasted as heat.

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 goals/requirements, desired behavior and assumptions made about the environment. Scenarios are sequences of events, similar to how engineers describe requirements to each other: *"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 forcontroller 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.

In this paper we present an approach for generating executable code for Programmable Logic Controllers (PLCs) from aforementioned scenario-based specifications. This enables engineers to use formal methods, e.g., checking if all requirements are consistent, to ensure the correctness of the specification and then 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 implementation of atomic actions to engineers. When looking at our approach 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 later 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. Sect. 2 introduces an example used for explanation and discussion throughout the paper. Sect. 3 and 4 introduce scenario-based modeling and controller synthesis. Sect. 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 Sect. 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.

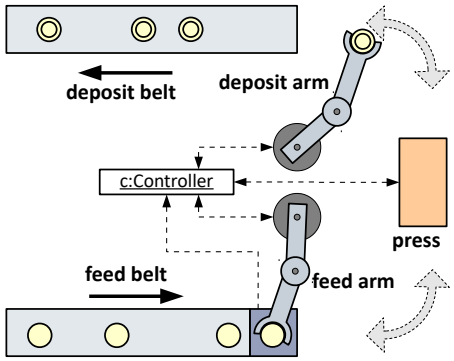

Fig. 1. A production system consisting of two robot arms, each adjacent to a conveyor belt, a press, and a software-based controller sending instructions to other component as well as processing their sensor inputs.

The specification for this example models the following goals **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.

Goals **G1-G5** define the systems desired behavior 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 waiting at the press.

Assumption **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 the system is able to handle its workload without a queue of unprocessed blanks forming 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.

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 DSL we developed for modeling scenarios, called the Scenario Modeling Language (SML).

3.1 Scenario Modeling Language

The Scenario Modeling Language (SML) [16] is a DSL we developed to offer 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. Listings 1 shows the specification of our production system example. Comments next to each scenario indicate which goal or assumption they represent. Some lines have been omitted for brevity.

A specification references a domain model (line 1) and has a name (line 2). The domain model is a class model of the different components in the specified system. In our example it contains classes such as *RobotArm* and *Press*. Theses classes model each component type's attributes and possible events it can receive. Events can be actions it should perform, e.g., a *RobotArm* can pick up an item, or sensor events it may be notified of, e.g., the *Controller* can be 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*, sometimes 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 *active scenario*, i.e., an instance of a scenario. Active scenarios have one or more references to enabled events, i.e., events which are expected next. When *PressEventuallyFinishes* (lines 51-54) is activated by a *startPressing*

```
import "../model/productioncell.ecore"
specification ProductioncellSpecification {
    domain productioncell
    controllable { Controller }
    non-spontaneous events {
        Controller.pickedUpItem
        Controller.arrivedAt
        Controller.releasedItem
        Controller.pressingFinished
        RobotArm.setCarriesItem
        RobotArm.setLocation
        Press.setHasItem
    }
    collaboration FeedBeltBehavior {
        static role Controller controller
        static role ConveyorBelt feedBelt
        static role RobotArm feedArm
        static role Press press

        guarantee scenario BlankArrives { // G1
            feedBelt -> controller.blankArrived()
            wait [feedArm.location == feedBelt && !feedArm.carriesItem]
            urgent controller -> feedArm.pickUp()
        }
        guarantee scenario ArmDeliversItemToPress { // G2
            feedArm -> controller.pickedUpItem()
            urgent controller -> feedArm.moveTo(press)
            feedArm -> controller.arrivedAt(press)
            wait [!press.hasItem]
            urgent controller -> feedArm.releaseItem()
            feedArm -> controller.releasedItem()
            urgent controller -> feedArm.moveTo(feedBelt)
        }
        // new blanks are picked up before next one arrives (A1)
    }
    collaboration PressBehavior {
        static role Controller controller
        static role RobotArm feedArm
        static role RobotArm depositArm
        static role Press press

        guarantee scenario PressStartsPressing { // G3
            feedArm -> controller.releasedItem()
            urgent controller -> press.startPressing()
        }
        guarantee scenario PickUpPressedItem { // G4
            press -> controller.pressingFinished()
            wait [depositArm.location == press && !depositArm.carriesItem]
            urgent controller -> depositArm.pickUp()
        }
        assumption scenario PressEventuallyFinishes { // A2
            controller -> press.startPressing()
            strict eventually press -> controller.pressingFinished()
        }
    }
    collaboration DepositBeltBehavior {
        // deposit arm transports pressed items (G5); similar to G2
    }
    collaboration RobotArmBehavior {
        dynamic role Controller controller
        dynamic role RobotArm arm
        dynamic role Location targetLocation
        static role Press press

        assumption scenario ArmMovesToLocation { // A3
            controller -> arm.moveTo(bind targetLocation)
            strict eventually arm -> controller.arrivedAt(targetLocation)
            strict committed arm -> arm.setLocation(targetLocation)
        }
        // arm picks up item (A4) and arm releases item (A5); both similar to A3
    }
}
```

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

event, it will point to line 53, indicating that this scenario waits for a *pressingFinished* event. When an event occurs, which corresponds to an enabled event in an active scenario, 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 an object model, itself an instance of the same domain model, to this role. The abstraction through roles allows reusing the same specification for different object models which model different configurations of the same type of system, e.g., production systems with varying numbers of robot arms. 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. E.g., when an object of class *Controller* sends the event *moveTo* to an object of class *RobotArm*, an active instance of 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 interrupt, i.e., terminate, a scenario. If line 22 in an active scenario is enabled and *blankArrived* occurs (line 21; same active scenario), this active scenario is interrupted, i.e., terminated. 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. 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 roles only being usable within the collaboration in which they are defined.

3.2 ScenarioTools

We implemented SML and algorithms for simulating and analyzing SML specifications as a collection of Eclipse plug-ins called ScenarioTools. Scenario-

Tools is built on top of the Eclipse Modeling Framework (EMF) [27] and leverages this fact to integrate other powerful tools, e.g., OCL [30] and Henshin [3]. This enables engineers to enhance SML specification with tools they are already 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, i.e., generating a strategy for the system to behave so that it fulfills a 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 and are events sent by uncontrollable objects. When at least one system event, i.e., an event sent by a controllable object, with a liveness condition, e.g., *urgent*, is enabled, play out will pick one of these events. It honors particular priorities (e.g., picking committed events first). In case all such events are flagged as *eventually*, the play out may also choose to wait for the environment to act first. Events, which due to strictness of an enabled event would directly lead to a safety violation, are considered to be blocked. The play out algorithm never picks blocked messages. 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 different valid events to choose from at any point. It is non-deterministic. This properties induces a state space or graph as shown in Fig. 2, an excerpt of the graph of our production system example (cf. Sect. 2 and 3). Each node represents a *state*, characterized by its active scenarios and the attribute values of all objects, and each edge/transition represents an event.

This state space is 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 goals infinitely often given the assumptions hold. I.e., 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 of the goals. The environment aims for the opposite. It tries to fulfill all assumptions the same way the system fulfills its goals. But at the same time it tries to force the system to violate at least one of its goals. This type of game is

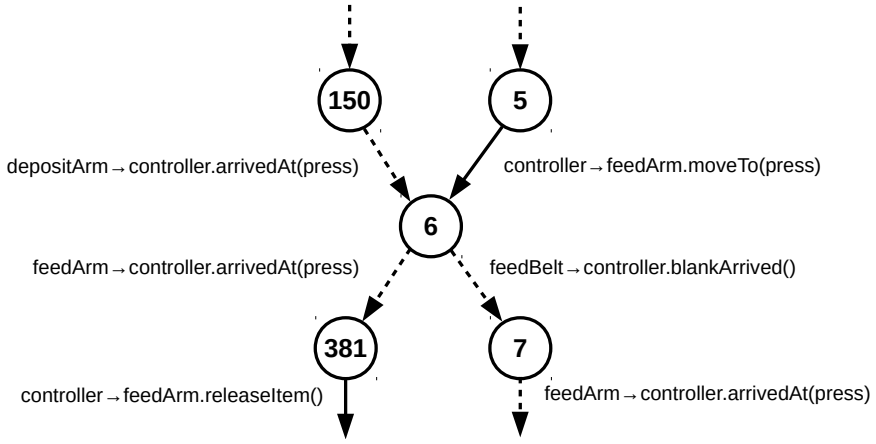

Fig. 2. Excerpt of a game graph induced by our example specification. Controllable events, i.e., events send by controllable objects, are represented by solid arrows. Uncontrollable events are represented by dashed arrows. *Set*-events, e.g., *setLocation*, have been omitted for brevity.

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, i.e., the system must eventually wait for external events again.

4.2 Synthesis

Our controller synthesis is an implementation of Chatterjee's attractor-based GR(1) game solving algorithm [9].

A General Reactivity of rank 1 (GR(1)) condition is based on assumptions and guarantees. Formally, as Linear Temporal Logic [25] formula, it is

$$\left(\bigwedge_i \Box\Diamond a_i\right) \implies \left(\bigwedge_j \Box\Diamond g_i\right)$$

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 run 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 all states, in which this scenario 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 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 conditions are all states from which a player can guarantee reaching a goal state of this condition. E.g., 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 cannot fulfill at least one guarantee. 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. Using a controller synthesized from a scenario-based specification as a basis implies that the generated code fulfills all requirements as intended and that the requirements are consistent. A synthesized controller contains 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

Fig. 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

thus 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 removal of 151, which is no longer necessary after removing *setLocation(*)*.

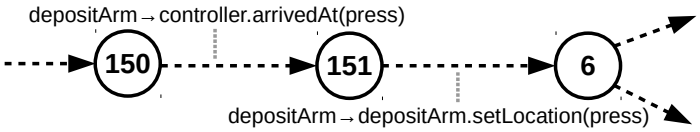

Fig. 3. Variant of Fig. 2 including a *setLocation* event previously omitted.

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, e.g., *controller*, sending an event to an uncontrollable object E, e.g., *feedArm*, instructing E to perform an action, e.g., *moveTo(press)*. Later, E signals back to S that 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, which is to actually generate code from the reduced controller.

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, 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. Listing 2 shows an example of the generated code.

```
1  CASE controllerState OF // primary state machine
2      0:
3          // perform start up; initialize components
4      1:
5          // ...
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         // ...
19 END_CASE
20 CASE feedArmState OF // secondary state machine for feed arm
21     0:
22         // idle
23     1:
24         // controller->feedArm.moveTo(press)
25         feedArmFB.xMoveRelExecute := TRUE; // perform example action
26         IF feedArmFB.xFunDone THEN // is example action done?
27             feedArmFB.xMoveRelExecute := FALSE; // clean-up after example action
28             feedArm_controller_arrivedAt_press := TRUE;
29             feedArmState := 0;
30         END_IF
31     2:
32         // controller->feedArm.moveTo(feedBelt)
33         // ...
34 END_CASE
35 CASE depositArmState OF
36     // ...
37 END_CASE
38 CASE pressState OF
39     // ...
40 END_CASE
41 // function blocks
42 feedArmFB(...);
43 depositArmFB(...);
44 pressFB(...);
```

Listing 2. Generated PLC code (Structured Text; excerpt)

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 (lines 1-19) to decide when to switch to which state (lines 10-16). 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 (lines 20-34) to switch to the proper state to perform this action. The same pair definition is used to generate line 28, in which the feed arm state machine informs the primary state machine 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.

The primary state machine is fully generated and does not need to be modified, unless specific operations need to be performed during start up (state 0, which is empty by default; lines 2-3). The secondary state machines are however actually only stubs after generation. Listing 2 shows an example after an engineer manually added the code in lines 25 and 27 and the condition in line 26. 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 28-29), and some comments telling the engineer which atomic action should be performed in this state. These stubs can then easily be extended by an engineer by setting and checking the inputs and outputs of the appropriate function block. The proper function block definitions (lines 42-44) 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 as well.

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, multiple variants of the specification can be generated, each modeling only one software controller as controllable and treating the other controllers as environment objects. We are looking into algorithms supporting engineers with this step. Keeping a distributed system properly synchronized is a difficult task.

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 to each possible outcome, e.g., reacting differently when an error is detected.

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 [17,6,29,8], and other forms of scenarios [31,22]. 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 [13,5].

Related is also other work on controller synthesis based on temporal logic specification, such as LTL and its GR(1) fragment; some specifically consider synthesis and code generation for robotics applications [7,23,11,2]. In contrast to temporal logics based approaches, LSCs/SML aims to be more intuitive.

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 bigger specifications, and then 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: Proc. 36th Int. Conf. on Software Engineering (ICSE). pp. 311–320 (2014)
2. Alur, R., Moarref, S., Topcu, U.: Compositional Synthesis of Reactive Controllers for Multi-agent Systems, pp. 251–269. Springer International Publishing, Cham (2016)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. 13th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 121–135 (2010)
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: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 338–341 (Sept 2012)
6. Bontemps, Y., Heymans, P.: From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE Transactions on Software Engineering 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. pp. 1347–1350. ICSE '13, IEEE Press, Piscataway, NJ, USA (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.) Fundamental Approaches to Software Engineering (FASE 2015), Lecture Notes in Computer Science, vol. 9033, pp. 51–65. Springer (2015)
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. In: Formal Methods in System Design. vol. 19, pp. 45–80 (2001)
11. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) Synthesis, pp. 333–339. Springer International Publishing, Cham (2016)
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 vol.4 (2000)
13. Frey, G., Litz, L.: Formal methods in plc programming. In: Systems, Man, and Cybernetics, 2000 IEEE International Conference on. vol. 4, pp. 2431–2436 vol.4 (2000)
14. Gordon, M., Marron, A., Meerbaum-Salant, O.: Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In: Proc. 17th Conf. 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. Foundations of Computer Science 13:1, 5–51 (2002)
18. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
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 (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. pp. 42–47. MiSE '15, IEEE Press, Piscataway, NJ, USA (2015)
22. Liang, H., Dingel, J., Diskin, Z.: 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. pp. 5–12. SCESM '06, ACM, New York, NY, USA (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: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)
26. Sacha, K.: Automatic Code Generation for PLC Controllers, pp. 303–316. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
27. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (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'06). vol. 2, pp. 228–233 (Nov 2005)
29. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. IEEE Transactions on Software Engineering 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 (2003)
31. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd international conference on Software engineering, ICSE '00. pp. 314–323 (2000)