Synthesizing Controllers for Automation Tasks with Performance Guarantees

Chih-Hong Cheng, Michael Geisinger, and Christian Buckl

fortiss GmbH, Guerickestr. 25, 80805 München, Germany
http://mgsyn.fortiss.org/

Abstract. We present an extension of the MGSyn toolbox that allows synthesizing parallelized controller programs for industrial automation with performance guarantees. We explain the underlying design, outline its algorithmic optimizations, and exemplify its usage with examples for controlling production systems.

1 Introduction

Game-based synthesis is a technique that automatically generates controllers implementing high-level specifications. A controller, under the game-based setting, corresponds to the finite representation of a winning strategy of a suitable game. Recent algorithmic improvements in games make synthesis possible to be applied in research domains such as programming languages, hardware design and robotics. Within the domain of industrial automation, we created the MGSyn toolbox [3] to synthesize controller programs in industrial automation that allow to orchestrate multiple processing stations. Uncertainties from sensor readings are modeled as uncontrollable (but fully specified) environment moves, thereby creating a game. The use of game-based modeling even allows the automation plant to be dependable with respect to the introduction of faults. Although the initial experiment is encouraging, the road to a solid methodology applicable to useful industrial settings is still long. One crucial requirement is to generate efficient controllers, where efficiency can be referred to several measures in production such as processing time, throughput or consumed power.

In this paper, we present an extension of MGSyn to allow synthesizing programs that not only win the corresponding game (i.e., to successfully accomplish the production tasks), but also provide explicit guarantees concerning user-specified quantitative measures. Admittedly, efforts within the research community target to synthesize optimal controllers [1, 8, 2, 4]. Nevertheless, we argue that finding optimal controllers can be difficult in practice – apart from complexity considerations, the optimality criteria are often multiple yet independent measures and no global optimum exists in general. By observing that performance criteria are often listed as secondary specifications that need to be *guaranteed*, creating engines that synthesize controllers and guarantee performance should be a reasonable alternative.

In summary, the extensions of MGSyn presented in this paper target the following aspects.

- Enable an intuitive method to select different *performance measures* in a cost-annotated model. For every type of performance measure, provide a corresponding synthesis engine.
- Allow identifying potential actions that are possible to be *executed in parallel*, as efficient execution of production tasks requires the exploitation of parallelization.
- Synthesize controllers that guarantee performance under *non-cooperative scenarios*. For many problems, a solution is only possible when the environment cooperates. For instance, the success of assembly might rely on correct assistance from humans.

Table 1. Semantics of sequential ($\odot$) and parallel ($\otimes$) composition (WC = worst case, ET = execution time).

cost ≈ ET	$\odot :=$ max	$\odot :=$ sum
$\otimes :=$ max	WCET of any single action	Total WCET
$\otimes :=$ sum	–	Total ET of all actions

cost ≈ power	$\odot :=$ max	$\odot :=$ sum
$\otimes :=$ max	Peak power consumption of any single action	–
$\otimes :=$ sum	WC peak power consumption	WC total power consumption

2 Approach

Cost annotation. For quantitative synthesis, the common model of computation is based on weighted automata [5] where costs of actions are annotated on edges. The quantitative extension of MGSyn allows specifying costs as a performance metric with the following restrictions: (1) The cost is annotated on a parameterized action as an upper bound and every concretized action (i.e., action instance with concrete parameter values) inherits that cost. (2) All costs are non-negative integers. (3) Uncontrollable actions (i.e., environment moves) have zero cost. The first restriction is due to the syntactic formats of the PDDL language [6]. The second restriction is used for symbolic encoding in *binary decision diagrams* (BDD). The last restriction allows applying a heuristic approach within the symbolic engine for pruning the search space as described later.

MGSyn allows the user to select a *sequential composition operator* ($\odot$) that defines a new value from (i) the value of the existing trace and (ii) the current cost associated with the selected edge. Two commonly seen operators are max and sum. For example, if cost annotation in the weighted automaton corresponds to power consumption, then a sequential composition based on the max operator models peak power consumption, while a sequential composition based on the sum operator models total power consumption.

Parallel execution. MGSyn assumes that two actions can in principle be executed in parallel when the operating positions affected by the actions are non-overlapping and the actions work on mutually different parameters (i.e., no "resource sharing"). Consequently, MGSyn generates syntactic guards when two parameterized control actions are composed. Consider conveyor belt action belt-move(dev, wp, pos_a, pos_b) which allows to use device dev to move a work piece wp from position pos_a to position pos_b. For parallel execution, MGSyn automatically derives action PAR_belt-move_belt-move$(dev_1, wp_1, pos_{1a}, pos_{1b}, dev_2, wp_2, pos_{2a}, pos_{2b})$ for moving two different work pieces on two different conveyor belts at the same time. In the precondition of this action, the constraints $dev_1 \neq dev_2$, $wp_1 \neq wp_2$, $pos_{1a} \neq pos_{2a}, pos_{2b}$ and $pos_{1b} \neq pos_{2a}, pos_{2b}$ are automatically added to ensure that parameter values are different[1].

To use quantitative synthesis, we also need to design parallel composition operators orthogonal to sequential composition operators. Table 1 provides some examples for cost semantics with respect to execution time and power consumption and the two operators sequential composition ($\odot$) and parallel composition ($\otimes$), where "–" indicates that no meaningful semantics was found.

The effects of parallel composition operators are statically created and are independent of the synthesis algorithm. For example, when action belt-move has cost 3, MGSyn creates parallel action PAR_belt-move_belt-move with cost 6 if $\otimes :=$ sum.

Synthesis engine. We outline how the synthesis engine supports sequential operators.

- For max, given a performance (i.e., cost) bound k, the engine statically removes every parameterized control action whose cost is greater than k. Notice that as the cost of any environment action is always zero (cf. restriction 3), we never restrict the ability of the environment. Then the game is created as if no cost is used. Therefore, max can be used in all game types.

[1] MGSyn does not generate constraints such as $dev_1 \neq pos_{2a}$, as dev_1 and pos_{2a} have different types.

- For sum, the support of quantitative synthesis is mainly within reachability games where a synthesized strategy does not contain a loop, as any loop with nonzero cost implies the overall cost to be infinite. Given a performance bound k, the synthesis engine starts with the set of goal states whose cost equals k and computes the reachability attractor. The controller wins the game if the attractor contains the initial state whose cost is above zero. Let a state be (q, α) where q is the state for the non-quantitative reachability game and α is the cost. During the attractor computation, given $\alpha_2 > \alpha_1$, then if two states (q, α_2) and (q, α_1) are both in the attractor, we can discard (q, α_1). This is because the environment has no control on the cost (cf. restriction 3). Therefore, if from q one can reach the goal state with cost $k - \alpha_2$, there is no need to consider another path having cost $k - \alpha_1$. In other words, (q, α_2) and (q, α_1) can be represented by taking only (q, α_2) without losing the determinacy of the game. This technique allows reducing the size of attractor during symbolic computation.

Non-cooperative environment. The last extension of MGSyn introduces *goal-or-loop* specifications. Such specifications are used within scenarios where the system is allowed to loop if assumptions do not hold, but should reach the goal if assumptions are met[2]. This concept can also be applied to synthesize low-level controllers realizing parameterized actions. For example, consider the action belt-move of the conveyor belt. Realizing such a controller requires a specification which checks when the work piece has arrived at the end of the belt, and the synthesized program should allow to loop as long as the work piece is not detected.

Moreover, it is in general undesirable that an automation system behaves arbitrarily (which still conforms to the specification) during the looping process, as this consumes excessive energy. This problem can be effectively handled by a game reduction that sets the cost of idle or sensor-triggering actions to be zero and all other actions to be greater than zero. By doing so, when specifying an upper bound on the total accumulated cost, the synthesized controller will ensure that the cost accumulation is zero during the looping process, since this is the only way to ensure that the accumulated cost does not exceed the threshold.

Given a looping condition $Loop$ and a goal condition $Goal$ where both are sets of states, the synthesis algorithm is based on an approach that solves reachability and safety games in sequence: first apply reachability game solving and compute the control attractor $A := \mathsf{Attr}_0(Goal)$ where states within A can eventually enter the goal regardless of choices made by the environment. Then use safety game to compute the environment attractor $B := \mathsf{Attr}_1(\neg Loop \wedge \neg A)$ where for every state $s \in B$, the environment can guarantee to reach $\neg Loop \wedge \neg A$ regardless of choices made by the controller. If a state is within A, a strategy to reach the goal exists. Otherwise, if a controllable state s is not within B, it has a strategy to stay outside $\neg Loop \wedge \neg A$, i.e., to stay within $Loop \vee A$. As s is not within A, it is within $Loop$.

Therefore, with the above computation, a feasible strategy can guarantee to loop within $Loop$, or eventually reach a state that is within A. From that state, the reachability strategy is used to guide the run towards the goal. The complexity of solving goal-or-loop winning conditions is linear to the size of the arena, making it feasible to be applied in larger scenarios. By annotating actions with cost, MGSyn allows to synthesize controllers that guarantees efficiency in looping (looping cannot increase cost).

3 Using MGSyn for Quantitative Synthesis

In the following, we demonstrate how quantitative synthesis is achieved in MGSyn in a simplified scenario. The FESTO Modular Production System (MPS)[3] is a modular system of mechatronic

[2] Although the underlying synthesis engine of MGSyn supports GR-1 specifications [7], such a specification is hardly used by us in the considered automation settings.

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

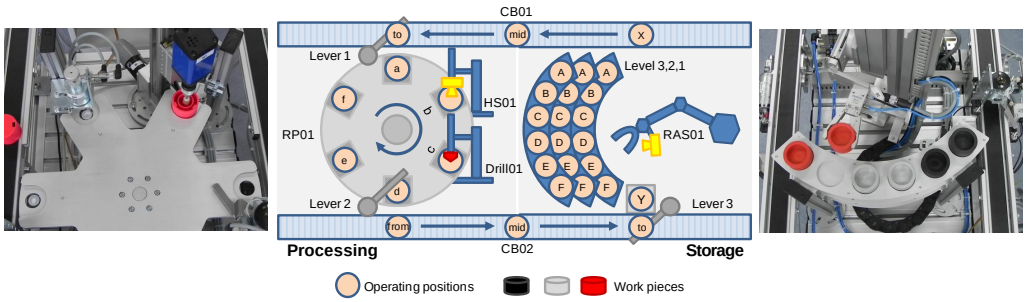

Fig. 1. FESTO MPS automation system and its simplified abstract model.

devices that model industrial automation tasks by processing simple work pieces. Our demonstration comprises two FESTO MPS units that form a circular processing chain, namely *storage* and *processing* (compare Figure 1). The formal model derived from this setup consists of:

- A list of formal predicates that describe the system state space, for example at(?work-piece ?position), drilled(?work-piece) and color(?work-piece ?value).
- A list of devices (instances of the predefined device types robot arm storage RAS, conveyor belt CB, lever Lever, rotary plate RP, height sensor HS, drill Drill) with operating positions.
- Behavioral interfaces (actions) associated with each device type (e.g., belt-move, plate-rotate, trigger-color-sensor) with annotated individual *costs*. Formally, a behavioral interface specifies preconditions and effects on the system state space.
- Quantitative properties (i.e., goal conditions over the system state space) with annotated *cost bounds* as well as sequential and parallel *composition operators*. Composition operators can be either sum or max as presented in Section 1.

We formulate a formal specification in PDDL which resembles the following informal specification: initially, work pieces wp1 and wp2 are located at CB01-mid and CB02-mid, respectively. The goal is to drill wp1 if it is facing up (which means the work piece's orientation is correct) and to move it to CB02-mid. wp2 should be stored in the storage rack level that corresponds to its color (red work pieces go to upper level and white work pieces to middle level), but when the rack is already occupied, it should be moved to CB01-mid. Costs are annotated as follows: behavioral interfaces robot-move (for RAS01) and belt-move (for CB01 and CB02) have cost 3, plate-rotate for RP-01 has cost 2 and all other behavioral interfaces (including sensor triggering) have cost 1. Furthermore, we formulate the following optimization goals:

1. *WCET time optimization:* Synthesize a strategy that does not exceed a specified maximal execution time. Cost corresponds to execution time with $\odot :=$ sum, $\otimes :=$ max.
2. *WC total power consumption optimization:* Synthesize a strategy not exceeding a given WC total power consumption. Cost represents power consumption with $\odot :=$ sum, $\otimes :=$ sum.
3. *WC peak power consumption optimization:* Synthesize a strategy not exceeding a given WC peak power consumption. Cost represents power consumption with $\odot :=$ max, $\otimes :=$ sum.

Table 2 summarizes the results. In case of feasibility, synthesis times also include C code generation for execution on real hardware or simulation. Worst case (WC) numbers of moves were directly extracted from the generated strategy. Worst case costs were derived by inspecting all possible paths in the generated strategy using simulation.

The results show that about one third of the control moves can be parallelized and that parallelization requires about three times the synthesis time of the non-parallel case for the given specification. Higher cost bounds require a slightly higher synthesis time. When the cost bound is very tight, the tool synthesizes a strategy with more, but cheaper moves (e.g., 15 instead of 14). The generated strategy for experiment 3 significantly differs from the strategy for 1 and 2.

Table 2. Results of synthesis from quantitative specifications. For comparison, results for experiments without cost model are provided. Times refer to a 3 GHz system with 4 GB of RAM (single-threaded algorithm).

Experiment	Max. degree of parallelization	Cost bound	$\odot$	$\oplus$	WC moves	WC cost	Synthesis time (sec)
1. WCET optimization	2	28	sum	max	inf.[1]	inf.[1]	18.7
	2	29	sum	max	15	29	19.4
	2	30[2]	sum	max	14	29	22.1
2. WC total power consumption optimization	2	41	sum	sum	inf.[1]	inf.[1]	20.8
	2	42	sum	sum	15	42	21.0
	2	43[2]	sum	sum	14	42	21.1
3. WC peak power consumption optimization	2	2	max	sum	inf.[1]	inf.[1]	14.9[3]
	2	3	max	sum	18	3	16.3
	2	4[2]	max	sum	15	4	19.2
Parallelization disabled	1	41	sum	N/A	inf.[1]	inf.[1]	6.5
	1	42	sum	N/A	22	42	7.1
	1	43[2]	sum	N/A	21	42	7.6
	1	2	max	N/A	inf.[1]	inf.[1]	5.4
	1	3[2]	max	N/A	21	3	6.3
Non-quantitative (no consideration of cost)	1	∞	N/A	N/A	21	N/A	6.4
	2	∞	N/A	N/A	14	N/A	19.0

[1] Infeasible (i.e., no solution) due to cost bound being too restrictive.
[2] The same strategy is generated also for higher cost bounds, only synthesis time differs.
[3] Since it is not obvious whether behavioral interfaces with cost 3 are actually used in the generated strategy, the infeasibility of this scenario can not be directly decided from the cost annotation/bound.

4 Conclusion

In this paper, we report how MGSyn is extended to synthesize controllers with performance guarantees. The key factors are (1) flexible interpretation of cost as a performance bound using sequential and parallel composition operators as well as (2) suitable integration into the symbolic synthesis engine. Experiments show that the resulting controllers are quantitatively better than controllers being synthesized without cost analysis. The extra synthesis time can be tolerated when controllers are generated offline.

References

1. R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, volume 5643 of LNCS, pages 140–156, Springer, 2009.
2. K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, volume 6174 of LNCS, pages 380–395. Springer, 2010.
3. C.-H. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll. MGSyn: automatic synthesis for industrial automation. In *CAV*, volume 7358 of LNCS, pages 658–664, Springer, 2012.
4. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for labelled Markov processes. *Theoretical Computer Science*, 318(3):323–354, 2004.
5. M. Droste, W. Kuich, and H. Vogler. *Handbook of weighted automata*. Springer-Verlag, 2009.
6. M. Ghallab, C. Aeronautiques, C. Isi, S. Penberthy, D. Smith, Y. Sun, and D. Weld. PDDL-the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control, Oct 1998.
7. N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive (1) designs. In *VMCAI*, volume 3855 of LNCS, pages 364–380. Springer-Verlag, 2006.
8. P. Černý, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV*, volume 6806 of LNCS, pages 243–259, Springer, 2011.

Appendix A: Availability of the Tool (not part of official submission)

The tool (together with a manual) is freely available at http://mgsyn.fortiss.org/.

Appendix B: Demonstration Plans (not part of official submission)

In the following, we outline our presentation plan.

1. Starting with a short introduction, we outline how MGSyn combines concepts from game-based synthesis and model-driven design and what its primary goals are (e.g., targeted application).
2. We then open a sample model and describe its features at a high level. We do not want to go into too many details at this point in time, since that will be explained later on.
3. We explain the usage of quantitative specifications (Figure 2).

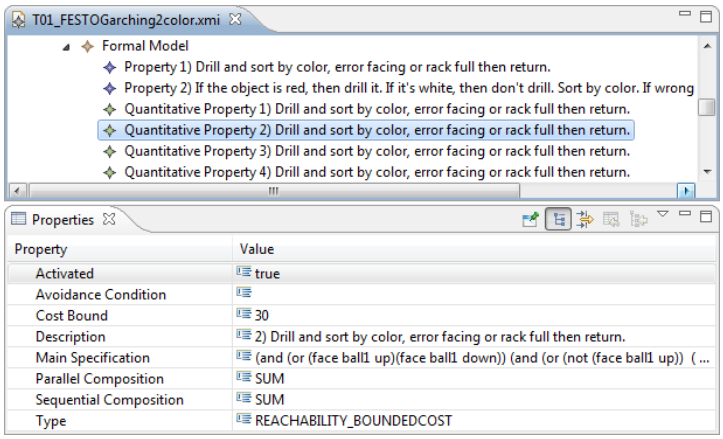

Fig. 2. FESTO MPS model and its quantitative specification.

4. We then perform automatic synthesis of the strategy with an indication of performance of the engine (Figure 3).
5. We then slightly refine the specification and perform synthesis again.
6. Finally, we use built-in functions to generate code, and execute it on concrete hardware. We will present three scenarios.
   - **(Video for FESTO MPS)** We show how the synthesized code is executed on our real FESTO Modular Production System demonstrator as a video.[4]
   - **(CIROS Simulator)** As the FESTO system is too large and is not portable, we will use the CIROS simulator to demonstrate the synthesized code running on the simulator. This enables users to examine the synthesized strategy without concrete hardware. CIROS Automation Suite (compare Figure 4) provides a 3D visualization of the system and is controlled over an OPC (OLE for Process Control) interface.
   - **(Interactive Simulator)** We will demonstrate the use of our own simulation platform, which is only text-based, but also considers the initial setup of the system and allows the operator to control all actions of the environment.

[4] Some related videos may be found at:
http://www.youtube.com/watch?v=7p5EK52TgBs
http://www.youtube.com/watch?v=Sb3bre916o4
http://www.youtube.com/watch?v=Foenmw31rB4
http://www.youtube.com/watch?v=daHLnx2IsIs

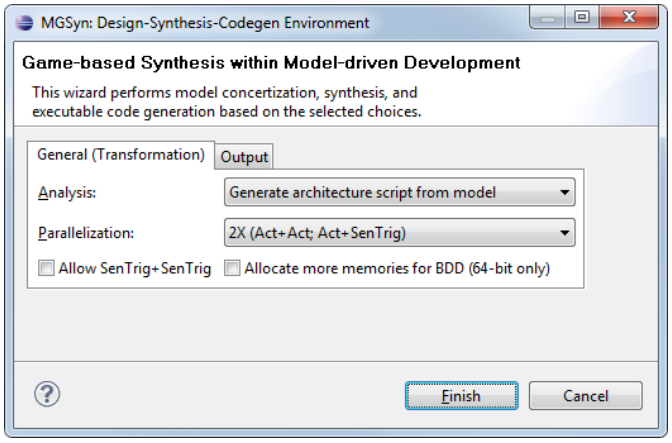

Fig. 3. MGSyn parameter setup.

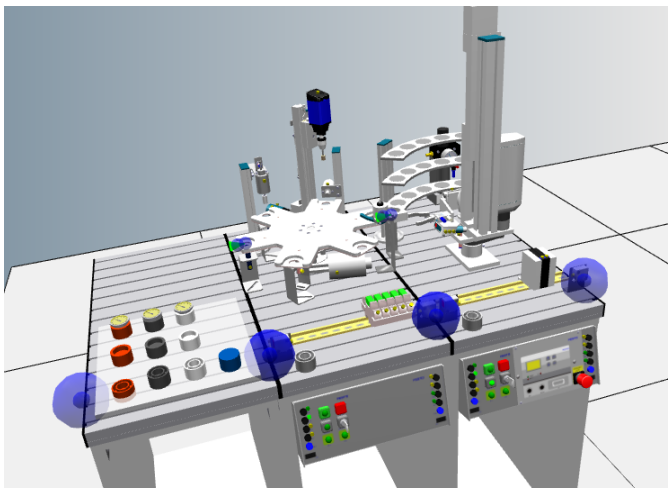

Fig. 4. CIROS Automation Suite simulation platform with a FESTO MPS model.

7. Now the audience shall have an overview of what MGSyn does and how it can be used. If time permits (depending on the time slot - whether a tool presentation is 15 or 30 minutes), we concentrate on showing the different implementation layers of the system:
   (i) PDDL-like code generation for the synthesis engine,
   (ii) invocation of the game based solver engine to generate a winning strategy,
   (iii) generation of C/C++ hardware mapping code that is only based on a minimal platform dependent code layer,
   (iv) translation of the generated strategy to C/C++ code that is based on the hardware mapping code, and
   (v) compilation of the C/C++ code and execution.
8. We will indicate how MGSyn can be extended with new hardware modules: when a new hardware module is to be added, its *operating positions*, *behavioral interfaces* and *hardware parameters* need to be specified. Also additional efforts with respect to extending the template library for code generation might be required.
9. Finally, we conclude the presentation and propose future work.