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 in the game-based setting corresponds to the finite representation of a winning strategy of a suitable game. Recent algorithmic improvements allow synthesis 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 centralized controller programs for industrial automation that 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 that allows synthesis of programs that not only win the corresponding game (i.e., successfully accomplish production tasks), but also provide explicit guarantees concerning specified quantitative measures. Admittedly, efforts within the research community target to synthesize optimal controllers [1, 7, 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. Creating engines that synthesize controllers and guarantee performance is a reasonable alternative to the typical approach of listing performance criteria as secondary specifications that need to be *guaranteed*.

The extensions of MGSyn presented in this paper target the following aspects:

- Enable an intuitive method to select *performance measures* in a cost-annotated model. For every type of performance measure, provide a corresponding synthesis engine.
- Identify sets of actions that may 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, completing the task is only possible when the environment cooperates. Our approach allows a synthesized controller to loop as long as the environment does not cooperate. The control task is achieved when the environment (e.g., a human operator) turns cooperative.

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

cost $\approx$ ET	$\odot := \mathsf{max}$	$\odot := \mathsf{sum}$
$\otimes := \mathsf{max}$	WCET of any single action	Total WCET
$\otimes := \mathsf{sum}$	–	Total ET of all actions

cost $\approx$ power	$\odot := \mathsf{max}$	$\odot := \mathsf{sum}$
$\otimes := \mathsf{max}$	Peak power consumption of any single action	–
$\otimes := \mathsf{sum}$	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) 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 format of the PDDL language [6]. The second restriction is used for symbolic encoding in *binary decision diagrams* (BDD).

MGSyn allows selecting a *sequential composition operator* $\odot$ that calculates a new value from the value of the existing trace and the current cost associated with the selected edge. Two common 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, whereas the sum operator models total power consumption.

Parallel execution. MGSyn by default generates a *sequence* of control actions that achieve the specified task. However, executing independent actions *in parallel* can be of advantage, for example by reducing the overall execution time. MGSyn assumes that two or more actions can in principle be executed in parallel when the workspaces affected by the actions are disjoint and the actions have disjoint parameters (i.e., no "resource sharing"). When parallel execution of degree d is used, MGSyn generates combinations of d actions with syntactic guards to prevent dependent actions from being executed in parallel[1]. Consider conveyor belt action belt-move$(\mathrm{dev}, \mathrm{wp}, \mathrm{p}_a, \mathrm{p}_b)$ which allows to use device dev to move a work piece wp from position p_a to position p_b. For $d = 2$, MGSyn automatically derives action PAR_belt-move_belt-move$(\mathrm{dev}_1, \mathrm{wp}_1, \mathrm{p}_{1a}, \mathrm{p}_{1b}, \mathrm{dev}_2, \mathrm{wp}_2, \mathrm{p}_{2a}, \mathrm{p}_{2b})$ for moving two different work pieces on two different conveyor belts at the same time. In the precondition of this action, the constraints $\mathrm{dev}_1 \neq \mathrm{dev}_2$, $\mathrm{wp}_1 \neq \mathrm{wp}_2$, $\mathrm{p}_{1a} \neq \mathrm{p}_{2a}, \mathrm{p}_{2b}$ and $\mathrm{p}_{1b} \neq \mathrm{p}_{2a}, \mathrm{p}_{2b}$ are automatically added to ensure that parameter values are different[2].

To use quantitative synthesis, we provide parallel composition operators orthogonal to sequential composition operators. Table 1 lists 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 in MGSyn and are independent of the synthesis algorithm. For example, if action belt-move has cost 3, MGSyn creates parallel action PAR_belt-move_belt-move with cost 6 if $\otimes := \mathsf{sum}$ and cost 3 if $\otimes := \mathsf{max}$.

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] We currently do not consider executing multiple sequential actions in parallel with another action.

[2] MGSyn does not generate constraints such as $\mathrm{dev}_1 \neq \mathrm{p}_{2a}$, because dev_1 and p_{2a} are of different types.

- For sum, the support of quantitative synthesis is mainly within reachability games where a synthesized strategy does not contain a loop, since 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. Let the state be (q, α), where q is the state of the non-quantitative reachability game and α is the cost. During the attractor computation, if (q, α) is in the attractor, one can reach the goal state from q with cost $k-\alpha$, because the environment has no control over the cost (cf. restriction 3). This allows reusing our existing game engine with reachability winning conditions. The controller wins the game if the attractor contains the initial state whose cost is greater than zero.

Non-cooperative environment. Lastly, MGSyn allows the backend solver to find strategies for *goal-or-loop* specifications. This extension focuses on specifying non-cooperative scenarios as a looping invariant. Whenever a run of the game leaves the invariant, the goal (i.e. the accomplishment of the task) should eventually be reached. 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 appeared at the start of the belt, and the synthesized program should allow to loop as long as the work piece is not detected.

It is undesirable that an automation system behaves arbitrarily during the looping process (although still conforming to the specification), because this would consume excessive energy. This problem can be handled by a game reduction that sets the cost of idle or sensor-triggering actions to be zero and all other actions greater than zero. When specifying an upper bound on the total accumulated cost, the synthesis engine will ensure that the cost accumulation is zero during the looping process, because 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 $A := \mathsf{Attr}_0(\mathit{Goal})$ 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 $B := \mathsf{Attr}_1(\neg \mathit{Loop} \wedge \neg A)$ where the environment can guarantee to reach $\neg \mathit{Loop} \wedge \neg A$ for every state $s \in B$ 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 $\neg \mathit{Loop} \wedge \neg A$, i.e., to stay within $\mathit{Loop} \vee A$. 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 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 specifications 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 (i.e., 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 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:

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

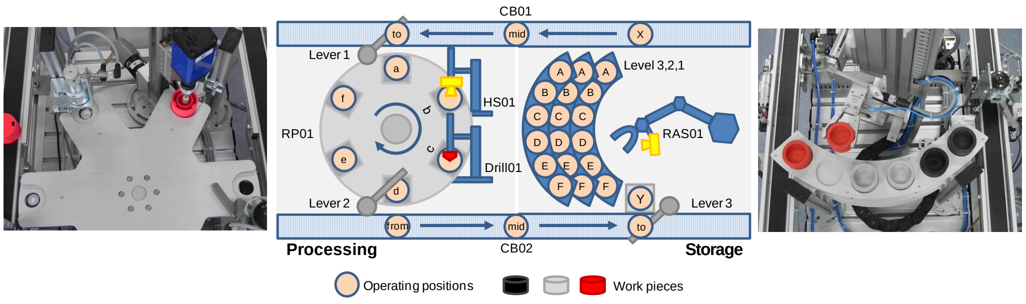

Fig. 1. FESTO MPS automation system and its simplified abstract model.

- 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 silver 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 triggerings) have cost 1. Furthermore, we formulate the following optimization goals:

1. *WCET 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 d**	**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 cannot 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, A. Howe, C. Krobnock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins. PDDL-the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control, Oct 1998.
7. 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.