MGSyn: Automatic Synthesis for Industrial Automation

Chih-Hong Cheng[1,2,⋆], Michael Geisinger[2,*], Harald Ruess[2],
Christian Buckl[2], and Alois Knoll[1]

[1] Department of Informatics, Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München, Germany
[2] fortiss GmbH, Guerickestr. 25, 80805 München, Germany

Abstract. MGSyn is a programming toolbox based on game-theoretic notions of synthesis for generating production code in the domain of industrial automation. Instead of painstakingly engineering sequences of relatively low-level program code, the designer selects pre-defined hardware components together with behavioral interfaces from a given library, specifies a topology for the interconnection of components, and specifies the programming/synthesis problem in terms of what needs to be achieved. Given the model and a problem specification, MGSyn synthesizes executable C/C++ code for a concrete execution platform and an interactive simulator. The synthesized code is used to control distributed industry-standard PLCs in a FESTO modular production system.

1 Introduction

Realizing distributed process control systems with their stringent real-time and dependability, in particular safety and security requirements, is a challenging problem. The prevalent state-of-the-practice, as determined by current industrial standards including IEC 61131-3, IEC 61804, or IEC 61499, is based on painstakingly engineering sequences of relatively low-level control code using standardized libraries of function blocks. All too often this traditional style of programming leads to inefficiencies in developing and maintaining industrial production control systems, it has negative impact on the quality and dependability of the control code itself, and it results in inflexibility of production processes with prolongated start-up and changeover times.

We are proposing a new style of programming industrial automation plants based on describing *what* needs to be achieved instead of *how* these plants are actually being controlled. An example of such high-level instructions is “Drill and store red work pieces if they are facing up.” More precisely, based on capability models of hardware components and a description of what needs to be achieved, we set up a game between the hardware component controllers and the observable (sensor) environment, and synthesize a control algorithm based on a winning strategy for the controllers. MGSyn (**M**odel, **G**ame, **Syn**thesis) is a tool for automating this high-level style of programming industrial automation plants by, first, synthesizing code for embedded control systems and, second, executing this code to control a distributed industrial programming logic control system.[1]

⋆ The first two authors contributed equally to this work.
[1] MGSyn is freely available under the GPLv3 license, including a step-by-step tutorial, at `http://www.fortiss.org/formal-methods`

Concerning related work, there is an ongoing interest in program synthesis; some recent works include [12,15,13,9,10,14]. For automation, models based on state-transition-diagrams have previously been used and synthesis is performed on a generalized Petri net models with input-output preservation [8,16,7,2]. In MGSyn, a developer provides a high-level specification for the desired behavior and the synthesis engine automatically creates a program (i.e., the state-transition-diagram) that fulfills the specification. In addition, MGSyn includes an automated deployment of such a high-level control program to lower-level executables.

We first illustrate the general modeling concepts of MGSyn in Section 2 . In Section 3, we describe the deployment of executable code for our FESTO modular production[2] demonstrator depicted in Figure 1. This industrial automation plant is built from a set of rearrangeable hardware modules including processing stations, conveyer belts, and storage facilities, similar to the ones used in large-scale production facilities. Finally, in Section 4, we describe how the back-end synthesis engine as well as the mapping to a concrete execution platform is implemented.

2 Modeling Industrial Automation Processes with Games

Given a description of the plant and a high-level problem specification, the two-player game of program synthesis for industrial automation is played between *Controller* and *Environment*. The moves of *Controller* correspond to (i) processing actions of the plant and (ii) sensor triggering actions, whereas *Environment*'s moves determine the plant's sensor inputs, and hence uncertainty and non-determinism within the system. Winning conditions of the game are specified in terms of a subset of linear temporal logic (LTL). A particularly simple winning condition, e.g., is a set of states, which are regarded as *goals* for encoding what needs to be achieved. In this case *Controller* wins the (reachability) game if it succeeds in driving the plant towards these goals irrespective of the moves (sensor inputs) chosen by *Environment*. Such a game is specified in MGSyn from models of the hardware modules, the topology of these modules and the operational behavior for specifying *Controller* and *Environment* moves; see Section 3 for a representative model. From these ingredients, the synthesis engine of MGSyn creates intermediate, platform independent control code, which may be validated through platform-independent simulation. The deployment of executable code is based on a description of execution units and their interconnection.

Plant Modules. An industrial production plant is built up from hardware modules including conveyor belts, robot arms, or rotary plates. The key is to specify each of these components together with clearly defined *behavioral interfaces*, which may also be viewed as contracts; these contracts are respected by realizations of the interface specifications. Behavioral interfaces model the players' available moves. They consist of a list of preconditions (i.e., "When is the move legal?") as well as effects of the move (i.e., "How does the move change the state of the system?"). Preconditions and effects make statements on *predicates* that in turn model the overall system state. An according set of predicates has to be added to the model when new moves are added.

[2] http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/

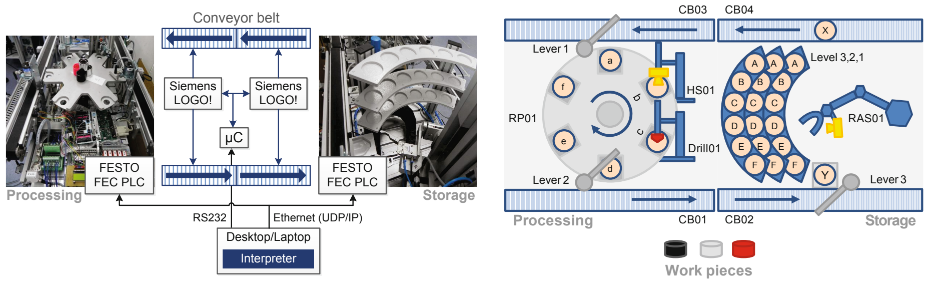


Fig. 1. The FESTO MPS demonstrator (left) and its abstract model (right)

Plant Topology. Work pieces are typically transferred between plant modules in an automation system. Therefore, we associate each hardware module with a set of *operating positions* and the topology of a plant is specified as the *overlapping of positions* among different hardware modules (see Section 3 for an example).

Plant Behavior. The so-called *problem specification* describes the desired behavior of the composed system (i.e., the goal or winning condition for *Controller*). MGSyn is restricted to specifications where controller strategies can be synthesized symbolically in time polynomial to the size of the translated game graph, ranging from reachability to *generalized reactivity-1* (GR-1) conditions [15].

Execution Platform and Networking. For generating executable program code from the intermediate, platform-independent representation as generated by the core MGSyn synthesis algorithms, it is necessary to specify the hardware setup, namely (i) the electronic control units (ECUs) that are attached to the hardware modules and (ii) the communication infrastructure.

MGSyn is implemented using the Eclipse Modeling Framework (EMF) [1]) and includes an extensible library of predefined plant modules, topologies, and behavioral interfaces. Therefore, a design engineer may specify control problems by selecting modules from this library and interconnecting them in a suitable way; the corresponding game is created automatically by the MGSyn backend engine. The synthesis engine of MGSyn is based on an extension of the GAVS+ solver library [5].

3 Example: Synthesizing Executable Code for FESTO MPS

We demonstrate the use of MGSyn for modeling and synthesizing executable code for the FESTO modular production system (MPS) in Figure 1 by means of a concrete example; see also [3]. This plant consists of the modules RobotArmStorage (RAS), ConveyorBelt (CB), Lever, RotaryPlate (RP), HeightSensor (HS), and Drill.

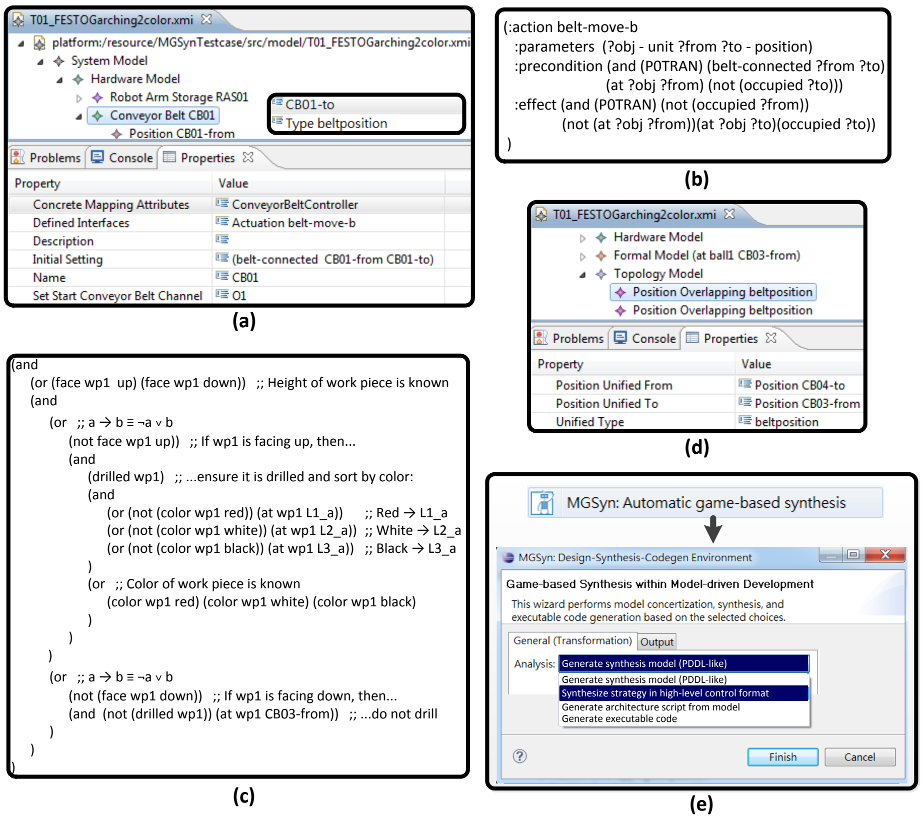
Modeling Hardware Modules. For example, conveyor belt CB01 specifies the following properties (compare Figure 2(a)):

- Two operating positions CB01-from and CB01-to.
- Initial state belt-connected CB01-from CB01-to, where belt-connected is a predicate.

- Behavioral interface belt-move-b (see Figure 2(b) for formal semantics). Intuitively, the interface enables to move a work piece from position ?from to position ?to if the hardware supports the transmission (belt-connected ?from ?to). Hence, CB01 is modeled as a unidirectional belt. at and occupied are further predicates defined in the model.

We also specify ECUs and their controlled hardware modules in the model. For example, as depicted in the left part of Figure 1, FESTO FEC PLCs control the storage and processing units, while the conveyor belts are controlled by Siemens LOGO! PLCs.

Topology Specification via Overlapping Positions. Once modules are specified, the second step is to specify their topology. For instance, for the system in Figure 1, the destination of CB04 is linked with the source of CB03. This spatial overlapping is characterized in the model (see Figure 2(d)). Similarly, the source of Lever1 overlaps



```
T01_FESTOGarching2color.xmi
  platform:/resource/MGSynTestcase/src/model/T01_FESTOGarching2color.xmi
    System Model
      Hardware Model
        Robot Arm Storage RAS01
        Conveyor Belt CB01
          Position CB01-from
```

CB01-to
Type beltposition

Property	Value
Concrete Mapping Attributes	ConveyorBeltController
Defined Interfaces	Actuation belt-move-b
Description	
Initial Setting	(belt-connected CB01-from CB01-to)
Name	CB01
Set Start Conveyor Belt Channel	O1

(a)

```
(:action belt-move-b
  :parameters (?obj - unit ?from ?to - position)
  :precondition (and (P0TRAN) (belt-connected ?from ?to)
                (at ?obj ?from) (not (occupied ?to)))
  :effect (and (P0TRAN) (not (occupied ?from))
                (not (at ?obj ?from))(at ?obj ?to)(occupied ?to))
)
```

(b)

```
(and
  (or (face wp1  up) (face wp1 down))   ;; Height of work piece is known
  (and
    (or   ;; a → b ≡ ¬a ∨ b
      (not face wp1 up))   ;; If wp1 is facing up, then...
      (and
        (drilled wp1)   ;; ...ensure it is drilled and sort by color:
        (and
          (or (not (color wp1 red)) (at wp1 L1_a))      ;; Red → L1_a
          (or (not (color wp1 white)) (at wp1 L2_a))  ;; White → L2_a
          (or (not (color wp1 black)) (at wp1 L3_a))   ;; Black → L3_a
        )
        (or   ;; Color of work piece is known
          (color wp1 red) (color wp1 white) (color wp1 black)
        )
      )
    )
    (or   ;; a → b ≡ ¬a ∨ b
      (not (face wp1 down))   ;; If wp1 is facing down, then...
      (and  (not (drilled wp1)) (at wp1 CB03-from))   ;; ...do not drill
    )
  )
)
```

(c)

```
T01_FESTOGarching2color.xmi
  Hardware Model
  Formal Model (at ball1 CB03-from)
  Topology Model
    Position Overlapping beltposition
    Position Overlapping beltposition
```

Property	Value
Position Unified From	Position CB04-to
Position Unified To	Position CB03-from
Unified Type	beltposition

(d)

MGSyn: Automatic game-based synthesis
MGSyn: Design-Synthesis-Codegen Environment

Game-based Synthesis within Model-driven Development

This wizard performs model concertization, synthesis, and executable code generation based on the selected choices.

Analysis: Generate synthesis model (PDDL-like)

Generate synthesis model (PDDL-like)
Synthesize strategy in high-level control format
Generate architecture script from model
Generate executable code
Finish
Cancel

(e)

Fig. 2. (a) Unidirectional conveyor belt modeled in MGSyn. (b) Behavioral interface of belt-move-b. (c) Sample specification (goal). (d) Unification of two belt positions (CB04-to and CB03-from). (e) Execution of MGSyn from within Eclipse with the list of synthesis and code generation steps.

with CB03-to while its destination overlaps with RP-a, implying Lever1 is able to push work pieces from the belt to the rotary plate.

***Synthesis, Architecture Mapping and Code Generation*.** The third step is to describe the winning condition and perform synthesis. For instance, assume that an operator (or a robot) is located at position CB03-from and feeds work pieces to the system. Due to space limits, Figure 2(c) shows only a simple specification, which is to drill and store a work piece based on its color (words after `;;` are comments). It also contains basic error handling: if the object is not facing up, it shall not be drilled and shall be returned back to the operator (position CB03-from).

To perform synthesis and code generation, the designer simply right-clicks on the EMF model to invoke MGSyn from the popup menu (Figure 2(e)). Our engine first creates a unified synthesis model by renaming overlapping positions to unique identifiers. Then the model is fed into the synthesis engine and the engine reports a winning strategy whenever possible. Synthesis for the specification in Figure 2(c) only takes seconds.

MGSyn uses a template-based approach to generate code (e.g., state variables, predicate functions, actions) from the model which is described in detail in Section 4.

4 Back-end Engine and Execution Platform Mapping

(Engine) The back-end engine first translates the EMF model to an intermediate format, which is based on PDDL [11] extended with game semantics [4]. As PDDL is very appropriate to specify behavioral interfaces, such a translation is very intuitive. However, to create a single model from multiple components, the engine needs to perform an automatic renaming over operating positions that physically overlap. Then the engine performs synthesis based on the intermediate model under the user-provided specification of the winning condition (also in a PDDL-like format). The output of the engine is a sequential program with of *Controller*'s moves, where each move is executed only if a set of conditions on the system state is true. The conditions encode the dynamic adaptation of *Controller*'s strategy to win the game in reaction to *Environment*'s moves.

The synthesis engine handles a subset of LTL properties such as GR-1 (known to be able to capture practical specifications in reactive synthesis [15]), where the complexity of game solving is polynomial in the size of the game arena, which itself is exponential in the number of Boolean variables used in system modeling. Useful optimizations for speeding up synthesis rely on analyzing the specification and identifying relevant parts of the game arena, thereby significantly reducing the number of Boolean variables in game encoding [4]. Overall, the synthesis engine of MGSyn incorporates well-known techniques for optimizing programs and bring them to assist optimizations in synthesis, such as constant propagation and cone-of-influence computations, as described in [4]. Experiments in GAVS+ confirm that the optimization techniques described in [4] often yield performance increase of at least an order of magnitude. These optimizations may also be useful as preprocessing steps for other reactive synthesis frameworks.

(Execution Platform Mapping) After the synthesis engine has generated a winning strategy for *Controller*, the strategy has to be mapped to an executable representation. This is done in two steps: First, an API matching the modeled plant modules, topology and behavior is generated from a code template written in Xpand language using EMF tooling. Second, the synthesized solution is transformed so that it calls the functions from the API. This second step is a simple text replacement that ensures that naming conventions of the C++ programming language are enforced. The code finally compiles to a console application, which covers both simulation and execution on real hardware. For the latter scenario, the API code is based on a thin manually implemented device driver layer for triggering the respective actions on the hardware.

The presented two-step approach has the advantage that the API is independent from the actually executed strategy; it may be generated once and re-used for different winning conditions as long as the plant model remains the same.

The model elements are mapped to API code as follows: Entities (e.g., *operating positions*, work pieces, colors) are mapped to enumerations and made available as data types. *Predicates* are transformed into state variables and predicate functions for retrieving and modifying the system state. For each of *Controller's moves*, a function with the following behavior is generated: If execution on real hardware should be performed, it calls the device driver library functions to trigger the respective control action(s). It then updates the state variables according to the action's effects specified in the model. For each of *Environment's moves*, a function is generated as follows: If execution on real hardware should be performed, it calls the device driver library functions to retrieve the respective sensor reading(s). If simulation should be performed, it prints a list of possible sensor readings extracted from the model and asks the operator to make a choice. Finally, it updates the state variables according to the (simulated) sensor reading(s).

Lastly, the main program is generated as follows: When the program starts, the device drivers are initialized if execution on real hardware should be performed. Moreover, the state variables are set to their initial values according to the respective specification in the model. Then, a function representing the synthesized strategy is called, which contains the synthesis result to which the following transformations have been applied: Evaluations of predicates within the conditions of an action call predicate functions or directly evaluate state variables within the API. Invocations of control actions call the functions representing the respective moves within the API.

5 Concluding Remarks

The strengths of MGSyn show when the automation task (winning condition) needs to be adapted, as only small changes in the formal model are required to generate correct-by-construction code compared to hours or days of manual modification. In practice it is also useful that MGSyn indicates infeasibility, that is, there is no winning strategy for the specified control problem.

These features set MGSyn apart from traditional programming paradigms in automation by increasing efficiency and reducing potential sources of error. So far we have adapted MGSyn to two different FESTO MPS plants with different modules, processors, and communication protocols. which were designed according to industrial standards.

Several extensions to MGSyn are planned, including the handling of real-time properties and for incorporating basic fault-tolerance mechanisms. The injection of faults, in particular, may be modeled by moves of the *Environment*, and fault-tolerance patterns may be incorporated into the synthesis engine as suggested in [6]. Currently, we are in the process of extending the FESTO MPS demonstrator with capabilities for communicating with the work pieces to be manufactured, which may then (compare "Internet of things / Industrie 4.0"), determine how they should be processed by the processing plant. In this way, work pieces become important new players in the game of industrial production control.

Acknowledgement. We thank Barbara Jobstmann for evaluating some optimization techniques in Anzu.

References

1. Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`
2. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: A tool for the synthesis and mining of petri nets. In: ACSD 2009, pp. 181–185. IEEE (2009)
3. Cheng, C.-H., Geisinger, M., Ruess, H., Buckl, C., Knoll, A.: Game solving for industrial automation and control. In: ICRA (to appear, May 2012)
4. Cheng, C.-H., Jobstmann, B., Geisinger, M., Diot-Girald, S., Knoll, A., Buckl, C., Ruess, H.: Optimizations for game-based synthesis. Technical Report 12, Verimag (2011)
5. Cheng, C.-H., Knoll, A., Luttenberger, M., Buckl, C.: GAVS+: An Open Platform for the Research of Algorithmic Game Solving. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 258–261. Springer, Heidelberg (2011)
6. Cheng, C.-H., Rueß, H., Knoll, A., Buckl, C.: Synthesis of Fault-Tolerant Embedded Systems Using Games: From Theory to Practice. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 118–133. Springer, Heidelberg (2011)
7. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on Information and Systems 80(315-325), 182 (1997)
8. Der Jeng, M., DiCesare, F.: A review of synthesis techniques for petri nets with applications to automated manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics 23(1), 301–312 (1993)
9. Dimitrova, R., Finkbeiner, B.: Synthesis of Fault-Tolerant Distributed Systems. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 321–336. Springer, Heidelberg (2009)
10. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
11. Ghallab, M., Aeronautiques, C., Isi, C., Penberthy, S., Smith, D., Sun, Y., Weld, D.: PDDL-the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control (October 1998)
12. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD 2006, pp. 117–124. IEEE (2006)
13. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A Tool for Property Synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
14. Madhusudan, P.: Synthesizing reactive programs. In: CSL 2011. LIPIcs, vol. 12, pp. 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
15. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
16. Uzam, M., Zhou, M.: An iterative synthesis approach to petri net-based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 37(3), 362–371 (2007)