2012 IEEE International Conference on Robotics and Automation
RiverCentre, Saint Paul, Minnesota, USA
May 14-18, 2012

Game Solving for Industrial Automation and Control

Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll

***Abstract*— An ongoing effort within the community of verification and program analysis is to raise the level of abstraction in programming by automatic synthesis. In this paper, we demonstrate how our synthesis engine GAVS+ achieves this goal by automatically creating control code for the FESTO Modular Production System. The overall approach is model-driven: we reinterpret planning domain definition language (PDDL) as a design contract to model two-player games played between control and environment, such that users can describe (i) basic abilities of hardware components, including sensors (as environment moves) and actuators (as control moves), (ii) topologies how components are interconnected, and (iii) desired specification under a restricted class of linear temporal logic. The model is processed by our game-based synthesis engine, from which intermediate code is generated. By mapping each behavioral-level action to a sequence of low-level PLC control commands, we transform the intermediate code into an executable program. The efficiency of our engine enables to synthesize every scenario presented in this paper within seconds. When the specification evolves, this implies a huge time-gain compared to manual program modification.**

I. Introduction

Engineering software controller systems contains two important subtasks, namely (a) creating the specification and (b) implementing control programs which satisfy the specification. Developers need to partition their labor-hours over design and implementation. Due to time constraints, two problems arise: (i) An implementation does not satisfy the specification. (ii) The specification might change over time due to modeling errors or feature enhancements. Once the implemented program is (almost) created but a slight change appears in the specification, tremendous re-engineering efforts might be required. For both two cases, the fixing process requires several loops of verification and code modification, which can be ad-hoc, tedious and error-prone. For safety critical systems, the invested efforts might even grow higher.

To alleviate these problems, in this paper we address how game-based program synthesis can be seamlessly integrated into the design flow for control automation, increasing productivity of developers. Precisely, given a concrete modular automation system, together with a logic description of desired behavior, we demonstrate a workflow to automatically generate corresponding control software. Program synthesis is a technique which makes the task of programming implicit: a user first provides the high-level specification defining the desired feature, and then the synthesis engine automatically creates a program that fulfills the specification (correctness by construction). Using such technique users can concentrate on making the specification right and leave the finding/creation of implementations to the engine. Apart from pioneering work in verification [7] or in control theory [14], recent examples for synthesis (to name a few) include sketching [15], synthesizing linear temporal logic (LTL) specifications [11], [8], [13], or partial synthesis for deadlock avoidance in component-based systems [4].

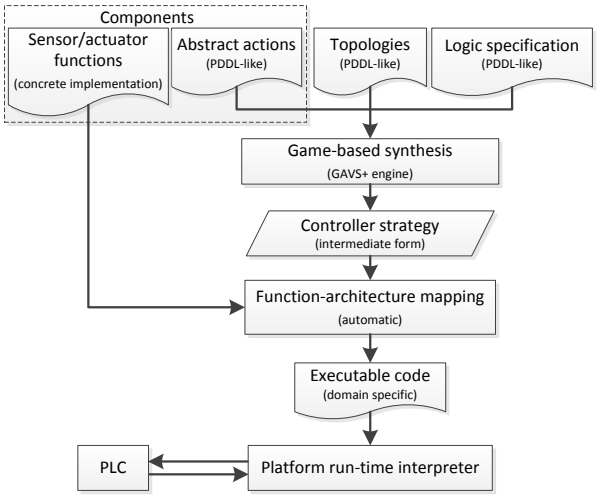

Fig. 1. The framework for game-based synthesis for industrial automation.

To perform synthesis in automation systems, it is natural to use game-theoretic approaches. In such a game, player Control has the ability to perform actuation and to trigger sensors while player Environment represents uncertainty in the system, e.g., select values returned from sensors. The system specification can be seen as conditions for Control to correctly react over all possible sensor readings offered by Environment. Then synthesizing a winning strategy for Control amounts to automatically creating control software.

Toward this goal, we specifically address the combination of (i) game-based synthesis and (ii) model-based design, as we found existing work in game-based LTL synthesis makes it difficult to specify (i) basic abilities of each component and (ii) the topology how components are interconnected. Figure 1 outlines our framework: it resolves limitations in previous work by using PDDL [9] in a game-theoretic setting, enabling to model behaviors of sensors, actuators and topologies under an appropriate level of abstraction. We use PDDL as it is known in planning for robotics. It is used as a contract between the implementation of a component and its envisioned environment (and thus could be replaced by another language). These interface descriptions of the components with the high-level specification of the overall

C.-H. Cheng, M. Geisinger, H. Ruess and C. Buckl are with fortiss GmbH, Germany. A. Knoll is with Department of Informatics, TU München, Germany. First two authors contributed equally to this work. cheng,geisinger,buckl,ruess@fortiss.org, knoll@in.tum.de

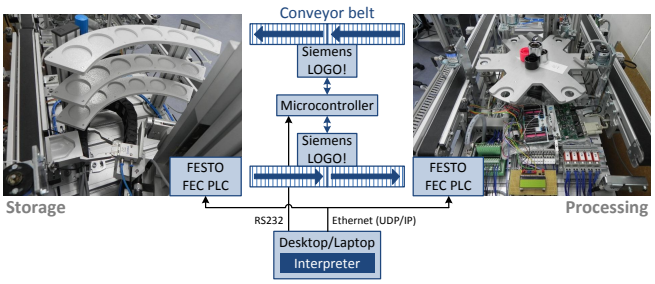

Fig. 2. A demonstrator setup of the FESTO Modular Production System.

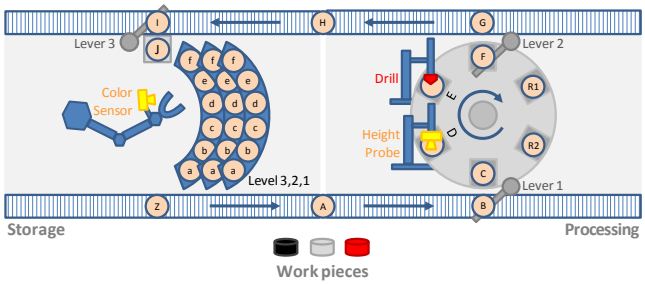

Fig. 3. The FESTO MPS demonstrator setup in abstract form.

system are required by our synthesis engine GAVS+ [6]. The result of synthesis is a high-level control with conditions for actions to execute. By mapping each behavioral-level action to a sequence of low-level PLC control commands, we transform the state machine to an executable program. The coordination between the strategy and actual running hardware requires architectural support in form of a run-time interpreter.

We give motivating scenario in Section II, followed by fixing notations in Section III. The paper continues with the workflow of automatic synthesis. Section IV describes our efforts to raise the level of abstraction for synthesis by extracting features from components. Section V outlines game solving techniques implemented in our game-based synthesis library GAVS+. Section VI describes architectural supports to bridge the result of synthesis to actual implementation. Lastly, we report our evaluation and outline further directions from Section VII to IX.

II. Scenario: Synthesis for FESTO Modular Production Systems

In this section, we describe our demonstrator from FESTO MPS (Modular Production System) for control automation. Each unit of the MPS processes small colored work pieces that are made out of plastic or metal. Our demonstrator setup (see Figure 2 for illustration) is composed of two modules, a processing unit and a storage unit.

1) The processing unit is built up from a rotating plate with six locations, a height probe sensor (which tests the shape of work pieces) and a drilling module (which processes the work pieces).
2) The storage unit contains a robot arm that is used to store and retrieve work pieces to/from a rack with six storage locations.
3) Both units are connected over two unidirectional conveyor belts that can deliver one work piece at a time.
4) Several levers are located at certain positions to move work pieces between the belt and the units.

(Basic specification) During the operation, a user can place a work piece at the center of the lower conveyor belt (see Figure 2). Our preliminary setting assumes that no work piece is initially stored on the rack, and the specification is to drill a work piece and store it on the rack based on its color. We then add the following error handling attributes to the specification.

(Error handling 1: height detection) As the work piece is provided by an operator, it is possible that the operator wrongly places an inappropriate work piece, resulting in breaking the drill head. Therefore, the new specification is to extend from the basic scenario with the requirement that a work piece shall only be drilled when it is of appropriate height. When violated, it should be moved back to the initial position for the operator to correct.

(Error handling 2: occupancy check) It can be observed that our preliminary setting does not consider the scenario where (a) some positions are initially occupied with work pieces and (b) the rack might not be empty. Therefore, the new specification is to extend from basic error handling by adding the following specification:

1) If the storage rack is full, then return the work piece back to the operator.
2) Place a work piece to a location only when the location is not occupied by another work piece.

(Problem statement) The problem under investigation is to automatically synthesize executable control code for the FESTO MPS demonstrator based on the above specifications. The synthesis process shall reduce re-engineering efforts subject to the change of specifications (e.g., from basic specification to advanced one).

III. Preliminaries

We fix the notation for games required in this paper; for complete theoretical background we refer readers to [10]. An *arena* is a directed graph $G = (V_0 \uplus V_1, E)$ where V_0 and V_1 are set of locations forming a partition over the set of all locations. In this paper, we consider two-player, turn-based games in the following and call them player 0 (Control) and player 1 (Environment). A *play* starting from node v_0 is a maximal path $\pi = v_0 v_1 \ldots$ in G where we assume that player i determines the *move* $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$ $(i \in \{0,1\})$. A *winning condition* defines when player 0 wins the play; if π is not won by player 0, it is won by player 1. A node v is won by player i if player i can always choose his moves in such a way that he wins any resulting play starting from v.

IV. Extracting Features and Game Creation

We explain how a game is created under the model depicted in Figure 3. The figure describes the resulting abstraction for defined components and topology.

(Step 1: Discretization) We first perform discretization such that we collect a set of positions of interest. For example, in Figure 3, we specify six belt positions $\{$A,B,G,H,I,Z$\}$.
(Step 2: Contracts for actions) Based on the discretization, we define basic abilities of each component by abstract actions. We use PDDL [9] for specifying the contract between the interface for individual components (used by Control) and environment moves (used by Environment). For example, the movement of a conveyor belt between two positions is modeled by a parameterized control action `belt-move` with preconditions and effects as shown below.

```
(:action belt-move
  :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))
)
```

The control action `belt-move` specifies that when two positions are connected (`belt-connected ?from ?to`), then a work piece can be moved to from position `?from` to position `?to`. Notice that one special predicate `P0TRAN` appears both in the precondition and the effect, implying that this is a move for Control, and the subsequent move belongs to Control as well. We use the valuation of `P0TRAN` to partition a game arena between Control and Environment: when `P0TRAN` evaluates to `true`, player 0 (Control) determines the move; otherwise player 1 (Environment) determines the move. E.g., the following action `return-color-value` is used by the environment to offer the color of the work piece. It is only available when the color sensor attached on the gripper is active (`color-sensor-on ?gri`) and will return the color of the work piece by setting the `color` predicate.

```
(:action return-color-value
  :parameters  (?obj - unit ?pos - position
                ?color - colortype ?gri - gripper)
  :precondition (and (not (P0TRAN)) (color-sensor-on ?gri)
                     (in-robot ?pos) (at ?obj ?pos))
  :effect (and (P0TRAN) (not (color-sensor-on ?gri))
               (color ?obj ?color))
)
```

(Step 3: Guarantees for concretization) The third step is to provide corresponding mappings from each abstract action to concrete control functions running on concrete hardware. For instance, performing `belt-move` between positions requires a collaboration of underlying sensors and actuators for the conveyor belt to achieve precise positioning.
(Step 4: Topologies and game creation) The last step is to build up topologies, create initial conditions, and construct specifications. For the abstract system in Figure 3, the constructed topology and initial condition (robot is at position `Z` without holding a work piece, the ball is provided at position `A`, the first move is done by Control) are specified in Figure 4.

V. Game Solving

We summarize game solving techniques used in synthesis. In algorithmic synthesis, efficient algorithms relies on the computation of *attractor*. For $i \in \{0,1\}$ and $S \subseteq V$, we define

```
1   (at-lever B C) (at-lever F G) (at-lever I J)
2   (belt-connected Z A) (belt-connected A B)
3   (belt-connected G H) (belt-connected H I)
4   (next C D) (next D E) (next E F)
5   (next F R1) (next R1 R2) (next R2 C)
6   (at-height-probe D) (at-drill E)
7   (in-robot Z) (at Ball A) (free-hand robot) (P0TRAN)
```

Fig. 4. The topology of the system (line 1 to 6) and the initial condition (line 7).

the map $\mathsf{attr}_i(S)$ as follows.

$$\mathsf{attr}_i(S) := S \cup \{v \in V_i \mid vE \cap S \neq \emptyset\} \cup \{v \in V_{1-i} \mid \emptyset \neq vE \subseteq S\},$$

i.e., $\mathsf{attr}_i(X)$ extends X by all those nodes from which either player i can move to X within one step or player $1-i$ cannot prevent to move within the next step (where vE denotes the set of successors of v). Lastly, define attractor $\mathsf{Attr}_i(X) := \bigcup_{k \in \mathbf{N}} \mathsf{attr}_i^k(X)$ to be the set of all locations where player i can force any play to visit the set X. The definition of attractor implies a fixpoint algorithm which can be solved in time linear to the size of the arena. It is commonly used in algorithmic synthesis: the following winning conditions (implemented in our solver library GAVS+) all utilize attractor computation.

- **(Reachability)** Given a set V_{goal} of goal locations, we can compute whether a strategy for Control exists by checking whether the initial location is in $\mathsf{Attr}_0(V_{goal})$.
- **(Safety/Reachability+Safety)** Given a set V_{risk} of risk configurations, we can compute whether it is unavoidable for Control to enter the risk by checking whether the initial location is contained in $\mathsf{Attr}_1(V_{risk})$. To examine there exists a strategy which reaches the goal while never touching any risk locations, perform first safety game solving followed by reachability game solving.
- **(Büchi or Generalized Reactivity-1 conditions)** Algorithms for computing Büchi (repeated reachability) or Generalized Reactivity-1 (GR-1: a subset of LTL enabling users to specify assumptions and corresponding guarantees) [13] all use nested attractor computation.

In our engine, we combine the following techniques to assist game solving on complicated problems.

- We develop optimization techniques as preprocessing steps. In theory, such optimization corresponds to *local game solving* techniques, i.e., the engine solves the game by exploring only a subarena of interest. Using optimization we can increase existing game-based LTL synthesis tools (e.g., Anzu [12]) by more than 40 times on selected examples. Our optimization also enables us to compete with existing sequential planners in the planning competition. Details can be found in our technical report [5].
- We use symbolic encoding to create compact representation and perform game solving symbolically. Engines for specifications above are implemented using JDD [1], a Java library for symbolic manipulation of ordered binary decision diagrams (OBDD) [3].

(FESTO MPS demonstrator) When designing the specification, we use reachability conditions. We can also use

Büchi conditions to describe repeated behaviors. However, our reachability game engine has an additional feature which can create a *witness tree* from the computed strategy. A witness tree is a representation of strategies which can be interpreted by forward reasoning from initial configuration. This makes our resulting strategy more understandable, compared to strategies collected from backward computation using attractor[1].

Consider the basic problem specified in Section II, which is to drill a work piece and store it on the rack based on the color. Initially the color value is unknown, so in our goal specification (reachability), we add an additional constraint specifying that a work piece shall be of color either in red, white or black. The set of goal states can be represented as follows (we use $\neg A \vee B$ to represent $A \rightarrow B$).

```
(and
  (drilled ball)
  (and (or (not (color ball red))   (at ball l1_a))
       (or (not (color ball white)) (at ball l2_a))
       (or (not (color ball black)) (at ball l3_a))
  )
  (or(color ball red)(color ball white)(color ball black))
)
```

An excerpt from the created intermediate code synthesized by our game solver engine is shown in Figure 5.

- In block 0, as conditions for executing `belt-move` and `robot-move` are the same, it implies that the controller can perform one of two actions. Indeed, as the robot is initially in position `Z`, the controller can first move the robot to position `J` to wait for the incoming work piece, or first trigger the belt to transfer the work piece (and later move the robot to position `J`).
- When executing sequentially until block 14, the color of the work piece has been recognized (due to the previous triggering of the color sensor), and the robot is positioned at different places based on the color of the work piece. E.g., when the ball is detected to be black, then the robot is at position `l3_a`. The ball is not in any position (`at_ball=ALL_FALSE`), as it is grasped by the robot arm. Our optimization technique is enabled to reason and conclude that the work piece can only appear at most at one position.

VI. Architectural Support for Game-based Synthesis

The following stage in the workflow of Figure 1 discusses required process how the synthesized code is transformed and executed on actual hardware.

Strategies created by our synthesis engine are monolithic, meaning that an implementation is a centralized control which coordinates multiple processing units. To achieve coordination, a run-time architecture shall be provided as an interpreter which (a) collects information from separate units, (b) interfaces with the controller running the synthesized strategy, and (c) delivers actions from control to the units.

[1]From the definition of $\mathsf{Attr}_i(X)$, we can observe that in synthesis, the computation is performed backwards. This is contrarily to game solving such as chess, where forward computation is used.

```
=======================
/* Start of block 0: */
IF (in-robot=Z && at-ball=A && free-hand(robot) && ... ) {
    belt-move({?obj=ball, ?from=A, ?to=B})
}
IF ( in-robot=Z && at-ball=A && free-hand(robot) && ... ) {
    robot-move({?from=Z, ?to=J})
}
/* End of block 0: */
=======================
...
====================
/* Start of block 14: */
IF (
( in-robot=l3_a && at-ball=ALL_FALSE &&
  !free-hand(robot) && color_ball=black && ... )
) {
    robot-drop({?gripper=robot, ?obj=ball, ?room=l3_a})
}
IF (
( in-robot=l2_a && at-ball=ALL_FALSE &&
  !free-hand(robot) && color_ball=white && ... )
) {
    robot-drop({?gripper=robot, ?obj=ball, ?room=l2_a})
}
IF (
( in-robot=l1_a && at-ball=ALL_FALSE &&
  !free-hand(robot) && color_ball=red && ... )
) {
    robot-drop({?gripper=robot, ?obj=ball, ?room=l1_a})
}
/* End of block 14: */
====================
```

Fig. 5. The intermediate code (some details omitted for clarity) synthesized from GAVS+ for the basic specification.

(Generating executable code) FESTO MPS units are usually controlled by PLCs that are directly attached to the units. However, a global control is needed to execute the generated strategy. We thus write an interpreter application, further called control program, that executes the synthesized strategy from Figure 5. For this purpose, the generated intermediate code is automatically translated into C code by (1) renaming all operators, variables, actions and predicates to match the syntax of C and (2) ordering the parameters to the function calls to match their original definition. The result is illustrated at the top of Figure 6.

(Platform run-time interpreter) The function that are called from the executable code (e.g., `belt_move()`, `robot_move()`) are part of the platform run-time interpreter as introduced in Figure 1. The currently hand-written implementation of `belt_move()` is shown in the lower part of Figure 6[2]. Ideally, the manufacturer of the automation system components would provide this code. The mapping of function calls to actions in the automation system needs to be implemented manually, however due to its modular nature, the implementation can be reused across different setups. We implement the platform run-time interpreter in C++ and use the language's object oriented concepts to build a class hierarchy of common functionality patterns. For example, we introduced special classes for executing actions (`Command`) as well as retrieving sensor values (`SensorCommand`).

(Communication) We also provide a C++-based abstraction library for the communication between the platform support

[2]Here for the ease of understanding, in the function we enumerate all control decisions by case split. In general, this function requires to reference another table which contains the mapping between the topology in the model and the concrete hardware.

```
/* Start of block 0: */
if (
( in_robot == Z && at_ball == A &&
  free_hand(robot) && ... )
) {
    belt_move(Ball, A, B);
}
...

void belt_move(unit_t obj, position_t from, position_t to)
{
  // Assert preconditions
  assert(belt_connected(from, to) && at(obj, from) &&
    !occupied(to));

  // Actions
  if (A == from && B == to) {
    SupplyWorkPieceCommand::exec(LP_DOMAIN_PROCESSING);
  } else if (G == from && H == to) {
    DeliverWorkPieceCommand::exec(LP_DOMAIN_PROCESSING);
  } else if (H == from && I == to) {
    SupplyWorkPieceCommand::exec(LP_DOMAIN_STORAGE);
  } else if (Z == from && A == to) {
    DeliverWorkPieceCommand::exec(LP_DOMAIN_STORAGE);
  }

  // Required since there is no status flag available
  DelayCommand::exec(8000);

  // Update variable and assert effects
  v_occupied.erase(from);
  v_occupied[to] = 1;
  v_at[obj] = to; // Object can only be at one location

  assert(!occupied(from) && !at(obj, from) &&
    at(obj, to) && occupied(to));
}
```

Fig. 6. An excerpt from the final C code (some details omitted for clarity) generated from the intermediate code as well as the hand-written platform run-time interpreter code for `belt_move()`.

library and a PLC. It currently supports communication over Ethernet (UDP/IP and TCP/IP), serial communication (RS232, RS422, RS485) as well as an implementation of the industrial communication protocol Modbus. The library is cross-platform. Due to its object oriented nature, the library can be easily extended.

In our scenario, the control program communicates with the FESTO FEC PLCs over Ethernet (UDP/IP). We modify the PLC programs so that every atomic action of the strategy can be triggered individually. When an action is triggered, the control program waits until execution has finished by polling status flags in the PLC. Since the conveyor belts are controlled by separate Siemens LOGO! controllers that do not have an appropriate communication interface, we furthermore add a microcontroller that translates incoming requests from the control program over serial bus (RS232) to simple commands sent to the conveyor belt controllers over their digital input ports and modified the conveyor belt control programs accordingly.

We want to stress that most of the required workarounds were of artificial nature. In a real automation system, PLCs are usually more powerful and connected to a common fieldbus, simplifying the task of controlling them from a global control program. We are currently working on using more powerful PLCs and OPC (OLE for Process Control) [2] to remove the workarounds and make the system compatible with many existing automation systems.

VII. Evaluation: "How Can Synthesis Help"

In this section, we report how can game-based synthesis assist controller design based on our evaluation for the FESTO MPS demonstrator.

A. Fast speed of synthesis

Given the desired specification (including the map of the control function) mentioned in Section II, the corresponding software is synthesized less than 5 seconds[3][4]. We have also constructed a larger system by extending the configuration in Section II with four additional FESTO MPS modules[5]. Our preliminary evaluation on such model shows that the synthesis engine is still able to synthesize strategy (create the intermediate code) within reasonable amount of time (depending on the complexity of the specification).

When the specification changes (while the system setup remains the same), the new software can be created automatically (unless there exists change of the mapping from abstract functions to concrete control commands). For instance, basic error handling requires to refine the specification in Section V by distinguishing the height.

```
(and
  (or (height Ball small) (height Ball large))
  (and
    (or (not (height Ball small))
      (and (drilled Ball)
        (and (or (not (color Ball red))   (at Ball L1_a))
             (or (not (color Ball white)) (at Ball L2_a))
             (or (not (color Ball black)) (at Ball L3_a))
        )
        (or (color Ball red)(color Ball white)
            (color Ball black))
      )
    )
  (or (not (height Ball large))
    (and (not (drilled Ball)) (at Ball P1) )
  )
)
```

This brings a huge efficiency gain compared to manual modification: in an initial experiment we invited a student to (i) understand the code for the basic scenario and (ii) extend the functionality to include error handling capabilities. Compared to several seconds used by the automatic synthesis engine, the student needs long work-hours to create an implementation by modifying existing code without explicit guarantee for correctness.

B. Concentrated focus on specification: experience report

With automatic synthesis, users can concentrate on the high-level specification and the modeling. This paragraph describes some initial experience gained during our evaluation.

Initially we first synthesized successfully the control software for the basic scenario (store work piece based on color). When switching to the scenario of error handling, we found that the system under control does not behave as expected: the work piece does not return to the user successfully. We realized that a problem occurred in position `Z`, which is a

[3]Evaluated under a 2.8 Ghz machine with 2 GB of memory.

[4]Video for actual execution of storing by color, height detection and occupancy check available at: http://www.fortiss.org/formal-methods

[5]These expanded modules are no. 195780 (Distributing), no. 195781 (Testing), no. 535246 (Pick&Place) and no. 195761 (Sorting).

position where (i) the belt can transfer a work piece and (ii) the robot arm can drop a work piece. When robot-arm drops a work piece, it opens its gripper. Nevertheless, triggering belt transmission does not move the work piece without first moving away the robot-arm: the belt is moving but the work piece remains blocked by the gripper. We fix the condition on the model to specify that the belt transmission from `Z` to `A` is only valid when the arm is not at `Z`. The change does not involve any change in the concrete mapping. The solver then synthesizes correct control software which moves the arm from `Z` first to a different position before moving the belt. The solver automatically assigns a new position for the robot that fulfills the added condition. Without worrying how to fix the code, the total time from model fixing to code generation is within 15 minutes.

VIII. Game-based Synthesis for Control Action Parallelization

Before we conclude the paper, we discuss how to extend our framework and synthesize parallelized control actions to manipulate multiple workpieces based on action independence. Denote two different control actions `action1`$(a_1,\ldots,a_n)$ and `action2`$(b_1,...,b_m)$ as *independent* if the following conditions hold:

- No parameter appears both in set $\{a_1,\ldots,a_n\}$ and in set $\{b_1,\ldots,b_m\}$.
- Two actions are not invoked by the same hardware.
- In each action, at most one workpiece may appear in the workspace[6].

For instance, for the synthesized strategy of FESTO demonstrator in Figure 5, actions `belt-move(?obj=ball, ?from=A,?to=B)` and `robot-move(?from=Z,?to=J)` are independent, meaning that they can be executed in parallel (see Figure 3 for illustration). Contrarily, action `activate-lever(?obj=ball,?from=F,?to=G)`, which uses `Lever 2` to push an item from `F` to `G` is not independent from action `belt-move(?obj=ball2, ?from=G,?to=H)`, which uses the upper conveyor belt to transfer a work piece from `G` to `H`: although two actions are executed on different hardware, the overlapping of position `G` is reflected on the parameter. Given the degree i of parallelization offered by the user, we can statically collect the independence relation in our synthesis engine. Then we create new symbolic transitions from the product of n independent actions, where $0 < n < i$ as follows:

1) We restrict that in the new symbolic transition for control, we only trigger at most one sensor (i.e., at most one action has its `:effect` field change the value of `P0TRAN` to `false`.
2) Given n independent control actions, create the symbolic transition. During the creation of each action, first omit constructing `P0TRAN`. Lastly, if the set of independent control actions contains an action which triggers a sensor, then the combined control action updates `P0TRAN` from `true` to `false`. Otherwise, `P0TRAN` remains the same.
3) We do not need to combine multiple environment actions, because of the restriction in 1).

[6]For an action which can manipulate simultaneously multiple objects in its workspace (e.g., the rotary plate in Fig. 2), the parallelization is explicitly in the model. Therefore, we manually refine such an action by considering the joint effect of multiple workpieces, and conservatively disallow it to be combined with other actions.

IX. Conclusion

We have demonstrated a complete flow how game-based synthesis can be used to generate executable code in control automation. Given the desired specification (including the map of the control function), the corresponding software is synthesized within seconds. When the specification changes, automatic synthesis enables to quickly generate correct software. It is significantly faster compared to hours of manual modification, which can also be ad-hoc and error-prone. The technique can be extended to synthesize extra-functional behaviors such as fault-tolerant controllers; the game-theoretic concept makes it applicable to describe fault models.

As stated in Section VI, the architectural support still needs to be improved. We will integrate an OPC client into our communication library for the control program to be able to apply apply our approach to many real-world scenarios. We are also planning to automatically synthesize at least part of the currently manually written platform run-time interpreter code from annotations in the domain model. This makes adaptations of the platform mapping more local and reduces manual implementation.

References

[1] JDD project. `http://javaddlib.sourceforge.net/jdd/`.
[2] The OPC foundation. `http://www.opcfoundation.org/`.
[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
[4] C.-H. Cheng, S. Bensalem, Y.-F. Chen, R.-J. Yan, B. Jobstmann, A. Knoll, C. Buckl, and H. Ruess. Algorithms for synthesizing priorities in component-based systems. In *ATVA'11*, LNCS. Springer-Verlag, 2011.
[5] C.-H. Cheng, B. Jobstmann, M. Geisinger, S. Diot-Girald, A. Knoll, C. Buckl, and H. Ruess. Optimizations for game-based synthesis. Technical Report 12, Verimag, 2011.
[6] C.-H. Cheng, A. Knoll, M. Luttenberger, and C. Buckl. GAVS+: An open framework for the research of algorithmic game solving. In *TACAS'11*, volume 6605 of *LNCS*, pages 258–261. Springer, 2011.
[7] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, 131:52–71, 1982.
[8] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *TACAS'11*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.
[9] 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 1998.
[10] E. Gradel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*. Springer, 2002.
[11] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD'06*, pages 117–124. IEEE, 2006.
[12] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV'07*, volume 4590 of *LNCS*, pages 258–262. Springer, 2007.
[13] N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive (1) designs. In *VMCAI'06*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.
[14] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
[15] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI'05*, pages 281–294. ACM, 2005.