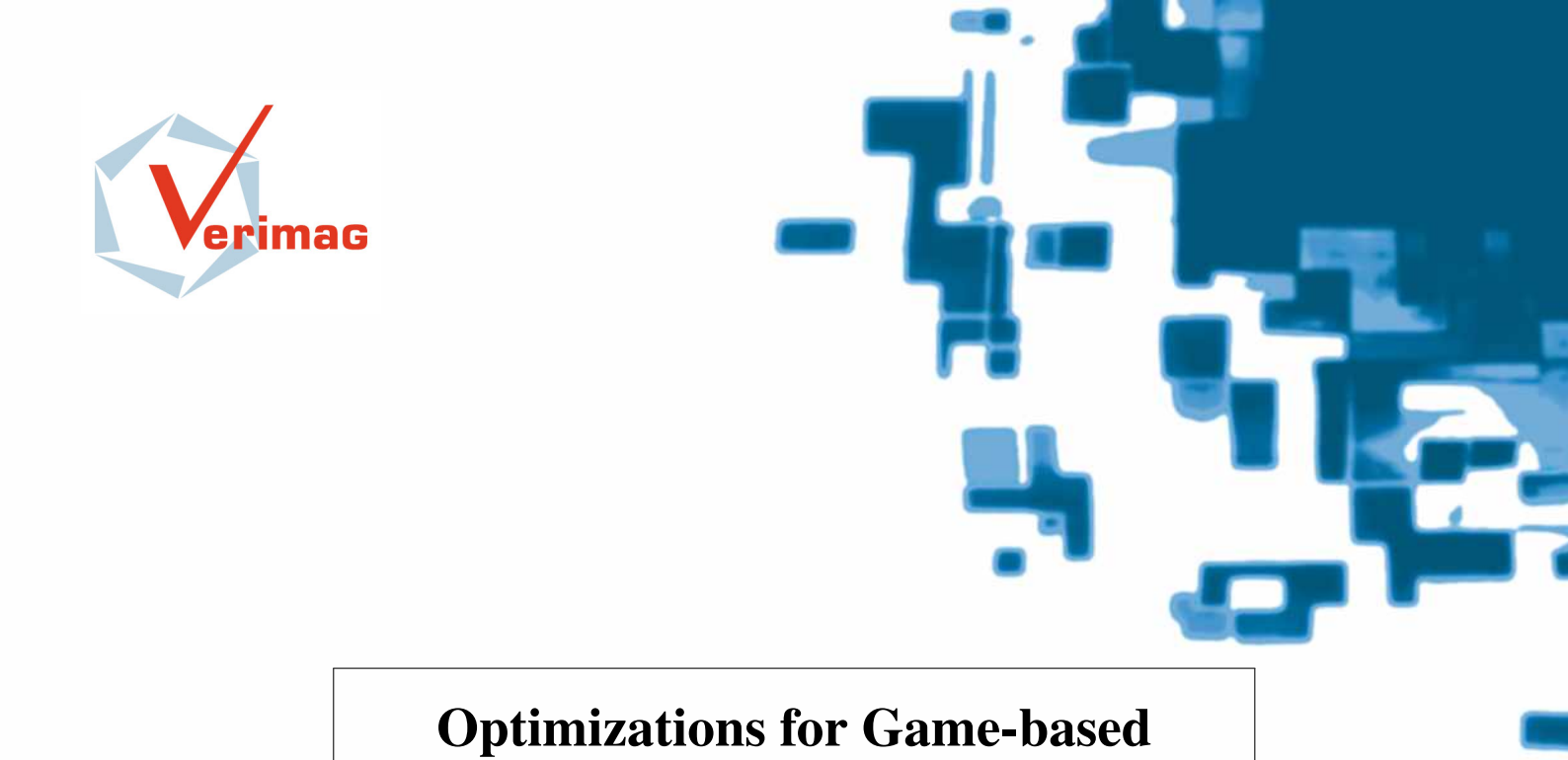

Optimizations for Game-based Software Synthesis

Chih-Hong Cheng[1,2], *Barbara Jobstmann*[3], *Michael Geisinger*[2], *Sarah Diot-Girard*[1], *Christian Buckl*[2], *Alois Knoll*[1], *and Harald Ruess*[2]

Verimag Research Report n^o TR-2011-12

August 22, 2011

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Optimizations for Game-based Software Synthesis

Chih-Hong Cheng[1,2]*, Barbara Jobstmann*[3]*,*
Michael Geisinger[2]*, Sarah Diot-Girard*[1]*,*
Christian Buckl[2]*, Alois Knoll*[1]*, and Harald Ruess*[2]

[1] Technischen Universität München, Germany
[2] fortiss GmbH, Germany
[3] CNRS/Verimag, France

August 22, 2011

Abstract

In this paper, we study how to make software controller synthesis more efficient and easy to use for engineers. In order to simplify specifying a system, we believe that existing methodologies in synthesis of reactive systems from Linear Temporal Logic (LTL) need to be extended with parameterized propositions. Therefore, we proposed to use the behavioral description language PDDL and reinterpret it under game concepts. Although the use of such high-level language eases modeling and results in understandable code, the size of the created game can be excessively large. Our main contribution is to view and adapt program optimization techniques as the key weapon for solving games locally, obtaining drastic performance gain. Results are implemented as an extension to the GAVS+ tool. In our evaluation, we have synthesized a control strategy for a demonstrator of FESTO's modular production system and translated the strategy into executable control code running on the production system.

Keywords: Games, Software Synthesis, Optimization

Reviewers:

How to cite this report:

@techreport {TR-2011-12,
title = {Optimizations for Game-based Software Synthesis},
author = {Chih-Hong Cheng, Barbara Jobstmann, Michael Geisinger, Sarah Diot-Girard, Christian Buckl, Alois Knoll, Harald Ruess},
institution = {{Verimag} Research Report},
number = {TR-2011-12},
year = {}
}

1 Introduction

Our work is motivated by the goal to provide end-users with a controller synthesis framework that is easy-to-use and reasonably efficient. During our tool construction, we have encountered the following challenges.

Challenge A: Ease of modeling. The first challenge is to provide the engineers with an adequate way to describe a system setup, i.e., from a user's perspective, the system should be describable under an appropriate level of abstraction. E.g, a user may define basic abilities of each component as atomic actions separately (for component reuse). Then, based on individual problems, the user specifies topologies defining how a system interconnects a set of components and gives the desired specification the composed system should adhere to. We believe that in order to address the above challenge existing approaches for synthesizing reactive systems, e.g., LTL synthesis, need to be extended with parameterized propositions. Therefore, we propose to use PDDL [13, 11] (the *de-facto* language in AI planning) together with a dedicated predicate for system-level modelling. The dedicated predicate allows us to distinguish between angelic (aka controlled) and demonic (aka uncontrolled) non-determinism. This enables the user to model a system with parameterized control actions as well as an adversary environment (Section 2).

Challenge B: Efficient game solving. After the system is modelled, our engine translates the model into a game with a winning condition (e.g., reachability, Büchi, Generalized Reactivity-1 conditions [23]) that depends on the specification. Then, it solves this game and synthesizes high-level control actions; from these actions, executable code for dedicated platform is created by automatic refinement (Section 3). Although high-level language facilitates the use of modeling, the size of the created game arena is usually very high. This lowers the speed of game solving drastically and hinders its applicability to complex systems. In this work, we show that efficient synthesis can be achieved by solving the game on a subarena of interest (based on the specification). Our main contribution is to demonstrate that *in the context of controller synthesis, techniques from program optimization are useful to solve game locally* leading to significant performance gain in the synthesis time (Section 4). Our methodology is based on static checking and can, therefore, be used as a preprocessing step in existing reactive synthesis frameworks.

Challenge C: Ease of understanding. Due to the use of PDDL, the generated strategy can already be presented in a user-understandable way. Many applications are reactive systems, in which code blocks chained together and executed repeatedly. As programmers are used to sequential forward reasoning, it is sometimes difficult to understand strategies for such systems, in their reactive form usually based on backwards reasoning. Therefore, for reachability games, we sequentialize the generated strategy based on extracting witness from the attractor computation. This transformation is based on performing a forward analysis using the synthesized strategy [1]. This allows the user to freely choose between sequentialized-reactive or reactive code.

In our to evaluate our approach (Section 5), we have extended our open-source tool GAVS+, which allows users to automatically synthesize Java-like controller programs (with support of various winning conditions) to accept PDDL descriptions with game-semantics. We have also implemented the optimizations proposed in this paper and evaluate them on several examples including the FESTO modular production systems (MPS). For FESTO MPS, we have created scripts supporting automatic translation from our synthesized behavioral-level code to executable[2]. Finally, we give a brief summary of existing work and propose future directions (Section 6, 7).

[1] Due to space limit details are omitted here.

[2] Visit (http://www.youtube.com/watch?v=Sb3bre916o4) for a short video showing the FESTO MPS executing automatically synthesized code.

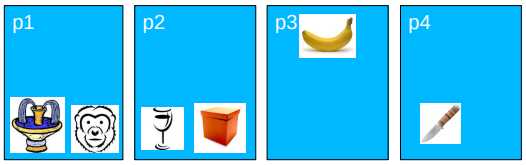

Figure 1: An illustration for the monkey experiment in AI.

```
(define (domain monkey)
  (:requirements :strips :negative-preconditions :game)
  (:constants monkey box knife bananas glass waterfountain)
  (:predicates (location ?x) (on-floor) (at ?m ?x) (hasknife) (onbox ?x)
               (hasbananas) (hasglass) (haswater) (P0TRAN) (banana-moved))

  ;; moves by the monkey
  (:action GO-TO
      :parameters (?x ?y)
      :precondition (and (location ?x) (location ?y) (on-floor)
                         (at monkey ?y) (P0TRAN))
      :effect (and (not (P0TRAN)) (at monkey ?x) (not (at monkey ?y))))
  (:action GRAB-BANANAS
      :parameters (?y)
      :precondition (and  (location ?y) (hasknife)
                          (onbox ?y) (at bananas ?y) (P0TRAN))
      :effect (and (not (P0TRAN)) (hasbananas)) )
  (:action CLIMB-UP ...)
  ...
  ;; moves by the experimenter (take banana)
  (:action MOVE-BANANAS
      :parameters (?x ?y)
      :precondition (and (not (P0TRAN)) (not (banana-moved))
                         (location ?x) (location ?y) (at bananas ?y))
      :effect (and (P0TRAN) (at bananas ?x) (not (at bananas ?y))
                    (banana-moved)))
  ;; moves by the experimenter (do nothing)
  (:action DO-NOTHING
        :parameters ()
        :precondition (and (not (P0TRAN) (banana-moved)) )
        :effect (and  (P0TRAN) (not (banana-moved))) )
)
```

Figure 2: The domain instance of the monkey experiment described using PDDL.

2 Modeling Games using PDDL

We illustrate our extension to PDDL using a modified version of the money example [2], a classical example from artificial intelligence with the goal to develop intelligence for a (robot) monkey to accomplish certain tasks. Our extension focuses on a part of PDDL, which we have commonly seen in our collected examples. The formal syntax of this part of PDDL can be found in the appendix. For details concerning the PDDL language, we refer the readers to [13, 11].

Monkey example [2, 14]. Consider the scenario shown in Figure 1. It has four positions `p1`, `p2`, `p3` and `p4`. In each of the positions a box, a knife, a fountain, a glass, or the monkey can be placed depending on the initial configuration. The banana is hanging on the top of one location. The goal of the monkey may vary from simple reachability (e.g., find the banana and eat it) to complicated ones involving subgoal achievement (e.g., repeatedly drink water and eat banana). In our setting, we assume that the experimenter can interfere and move the banana once in a while (e.g., every second step). We take a turn-based view, i.e., the monkey and the experimenter perform their moves in alternation. This is a common assumption because it rules out only truly concurrent actions. Throughout the paper, we refer to the system (the monkey) as player-0 and the environment (the experimenter) as player-1.

In general, an instance in PDDL consists of two parts [13]:

```
1  (define (problem pb1)
2    (:domain monkey)
3    (:objects p1 p2 p3 p4)
4    (:init (location p1) (location p2) (location p3) (location p4)
5           (at monkey p1) (on-floor) (at box p2) (at bananas p3) (at knife p4)
6           (at waterfountain p1) (at glass p2)
7    )
8    (:goal (hasbananas))
9  )
```

Figure 3: The problem instance of the monkey experiment described using PDDL.

1. A ***domain*** containing a parameterized system description that includes *constants*, *predicates* (ranging over constants and objects) and *actions* (*action schemata* that modify predicates). A domain usually also comes with a set of *requirements*, which gives details about with concepts a planner needs to understand in order to solve problems in this domain.
2. A ***problem*** specifying the task the planner is intended to solve. It contains *objects*, the *initial configuration* and the *goal specification*.

Underlying transition system. A PDDL domain together with a problem definition defines a transition system. A state of this system is given by an evaluation of the predicates. A predicate is either true or false depending on the value of its arguments, which can be an object or a constant.[3] E.g., assume we have a predicate `at` taking two arguments, written as `(at ?x ?y)`, and we have a constant `monkey` and two objects `p1` and `p2`, then there are $3 \cdot 3$ instances of the predicate `at`, each of them can be true are false. So, the PDDL description defines a system with $2^{3 \cdot 3}$ states (of which some made be unreachable depending on the initial configuration and the action definitions). Formally, let P_i to be the set of predicates of arity i with i ranging from 0 to some finite number a. Furthermore, assume that all parameters of the predicates are from the same domain D of constants and objects. Then, a configuration or state of the system is an element of the tuple $\mathbb{B}^{|P_0|} \times \mathbb{B}^{|P_1| \cdot |D|} \times \ldots \times \mathbb{B}^{|P_a| \cdot |D|^a}$.

The initial configuration, defined in the problem definition, describes the initial state of the transition system. In PDDL, all predicates that do not appear in the description of the initial configuration are considered to be false. Action schemata describe how predicates changes and, therefore, define the transitions in the underlying transition system. In general the transition system is non-deterministic. It is the task of the planner to find a path through the system that satisfies the given goal specification. (PDDL assumes that all predicates that are not explicitly used in the goal specification can take an arbitrary value.) In our setting some of the transitions (i.e., changes of predicates) cannot be controlled by the planner, e.g., if the experimenter moves the banana, the corresponding changes of the predicates it not controlled by the monkey (the planner).

Adding uncontrollable behavior. We add a "degree of control" to the core of PDDL allowing us to model the uncontrollable moves of the environment. It consists of two simple extensions.

1. We introduce a binary predicate `P0TRAN` in the domain description to partition player-0 and player-1 states and transitions. For a given configuration, when `P0TRAN` is evaluated to true, it is a system (player-0) configuration; otherwise it represents an environment (player-1) configuration. Note that we do not require a syntactic extension of PDDL, as the use of `P0TRAN` falls within the syntax of the PDDL language. However, we use a non-standard interpretation for the predicate `P0TRAN` by assuming uncontrollable (demonic) non-determinism, when the predicate is true. The change between system and environment configurations is done by executing actions and is in the responsibility of the user defining the actions.
2. In the requirement field of the domain, denoted by (`:requirements`), we introduce the new requirement `:game` to indicate that we expect that the predicate `P0TRAN` exists and that it has a different semantics. (For technical reasons, we also have to add the requirement `:negative-preconditions`,

[3] PDDL also allows using types. Our system can handle types and we extensively use them in our experiments but for simplicity we do not describe them here.

because in environment moves, we use the negative precondition `(not P0TRAN)`, and PDDL requires to state the use of negation in preconditions explicitly.)

Monkey example (continued). Figure 2 shows the domain description of the monkey experiment.

- There are six constants (Line 3), called `monkey`, `box`, `knife`, `bananas`, `waterfountain`, and `glass`.
- In Line 4 and 5, we define several predicates. E.g., `(location ?x)` takes an object (using one variable `?x`) and is evaluated to `true` or `false`. Note that predicates get their meanings through the way they are used and changed in actions. As described previously, the predicate `P0TRAN` is introduced for state partitioning.
- Starting from Line 7, we define action schemata for the domain. For example, consider the action `GRAB-BANANAS` which takes one *parameter* `?y` (Line 14). The action can only be executed if the *precondition* is satisfies. For the action `GRAB-BANANAS` (Line 15-16), we required that the monkey is at location `?y`, has the knife (`hasknife`), and stands on the box (`onbox ?y`). Furthermore, the banana needs to be at this location `?y` as well. Finally, the predicate `P0TRAN` specifies that it is a player-0 transition. The *effect* of `GRAB-BANANAS` (Line 17) is that the predicate `hasbananas` is set to `true` and `P0TRAN` is set to false (allowing the experimenter to perform the next move).
- The two actions `MOVE-BANANAS` and `DO-NOTHING` can be performed by player-1 (the experimenter); the others belong to the monkey. For `MOVE-BANANAS`, it can only be triggered when predicate `banana-moved` is `false`. After execution, it is set to `true`, and the experimenter need to wait until `banana-moved` is reset `false` after executing `DO-NOTHING`.

Figure 3 shows the problem description of the monkey experiment.

- The problem is named `pb1` (Line 1) and is connected to the `monkey` domain (Line 2).
- The objects using in the problem are given in Line 3. Except constants existed in the domain, four additional objects (`p1`, `p2`, `p3`, and `p4`) representing locations are included.
- The *initial configuration* specifies the scenario, in which the monkey is in `p1`, standing on the floor having neither the knife nor the glass. E.g., `location(p1)` appears in the initial configuration means that `location(p1)` is evaluated to `true` in the initial state; `location(monkey)` does not appear in the initial configuration means that `location(monkey)` is initially evaluated to `false`. As no typing is used in this example, setting `location(p1)`, ..., `location(p4)` to be `true` ensures that the monkey can only move between `p1` to `p4` (see `GO-TO` action for details).
- The goal of the monkey is described in the *goal statement* (Line 8) saying that the monkey should get the banana, i.e., the purpose of the planning problem is to reach the configuration, in which `hasbananas` is true. The goal statement turns into a reachability, Büchi, or safety condition depending on the selected synthesis engine.

3 Constructing and Solving Symbolic Games

Our engine first combines an input instance from the domain and the problem (domain-problem binding) and builds a symbolic representation of the underlying transition system. Then, it searches for a winning strategy using different synthesis algorithms depending on the specification. These two tasks can be divided into the following steps.

1. Based on the number of objects and predicates used in the problem description, we declare corresponding Boolean variables to represent sets of states in the symbolic form. Each (BDD) variable is represents in instantiated predicate, i.e., each parameter is concretized with a constant or an object from the domain. More precisely, for each a predicate $p(x_1, \ldots, x_n)$ with arity n over domain D,

we generated $m^{|D|}$ variables. We use $p_{(d_1,\ldots,d_n)}$ with $d_i \in D$ to denote the variable indicating if predicate p concretized with objects $d_1, \ldots, d_n$ is true or false. Note that in general this number of variables is required, because the number of states described by a PDDL description with an n-ary predicate over a domain D is $2^{m^{|D|}}$. In the next section, we show how to reduce this number by analyzing the domain-problem binding.

2. The symbolic transition relation is created as follows: for each action, the algorithm first assigns an object from the domain to each parameter. Then, it recursively traverses the abstract syntax tree (AST) of the precondition and postcondition to build up the formula. During the traversal of the AST of the postcondition, it explicitly records variables being updated. After traversal, if a variable v is not updated, we conjunct the formula $v \leftrightarrow v'$ to ensure that the value of variable v remains unchanged (v' stands for the primed version). The detailed steps are as follows.

   (a) Perform symbolic encoding to create the set of transitions for player-1.

   (b) Create a list L, where for each element in the list, it is an player-0 action whose parameters are concretized by the domain of objects. E.g., in the monkey example, the symbolic representation of `GRAB-BANANAS(p1)` is stored as an element in the list. List L is used to interpret the resulting strategy in later stages[4]. The set of transitions for player-0 amounts to the disjunction over elements in list L.

3. The selection of engines is (e.g., reachability, GR-(1)) done by the user. For reachability games, a backward attractor computation engine is invoked. During the attractor computation, we continuously record the set of transitions T that be forced to reach a goal state. When the initial state is contained in the attractor, the computation stops, and we intersect T with each element δ (concretized action) in L to derive the precondition to perform action δ (e.g., derive the precondition when to execute `GRAB-BANANAS(p1)`). Finally, we parse the precondition and create Java-like statements. For safety and Büchi games similar steps are applied. The engine also handles combinations of reachability and safety conditions, and the Generalized Reactivity-(1) condition [23].

4 Program Optimization for Local Game Solving

The standard translation, described in the previous section, of a PDDL description to a corresponding symbolic game usually creates a very large number of Boolean variables. For instance, consider the Hanoi tower domain (a classical domain taken from the PDDL4J library [2]) shown in Figure 4. It contains one unary predicates `clear` and two binary predicates `on` and `smaller`. Under a problem of three pegs and eight disks, written in PDDL as `(:objects peg1 peg2 peg3 d1 d2 d3 d4 d5 d6 d7 d8)` and leading to a domain of size eleven, the number of variables for the symbolic encoding is equal to $2 \cdot (11 + 2 \cdot 11^2) = 506$. (The constant factor 2 in front is due to the fact that we have the variables and their primed versions to encode the transitions relation.) Although, the use of typing (i.e., separate between pegs and disks) helps to alleviate the number of used variables, it does not solve it, and it is easy to find examples with typing, in which the number of variables go beyond the scope of any BDD package[5]. Recall that in general we cannot do better, because the number of states described by a PDDL description with one unary and two binary predicates over a domain with eleven elements, is $2^{11+2\cdot 11^2}$. However, by carefully analyzing the domain-problem binding and identifying relevant regions, we can be much more efficient as our experimental results show.

In the following we present a set of optimizations techniques that identify relevant regions of the game graph, which we call subarenas or local games, that are sufficient to solve the game. The key idea behind the optimizations is to use simple static analysis techniques over (a) domain-problem binding and (b) the

[4] When viewing the PDDL engine as LTL synthesis tools, this step brings difference to others such as Unbeast [8], where in these tools only bit-patterns can be represented as output. With such list L, we can conjunct the generated strategy with concretized actions (both represented in BDDs) in L to derive precise preconditions for executing each action. This makes our generated strategy easier to understand.

[5] Note that since we are in a game setting, every step in the game requires a quantifier alternation, and BDDs are currently the most efficient way to handle quantified Boolean formulas.

```
(define (domain hanoi)
    (:requirements :strips)
    (:predicates  (clear ?x) (on ?x ?y) (smaller ?x ?y) )
    (:action MOVE
        :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to ?disc) (on ?disc ?from) (clear ?disc)(clear ?to))
        :effect (and (clear ?from) (on ?disc ?to) (not (on ?disc ?from))(not (clear ?to)))
    )
)
(define (problem pb1)
    (:domain hanoi)
    (:requirements :strips)
    (:objects peg1 peg2 peg3 d1 d2 d3)
    (:init
        (smaller peg1 d1) (smaller peg1 d2) (smaller peg1 d3)
        (smaller peg2 d1) (smaller peg2 d2) (smaller peg2 d3)
        (smaller peg3 d1) (smaller peg3 d2) (smaller peg3 d3)
        (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
        (clear peg2) (clear peg3) (clear d1)
        (on d3 peg1) (on d2 d3) (on d1 d2)
    )
    (:goal  (and (on d3 peg3) (on d2 d3) (on d1 d2)))
)
```

Figure 4: Hanoi tower domain and problem (3 disks and 3 pegs) described using PDDL.

goal specification to discover regions that are sufficient to solve the game but can be represented by a smaller set of possibly fresh variables.

Similar ideas have been applied in verification to, e.g., reduce the number of variable in a SAT instance (see Section 6). Intuitively, in verification any reachable state is a "relevant" state, while in synthesis only winning reachable states are relevant. There are several reasons why such techniques are particularly interesting in the context of synthesis.

1. They simplify the computed strategy and therefore improve the quality of the synthesized code by making it easier to understand.

2. Currently, the most efficient reactive synthesis engines are based on BDDs and a reduction of the number of used variables, can be crucial for applying synthesis at all.

3. Finally, we can re-encoding the problem with the specification in mind, e.g., if a safety specification requires two predicates to have the same value, we can encode them using a single variable. This allows us also to do optimizations directly on the specification (see Section 4.4).

4.1 Constant Replacement

The aim is to find a set of variables (predicates with concretized predicates) that have a constant value during the synthesis process. These variables can be replaced by their value (`true` or `false`) during the creation of symbolic transition system.

Analysis step. We present a simple heuristic to detect predicates with constant values proceeds. Given a predicate $p(x_1, \ldots, x_n)$ with $x_1, \ldots, x_n$ from the domain D. If p does not occurs in an effect (i.e., post-condition) of any action schema, then p is constant for any instantiation and every variable in $\{p_{(d_1,\ldots,d_n)} \mid d_1, \ldots, d_n \in D\}$ can be replaced by its value. More precisely,

- if $p(d_1, \ldots, d_n)$ occurs positively in the initial configuration of the problem, then $p_{(d_1,\ldots,d_n)}$ is replaced by `true`,
- otherwise $p_{(d_1,\ldots,d_n)}$ is replaced by `false`.

Recall the problem of Hanoi tower (Figure 4), the predicate `smaller` is used to describe the relation between (a) peg and disks and (b) two disks. From the action schema in Figure 4 it follows that predicate `smaller`, once concretized, never changes its value (e.g., `(smaller peg1 d1)` remains `true`).

Therefore, we can safely ignore the construction of `smaller` and replace the occurrence in concretized transitions by its initial value. E.g., in Figure 4 during the symbolic game construction, when we encounter `(smaller peg1 d1)` in the precondition, we replaced it by `true`. Under a problem of three pegs and eight disks, the number of variables changes from 506 to $2(11 + 11^2) = 264$ leading to a reduction of variables by 48%.

4.2 Binary Compaction

The idea of our second technique is to perform binary encoding of a set of Boolean variables that are mutually exclusive, i.e., at most one of them is true in any reachable state. E.g., recall the predicate `(at ?object ?location)` from the money example with the intuitive meaning that it specifies whether an object is at a certain location. Assume we have already concretized the parameter `?object` to be the `monkey`. The domain of the parameter `?location` contains four locations: `p1`, `p2`, `p3`, and `p4`.[6] The direct encoding would generate four variable, one for each location. If we assume that the monkey can be in at most one location, then we can encode its location (being in `p1`, `p2`, `p3`, `p4` or in none of them) using only three variables. In general, we can reduce the number of variables from n to $\lceil \log(n+1) \rceil$. If we check for exactly one instead of at most one the number reduces to $\lceil \log n \rceil$.

Analysis step. We present a heuristic to detect variables that are mutually exclusive for action schemata with pre- and postconditions that are given as a conjunction of literals, which are predicates or their negations. (All our benchmarks as well as the examples from the PDDL4J library follow this convention.)

Predicate with one parameter. Given predicate $p(x)$, where x ranges over elements from the domain D, we apply binary compaction, if following conditions hold:

1. In the initial configuration, there is at most one element $d \in D$ such $p(d)$ holds.
2. For every parameterized action and for all of its parameters $x_1, \ldots, x_n$, one of the following conditions holds:
   (a) $p(x_i)$ does not occur in the postcondition, or
   (b) $p(x_i)$ appears only negatively in the postcondition, i.e., $\neg p(x_i)$, or
   (c) if $p(x_i)$ appears in the postcondition, then either (i) $p(x_i)$ appears in the precondition, or (ii) there is another parameter x_j such that $p(x_j)$ is in the precondition and $\neg p(x_j)$ is in the postcondition.

If the above check succeeds, we use the Boolean variables $V = v_0, \ldots, v_m$ with $m = \lceil \log(|D| + 1) \rceil - 1$ to represent the truth assignments to the predicates $p(d_1), \ldots, p(d_{|D|})$. More precisely, we first assign to each element $d \in D$ an integer value between 0 and $|D| - 1$. Let $\mathrm{val}(d)$ be the function mapping d to its value; the value is used to represent that the predicate p concretized with element d is evaluated to true, while for all other elements $d' \in D$, $p(d')$ is false. We use $\mathrm{enc}(d)$ to represent this encoding, e.g., assume p ranges over $D = d_1, d_2, d_3, d_4, d_5$ and $\mathrm{val}(d_1) = 3$, then $\mathrm{enc}(d_1) = v_0 \wedge v_1 \wedge \neg v_2$ with $V = \{v_0, v_1, v_2\}$. We use the value $|D|$ to represent that none of the predicates is true, i.e., $\forall d \in D : \neg p(d)$. We restrict the valid assignments to the variables in V to values $\leq |D|$. This can be done by conjoining a Boolean formula encoding $\leq |D|$ to the symbolic transition relation. Note that the length of this formula is at most $2 \cdot |V|$, i.e., ≤ 5 can be encoded as $\neg v_2 \vee (v_2 \wedge \neg v_1)$. During the construction of symbolic transition relation for an action, the general idea is to replace every occurrence of p by its corresponding encoding, i.e., for all $d \in D$,

- if $p(d)$ or $\neg p(d)$ appears in the precondition translate it to $\mathrm{enc}(d)$ or $\neg\mathrm{enc}(d)$, respectively.
- if $p(d)$ or $\neg p(d)$ appears in the postcondition translate it to $\mathrm{enc}'(d)$ or $\neg\mathrm{enc}'(d)$, respectively, where $\mathrm{enc}'(d)$ refers to the formula obtained from $\mathrm{enc}(d)$ by replacing the variables in V with their primed version.

[6] Strictly speaking, without typing the domain contains also the constants `monkey`, `box`, `knife`, ...

- If neither $p(d)$ nor $\neg p(d)$ appears in the postcondition, we add the constraint $p(d) \leftrightarrow p'(d)$.

In order to avoid adding many constraints of the form $p(d) \leftrightarrow p'(d)$ and speed up the implementation, we implemented a set of additional rules, e.g., if for all $d \in D$, $p(d)$ does not appear in the postcondition (meaning the assignments to $p(d_1), \ldots, p(d_{|D|})$ do not change) and we can use the formula $\bigwedge_{v \in V} : (v \leftrightarrow v')$, or alternatively, if there is only one $d \in D$ s.t. if $p(d)$ appear in the precondition and $\neg p(d)$ in the postcondition, we use the formula $\text{enc}(d) \wedge \text{enc}'(|D|)$.

Predicate with two or more parameters. Currently, for predicates with more parameters, we check if binary encoding can be applied to the last parameter. E.g., for predicate $p(x_1, x_2)$ with x_1, x_2 from the domain D, we check for every $d \in D$, if the value of the predicates $p(d, d_1), \ldots, p(d, d_{|D|})$ with $d_i \in D$ are mutual exclusive during the synthesis process.

4.3 Goal-indifferent Variable Elimination

Our last optimization aims to identify variables that are unnecessary to satisfy the goal specification, i.e., if there exists a winning strategy that modifies these variables, then there exists another winning strategy that leaves them unchanged.

Recall the monkey example, if the goal of the monkey is to fetch the banana, then whether the monkey holds the glass can be viewed as irrelevant: whenever there exists a strategy which grasps the glass (and thus changes the `hasglass` variable), there exists another strategy that does not include grasping the glass. This knowledge corresponds to *inferring from the specification a set of actuators that are not required in the concrete setup to reach the goal*. This computation corresponds to computing the cone-of-influence computation used in verification.

Analysis step. The analysis step for reachability games, which can also be applied to GR-1 games, proceeds as follows: first, add all variables that appears in the goal condition, i.e., the variables that do not have a "don't care" value in the goal condition, in a set S. We use the set S to store all the variables that have a potential influence on the goal. Then select a set of actions Δ whose postcondition (i.e., the `:effect` field in an action) changes variables in S. Add the set all variables S' which appear in the precondition (i.e., the `:precondition` field in an action) of Δ to S. Repeat until S saturates, then treat variables not contained in S as $\widehat{S}$. Denote the set of variables used either in the precondition or in the postcondition by player-1 transitions as S_{P_1}. Then variables within $\widehat{S} \setminus S_{P_1}$ can be omitted for construction.

The correctness relies on the following observations:

- For player-1, whether it can perform a move is insensitive to variables in $\widehat{S} \setminus S_{P_1}$ (i.e., outside S_{P_1}). Therefore, these variables can be omitted for player-1, and we do not restrict the ability of player-1.
- For player-0, to reach the goal, variables within $\widehat{S} \setminus S_{P_1}$ (i.e., within $\widehat{S}$) are not useful at all, as they do not appear in all of the possible control decisions leading to the goal.

E.g., consider the monkey example, where the goal is to fetch the banana. Initially, S contains one element `hasbanana`. Action `GET-BANANA(p3)` contains `hasbananas` in the postcondition, so we add `(location p3)`, `hasknife`, `(at bananas p3)`, and `(onbox p3)` to S, as their values can be sensitive to `hasbananas`. Continue the process, it can be concluded that `hasglass` and `haswater` are not included in S. Also, for the experimenter, the set of variables used in `MOVE-BANANAS` and `DO-NOTHING` do not include `hasglass` and `haswater`. Therefore, in our symbolic encoding we safely ignore `haswater` and `hasglass`. Subsequently, we ignore constructing actions `PICK-GLASS` and `GET-WATER`, as they only update `hasglass` and `haswater`.

4.4 Optimizing Specifications

In the following, we show on an example that we can also apply the optimizations directly on the goal specification. We consider specifications in Generalized Reactivity-1 (GR-1) format [23], a subset of LTL expressive enough to encode most of specifications that occur in practice.

Recall that the general ideas is to detect a set of states such that computing a strategy on these states is sufficient to solve the game. Then, we re-encode this set of states with a hopefully smaller set of possibly fresh variables such that we can derive the initial variables from the new set of variables. E.g., assume we aim to synthesis a fair arbiter for n clients. The interface to each client consists of two variables (i) r_i and (ii) g_i indicating if (i) the client requested the resource and (ii) if the client is granted the resource. Furthermore, we assume that the arbiter follows the following protocol: once the resource if granted, the grant signal g_i is kept high until the client drops its request r_i. Note that in order to be able to implemented a fair arbiter, we need to assume that requests are drop eventually. Formally, our specification consists of the following sets of desired properties. Given the specification using LTL-operators **always**(φ) (i.e., φ is always true), **next**(φ) (i.e., φ holds in the next step), and **eventually**(φ) (i.e., φ holds at some point in the future). Note that **always**(**eventually**(φ)) means that φ will hold infinitely often. For a details description of LTL we refer the reader to [19].

1. **Mutual exclusion** φ_e**:** for client i and $j \neq i$, **always**$(\neg g_i \vee \neg g_j)$
2. **Fairness** φ_f**:** for all client i, **always**$(r_i \rightarrow$ **eventually**$(g_i))$. In GR-1 each of these properties is encoded using a new atomic proposition s_i and the following formulas. Note that these formulas represent a symbolically encoded deterministic Büchi automaton that keeps track of the property.
   (a) **Initialization:** $\neg s_i$
   (b) **Transition:** (i) **always**$(\neg s_i \wedge (\neg r_i \vee g_i) \rightarrow$ **next**$(\neg s_i))$, (ii) **always**$(\neg s_i \wedge r_i \wedge \neg g_i \rightarrow$ **next**$(s_i))$, (iii) **always**$(s_i \wedge \neg g_i \rightarrow$ **next**$(s_i))$, and (iv) **always**$(s_i \wedge g_i \rightarrow$ **next**$(\neg s_i))$
   (c) **Fairness: always**(**eventually**$(\neg s_i))$
3. **Protocol** φ_p**:** for all client i, **always**$(r_i \wedge g_i \rightarrow X(g_i))$
4. **Fair clients** φ_a**:** for all client i, **always**$(r_i \wedge g_i \rightarrow$ **eventually**$(\neg r_i))$. The GR-1 encoding again introduces for each client a new atomic proposition t_i. Formally, the corresponding specification is given by
   (a) **Initialization:** $\neg t_i$
   (b) **Transition:** (i) **always**$(\neg t_i \wedge (\neg r_i \vee \neg g_i) \rightarrow$ **next**$(\neg t_i))$, (ii) **always**$(\neg t_i \wedge r_i \wedge g_i \rightarrow$ **next**$(t_i))$, (iii) **always**$(t_i \wedge r_i \rightarrow$ **next**$(t_i))$, and (iv) **always**$(t_i \wedge \neg r_i \rightarrow$ **next**$(\neg t_i))$
   (c) **Fairness: always**(**eventually**$(\neg t_i))$

The final specification is given by $\varphi_a \rightarrow \varphi_e \wedge \varphi_f \wedge \varphi_p$ stating that if the clients are fair, then the arbiter satisfies its requirements. This specification refers to $4 \cdot n$ atomic propositions (r_i, g_i, s_i, and t_i for each client).

Optimization step 1. In order to reduce the number of atomic propositions, we can use the fact the grant signals g_i are mutually exclusive. We introduce a set of new variables $g'_0, \ldots, g'_{\lceil \log(n+1) \rceil - 1}$ to represent the grant signals in binary encoding, e.g., for three clients, we get two variables g_0 and the grants signals g_0, g_1, and g_2 are encoded as usual, i.e., $g_0 = \neg g'_1 \wedge \neg g'_0$, $g_1 = \neg g'_1 \wedge g'_0$, and $g_2 = g'_1 \wedge \neg g'_0$. The unused assignment $g'_1 \wedge g'_0$ is used to encode that no client is currently accessing the resource. Let enc(i) be the formula representing the binary encoding of grant i using the variables g'_i. We obtain for alternative encoded specification by replacing all occurrences of g_i by $enc(i)$. Furthermore, since our encoding ensure mutual exclusion, Property (1) is not required anymore. Note that the new specification refers only to $3 \cdot n + \lceil \log(n+1) \rceil$ atomic propositions.

Optimization step 2. By inserting Property (3) into Property (4), we obtain the following four properties: (i) **always**$(\neg t_i \wedge (\neg r_i \vee \neg g_i) \rightarrow$ **next**$(\neg t_i))$, (ii) **always**$(\neg t_i \wedge r_i \wedge g_i \rightarrow$ **next**$(t_i \wedge g_i))$,(iii) **always**$(t_i \wedge r_i \wedge g_i \rightarrow$ **next**$(t_i \wedge g_i))$, and (iv) **always**$(t_i \wedge \neg r_i \wedge g_i \rightarrow$ **next**$(\neg t_i))$from which we can easily deduce that a synthesized system is only correct if t_i is true, implies that g_i is true as well, i.e., it needs to respect the derived property **always**$(t_i \rightarrow g_i)$ for all clients i.[7] This allows us to conclude that the variables

[7]Note that if we cannot deduce a property necessary for simplification by syntactic checks, we can always build a transition system by considering only the involved requirements and model check the property.

$t_1, \ldots, t_n$ are also mutual exclusive, and we could apply the same reduction as shown above. Intuitively, such an encoding would remember only the id of the client for which Property (4) needs to be tracked. However, we can also establish a stronger property, namely, that this id is given by the value of the variable g'_i used in enc(i). Using this property, Property (4) simplify to **always**$(r_c \wedge g_c \rightarrow$ **eventually**$(\neg r_c))$, where c refers to the *current client* (given by value of the variables g'_i). This allows us to replace a conjunction of n properties (one for each client) by a single property. The corresponding variable encoding uses only three new variables r_c, g_c, and t. Variables r_c and g_c point to the request and grant signal of the client currently accessing the resource, i.e., $\forall_i$: **always**$(\text{enc}(i) \rightarrow (r_c \leftrightarrow r_i) \wedge g_c)$. Note that since Property (4) is an assumption on the environment, the variables r_c and g_c can be chosen arbitrarily by the environment if none of the clients accesses the resource. Variable t is used to check if the client currently accessing the resource is fair. Formally, we obtain the following specification, which is similar the one for the t_i-variables. (Recall that enc$(n+1)$ means that none of the clients is currently accessing the resource.)

1. **Initialization:** $\neg t$
2. **Transition:** (i) **always**$(\neg t \wedge (\neg r \vee \text{enc}(n+1)) \rightarrow$ **next**$(\neg t))$, (ii) **always**$(\neg t \wedge r \wedge \neg\text{enc}(n+1) \rightarrow$ **next**$(t))$, (iii) **always**$(t \wedge r \rightarrow$ **next**$(t))$, and (iv) **always**$(t \wedge \neg r \rightarrow$ **next**$(\neg t))$
3. **Fairness: always**(**eventually**$(\neg t)$)

The variables $t_0, \ldots, t_n$ are derived from t as follows: $t_i = (\text{enc}(i) \rightarrow t) \wedge (\neg\text{enc}(i) \rightarrow \text{false})$

Our final encoding used (i) n request signals (r_i), (ii) $\lceil \log(n+1) \rceil$ grant signals (g'_i), (iii) n variables to encode Property (2) (s_i), and (iv) 2 variables to encode Property (4) (s and t), which give a total of $2 \cdot n + \lceil \log(n+1) \rceil + 2$ variables.

5 Implementation and Evaluation

In order to evaluate our optimizations, we have extended the functionality of our tool GAVS+[8] and performed a case study. In the following, we give a summary of the experiments and results. The models, demonstrators, and a detailed discussion over results can be found in the appendix.

Basic evaluation. To obtain an idea concerning our preprocessing scheme in subarena creation, we use the Hanoi tower example as basic evaluation. Our technique enables us to solve significantly larger problems (Table 1). We also applied the techniques to basic planning problems, in which no adversary environment is modeled, and compare with other PDDL-based planners from the International Planning Competition (IPC) taking place every three years. We compare two planners (i) seq-opt-gamer [18], the winner of the last edition (IPC'2008)[9] and (ii) FastDownward [15], used as a starting point for many teams entering the competition. Despite the fact that our program is implemented in Java, it competes well with other two planners implemented in C++ on our benchmarks (see Table 3). In addition, we can create controllers for adversary environments and more complex conditions than considered by traditional planners. As expected, on smaller examples, our implementation is less efficient than the other tools.

Optimized specification evaluation . To show that the influence of optimizing the specification, we have synthesized an arbiter using the initial and the optimized specification presented in Section 4.4 using the synthesis tool Anzu [16]. We have used Anzu for two reasons: first, the core PDDL language currently implemented in GAVS+ does not support object equality, which makes it hard to model simultaneous requests from the environment. We will add equality in the next version of GAVS+. Secondly, using Anzu also allows us to show how the optimizations can be applied independently of the underlying synthesis tool. In Table 4, we show the total time Anzu needs to synthesize the arbiter for the initial and optimized specification depending on the numbers of clients. The times needed for the initial and optimized specification differ by an order of magnitude, e.g., for 20 clients the time reduces from about 500sec to 15sec.

[8] Software and examples (together with the synthesized behavioral code) are available at http://www6.in.tum.de/~chengch/gavs/simbesyn.html.

[9] Unfortunately, we are unable to obtain the winner of this year's competition.

Table 1: Experimental results (seconds)

Problem	GraphPlan[l]	Symbolic[n]	Symbolic[o]	Steps to goal
Hanoi 4	0.69	1.13	0.55	15 steps
Hanoi 5	9.11	1.83	0.60	31 steps
Hanoi 6	t/o	4.34	0.93	63 steps
Hanoi 7	t/o	23.98	1.612	127 steps
Hanoi 8	t/o	t/o	1.92	255 steps
Hanoi 9	t/o	t/o	2.83	511 steps
Hanoi 10	t/o	t/o	5.39	1023 steps
Hanoi 11	t/o	t/o	13.52	2047 steps

[l] Prebuilt algorithm in PDDL4J
[t/o] Timeout ($>$ 400 seconds)
[n] GAVS+: No optimization
[o] GAVS+: Optimization based on variable reduction

Table 2: Results for FESTO MPS domain

Testcase	Execution Time (sec)[H]	Remark
1	1.22	Simple planning
2	30.42	Storing based on color
3a	29.93	2 + object detection
3b	86.34	2 + object detection
3c	97.47	2 + object detection
4a	231.21	3 + capacity detection
4b	420.06	3 + capacity detection

[H] Change the default memory setting to 6000Mb for Java Virtual Machine

Table 3: Experimental results (seconds)

Problem	Symbolic^{o1}	Symbolic^{o2}	seq-opt-gamer	FastDownward^{H1}
Logistics pb0	1.09	1.02	0.88	3.29
Logistics pb1	6.85	7.12	9.47	m, t/o
Logistics pb2	10.62	16.92	p.e.	m, t/o
Logistics pb3	3.02	2.18	p.e.	p.e.
Logistics pb4	16.63	16.27	14.48	m, t/o
Barman pb1	72.52	23.74	45.63	39.26
Barman pb2	66.09	22.24	70.70	39.97
Barman pb3	66.67	20.73	99.15	39.47
Barman pb4	55.57	21.50	118.00	39.67
Visitall pb4full	0.90	0.87	0.68	0.09
Visitall pb5full	1.76	1.48	3.37	46.52
Visitall pb6full	15.65	23.63	t/o	m, t/o
Gripper pb4	1.91	3.33[1]	0.70	0.06
Gripper pb5	1.80	4.22[1]	1.15	0.80
Gripper pb6	31.22	57.18[1]	19.81	m, t/o
Gripper pb7	3.76	8.09[1]	6.24	m, t/o

[t/o] Timeout ($>$ 300 seconds)
[p.e.] Parser error
[m] Memory error
[$o1$] GAVS+: Optimization based on variable reduction
[$o2$] GAVS+: Optimization based on variable reduction and FORCE ordering [3]
[1] GAVS+: The FORCE heuristic is applied only once
[$H1$] FastDownward: option `-search "astar(blind)"` (see manual)

Table 4: Synthesis times (seconds)

#Clients	Time (original)	Time (optimized)
1	0.04	0.07
2	0.13	0.16
3	0.29	0.23
4	0.60	0.39
5	1.20	0.50
6	1.83	0.71
7	2.89	0.90
8	5.50	1.21
9	8.68	1.56
10	11.86	2.22
11	15.60	2.38
12	33.95	3.64
13	50.82	4.39
14	63.15	5.04
15	68.26	7.23
20	495.13	14.13
25	t/o	38.41
30	t/o	96.80

[t/o] Timeout ($>$ 1800 seconds = 30 min)

FESTO modular production system. We have also applied our techniques to synthesize controllers for the FESTO MPS[10]. The MPS was developed for educational purposes, and is a good approximation for a real industrial automation process. Each unit of the MPS processes small colored pieces that are made out of plastic or metal. Our demonstrator setup (see Figure 5) is composed of two modules, a processing unit and a storage unit. The processing unit is built up from a rotating plate with six locations, a height probe sensor that analyzes the shape of pieces and a drilling module that processes the pieces. The storage unit contains a robot arm used to store and retrieve the object to/from a three-layer rack with six storage locations each. Both units are connected via two unidirectional conveyor belts that deliver one piece at a time. Several levers/rods move the object between the conveyor belt and the units.

In order to model the system, we defined atomic actions for each component. Table 5 summarizes some parameterized behavioral-actions used to specify the system (see appendix for the detailed system model). The synthesized strategy specifies conditions with corresponding parameters about when to execute these actions. Our generated behavioral-level code (a file with roughly 500 lines including automatic indenting and comments) is easy to read. We use a script to perform translation from generated strategy to executable (each behavioral-action maps to a sequence of machine instructions). Table 2 summarizes the synthesis times for different testcases (see appendix for a detailed description). E.g., for Testcase 4b in Table 2, the standard encoding needs a total number of 593×2 variables, while with optimizations we only use 60×2

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

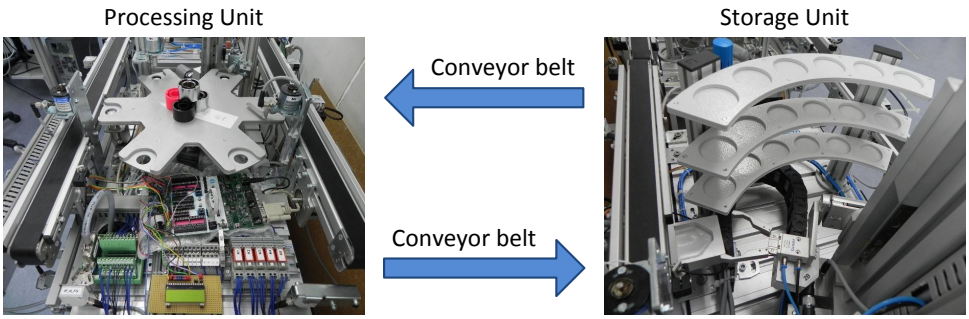

Figure 5: A demonstrator setup of the FESTO modular production system.

Table 5: Actions defined in the PDDL domain of a FESTO MPS system

Action	Parameter - Type	Intuitive meaning	Change
robot-move	`?pos1 ?pos2` - robotposition	robot arm moves from `?pos1` to `?pos2`	Sys ⇒ Sys
robot-pick	`?gri` - gripper `?pos` - robotposition, `?obj` - object	robot arm `?gri` picks object `?obj` (both at position `?pos`)	Sys ⇒ Sys
robot-check	`?pos` - robotposition	trigger sensor to check if `?pos` is occupied	Sys ⇒ Env
return-check	`?pos` - robotposition	return whether position `?pos` is occupied	Env ⇒ Sys
robot-drop	`?gri` - gripper `?pos` - robotposition, `?obj` - object	robot arm `?gri` drops object `?obj` (both at position `?pos`)	Sys ⇒ Sys
belt-move	`?obj` - object, `?pos1 ?pos2` - beltposition	belt transfers object `?obj` from `?pos1` to `?pos2`	Sys ⇒ Sys
drill-in	`?obj` - object, `?pos` - plateposition	drill (at position `?pos`) drills the object	Sys ⇒ Sys
plate-rotate	`?obj1 ... ?obj6` - object `?pos1 ... ?pos6 - ?plateposition`	rotate the plate to move the object	Sys ⇒ Sys
rod-push	`?obj` - object `?pos1 ?pos2` - (robotposition, beltposition, plateposition)	push the rod located between `?pos1` and `?pos2` to change the position of object `?obj`	Sys ⇒ Sys
trigger-color-sensor	`?obj` - object `?pos` - (robotposition, beltposition, plateposition)	trigger the color sensor located at `?pos`	Sys ⇒ Env
return-color-value	`?color` - colortype `?obj` - object `?pos` - (robotposition, beltposition, plateposition)	return the color value `?color` when the color sensor located at	Env ⇒ Sys
trigger-form-sensor	`?obj` - object `?pos` - (robotposition, beltposition, plateposition)	trigger the shape-detector located at `?pos`	Sys ⇒ Env
return-form-value	`?form` - formtype `?obj` - object `?pos` - (robotposition, beltposition, plateposition)	return the shape `?form` when the shape-detector located at `?pos` is on	Env ⇒ Sys

variables.

6 Related Work

We briefly compare our results with existing work in local game solving, planning in artificial intelligence, and LTL synthesis.

Our key insight is to perform faster synthesis by solving games locally. The concept of local game solving was largely used in parity or winning conditions with known complexity at least $\mathrm{P}\cap\mathrm{co}\text{-}\mathrm{NP}$ [12, 22]. Our concept to view program optimization as techniques for local game solving is new, and our focus is only to solve games with strategy finding in polynomial time, as when encountering large examples, we need to solve games symbolically. Related to our work is also work that aims to optimized the size of a SAT formula, e.g., Marinov et al. [20] use program optimization techniques to efficiently encode verification problem from the Alloy Analyzer. We use different optimization techniques and focus on the synthesis setting, which allows us to extract information directly from the specification.

In artificial intelligence, we can view the concept of synthesis as part of the nondeterministic planning (see an early Dagstuhl report [17] for their relations). PDDL-based tools such as seq-opt-gamer [18] or MBP [4] also use symbolic techniques to solve nondeterministic planning problems. However, to the best of our knowledge these tools do not support games with complex winning conditions (e.g., GR(1)). Furthermore, our program optimization scheme enables to scale to larger examples (see Table 3). ALisp [21] also performs synthesis by letting a user specify non-determinism over choices, but its strategy finding is based on machine learning.

There has been a lot of recent work on LTL synthesis [24, 8, 10, 23, 9, 5]. We use LTL synthesis techniques underneath but we focus on using a language that is easier to use. In addition, our optimization techniques can be applied as preprocessing step as shown in Section 4.4.

The tool implementation used in this paper is based on an early version of our tool [7] that could already translate PDDL with game extensions to symbolic game. The original purpose of the earlier version was to bridge between the AI and the verification community. However, we found the concept very powerful for system-level modeling and synthesis. The previous version was very basic: (i) it supported only reachability games, (ii) useful constructs such as typing and conditional effects were not included, and (iii) it used none of the presented optimization techniques, which made it very inefficient even for medium-size examples.

7 Summary and Future Work

We summarize the main contribution in our paper.

- For the ease of behavioral-level synthesis, we reinterpret PDDL to model games and reactive system design.
- To increase the speed of synthesis, we use program optimization techniques to solve games locally with high efficiency. These techniques can be used as a preprocessing step in any LTL synthesis tool.
- The concept is implemented as an extension of the GAVS+ tool, and we automatically synthesize and deploy programs for a control automation system.

In our ongoing work we aim to synthesize controllers with optimality criterion. E.g., it would be interesting to use sketching techniques [25] to parallelize or distribute the generated program to achieve improved performance. We are currently implementing an extension that allows us to assign finite costs to actions. Finally, in order to increase the speed of synthesis, we aim to (i) study how to develop more general program optimization techniques and (ii) replaced our Java-based BDD engine with native C implementation.

References

[1] CUDD: CU decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/. 13

[2] PDDL4J: Parser library for PDDL. http://sourceforge.net/projects/pdd4j/. 2, 2, 4, 6, B.1

[3] F. Aloul, I. Markov, and K. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI (GLSVLSI'03)*, pages 116–119. ACM, 2003. 3

[4] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Mbp: a model based planner. In *IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information (PRO-2)*, 2001. 6, B.4

[5] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy - a new requirements analysis tool with synthesis. In *CAV*, pages 425–429, 2010. 6

[6] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997. B.1

[7] 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-Verlag, 2011. 6

[8] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 272–275. Springer-Verlag, 2011. 4, 6

[9] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas. Temporal logic motion planning for mobile robots. In *ICRA*, pages 2020–2025, 2005. 6

[10] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for ltl realizability. In *CAV*, pages 263–277, 2009. 6

[11] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003. 1, 2, A

[12] O. Friedmann and M. Lange. Local strategy improvement for parity game solving. In *Proceedings of the 1st International Symposium on Games, Automata, Logics and Formal Verification (GandALF'10)*, volume 25, pages 118–131. EPTCS, 2010. 6

[13] 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. 1, 2, 2, 6, A, 1, 2, 11, 12, 6, 7

[14] C. C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI'69)*, pages 219–240, 1969. 2

[15] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006. 5, B.2

[16] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV'07*, pages 258–262. Springer-Verlag, 2007. 5

[17] H. Kautz, W. Thomas, and M. Y. Vardi. 05241 Executive Summary – Synthesis and Planning. In *Synthesis and Planning*, Dagstuhl Seminar Proceedings, 2006. 6

[18] P. Kissmann and S. Edelkamp. Gamer, a general game playing agent. *KI-Künstliche Intelligenz*, 25(1):1–4, 2011. 5, 6, B.2

[19] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*, volume 2. 1995. 4.4

[20] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard. Optimizations for compiling declarative models into boolean formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 632–637, 2005. 6

[21] B. Marthi, S. Russell, and D. Latham. Writing stratagus-playing agents in concurrent alisp. In *the IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005. 6

[22] D. Neider, R. Rabinovich, and M. Zimmermann. Solving muller games via safety games. Technical Report AIB-2011-14, RWTH Aachen, 2011. 6

[23] N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive (1) designs. In *VMCAI'06*, pages 364–380. Springer-Verlag, 2006. 1, 3, 4.4, 6, B.4

[24] S. Schewe and B. Finkbeiner. Bounded synthesis. In *ATVA*, pages 474–488, 2007. 6

[25] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 281–294. ACM, 2005. 7

```
<domain>          := (define (domain <name>)
                      [<extension-def>]
                      [<require-def>]
                      [<constants-def>]
                      [<predicate-def>]
                      <action-def>*)
<extension-def>   := (:extends <domain-name>)
<require-def>     := (:requirements <require-key>+)
<constants-def>   := (:constants <name>+)
<predicates-def>  := (:predicates <atomic formula skeleton>+)
<atomic formula skeleton> := (<predicate> <variable>*)
<predicate>       := <name>
<variable>        := ?<name>
<name>            := identifier
<require-key>     := Follow Table 5.1

<action-def>      := (:action <action functor>
                              :parameters ( <variable>* )
                              <action-def body>)
<action functor>  := <name>
<action-def body> := :precondition <goal description> :effect <effect>
<goal description> := (and <goal description>*) | (not <goal description>)
                        | (not <goal description>) | <literal(term)>
<literal(t)>      := <atomic formula(t)>
                      | (not <atomic formula(t)>)
<atomic formula(t)> := (<predicate> t*)
<term>            := <name> | <variable>
<effect>          := (and <effect>*) | <atomic formula(term)>
                      | (not <atomic formula(term)>) | (when <goal description> <effect>)
```

Figure 6: Extended BNF for the domain in PDDL (partially modified from [13]).

A Appendix A: PDDL Syntax

As the entire PDDL language is too broad for processing, we focus on a small portion of PDDL (which we refer it as *core PDDL*) which is commonly seen from our collected examples[11]. Table 6 summarizes the result, where requirements represent subsets of features categorized in PDDL. For the ease of explanation, we omit typing in all of our definitions (while it is implemented). For details concerning the PDDL language, we refer interested readers to [13, 11] for a full-blown manual.

Definition 1 (PDDL domain: Syntax [13]) *Define the Extended BNF[12] for a domain in PDDL by contents in Figure 6. Lines starting with characters ";" are comments.*

Definition 2 (PDDL problem: Syntax [13]) *Define the Extended BNF for a problem in (core) PDDL using contents in Figure 7 (for items defined previously in Definition 1, it is not defined repeatedly). We use the*

[11] PDDL is designed with the anticipation that only a few planners will handle the entire PDDL language [13].

[12] In Extended BNF form [13], symbol "*" represents *zero or more elements*, symbol "+" represents *at least one element*, symbol "|" represents `or`, and objects enclosed in square brackets are optional fields.

Table 6: Requirements supported in our implementation. Descriptions of each requirement are from the PDDL tutorial [13] and the PDDL4J library [2]

Requirement	Description	Supported?
`:strips`	Basic STRIPS-style.	Supported[1]
`:negative-preconditions`	Allows `not` in goal and preconditions descriptions.	Supported
`:disjunctive-preconditions`	Allows `or` in goal and preconditions descriptions.	Supported
`:equality`	Supports = as built-in predicate.	Supported
`:conditional-effects`	Allows `when` in action effects	Supported
`:typing`	Allows type names in declaration of variables.	Supported
`:safety`	Allows `:safety` conditions for a domain.	not supported in PDDL4J[3]

[1] Currently quantification over objects is not supported for simplicity issues. Users may rewrite the quantification by enumerating concrete objects in the problem. Also, we only support predicates with two parameters.

[2] We use (and extend) PDDL4J as our front-end language parser.

[3] In our implementation the user may specify safety constraints.

```
<problem>          := (define (problem <name>)
                              (:domain <name>)
                     [<require-def>]
                     [<object declaration>]
                     [<init>]
                     <goal>)
<object declaration>  := (:objects <name>+)
<init>             := (:init <literal(name)>+)
<goal>             := (:goal <goal description>)
```

Figure 7: Extended BNF for the problem in PDDL (partially modified from [13]).

<goal> field to store reachability, safety and Büchi conditions, and for generalized reactivity conditions they are offered to the synthesis engine as additional input.

B Appendix B: Evaluation

In this section, we compare our work with others together with a preliminary evaluation over our implementation. Results are collected from an Intel 3.0 Ghz Machine (with 4GB RAM). For GAVS+, we allocate 3000MB memory for the Java Virtual Machine.

B.1 Effect of variable reduction (subarena creation)

We first give a general evaluation concerning the improvement by our preprocessing. Table 1 (see main text) summarizes the performance measure on the classic example of Hanoi tower. We have also listed the result of the built-in GraphPlan algorithm [6] (a classic algorithm in AI planning) implemented in PDDL4J [2]. Under the preprocessing, we are able to solve more complex problems within reasonable amount of time. GraphPlan algorithm scales badly.

B.2 Plan Generation

We continue our evaluation by comparing our implementation with other tools. For this we have taken PDDL planners from the International Planning Competition (IPC'11). We have downloaded planners with executables available for download, namely *seq-opt-gamer*[13] [18] and *FastDownward* [15][14].

Examples in our benchmark suite are either from the PDDL4J library or from the planning competition[15].

- (LOGISTICS from PDDL4J) In this domain a set of trucks, airplanes, airports are located at different cities. The goal is to find a plan of transportation to send passengers/goods to desired destinations.
- (BARMAN from IPC'11) The following description is from the IPC'11 website: *In this domain there is a robot barman that manipulates drink dispensers, glasses and a shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks. In this domain deletes of actions encode relevant knowledge given that robot hands can only grasp one object at a time and given that glasses need to be empty and clean to be filled.*
- (VISIT-ALL from IPC'11) In this domain, a robot tries to traverse through all terrains, where connections between terrains are set as constraints.
- (GRIPPER from PDDL4J) In this domain, a robot with a specified number of arms is located in a room. The robot contains several predefined actions such as `pick-object`, `place-object`, or `move`. The goal is distribute objects to the their destinations.

[13] It won the *first* place in the 2008 international planning competition (sequential optimization track). Also, it uses CUDD [1] for symbolic manipulation.

[14] FastDownward is now used by many teams as the initiative for attending the planning competition. It includes many algorithms prebuilt.

[15] Examples are directly taken from the repository. We remove the cost function in the Barman example, so all planners work on actions with unit cost.

Our implementation (Java-based, using JDD as our BDD package) is comparable with other implementations despite the implemented language. On larger examples, GAVS+ outperforms on BARMAN and VISIT-ALL domains, while results vary on the LOGISTICS and GRIPPER domain[16]. We are unable to win against all examples, partly because for some benchmarks, the effect of binary-encoding is not strong enough. Notice that using SAT or SMT solvers rather than BDD may achieve better performance. Nevertheless, our purpose is to have an idea concerning the compactness of our encoding for the use of game solving, and we view this result only as an additional benefit.

B.3 Game Solving (A): the FESTO MPS System

In this section, we describe our demonstrator from FESTO MPS (Modular Production System) for control automation. MPS is mostly used for teaching purposes, but is a very good approximation for a real industrial automation process. Each unit of the MPS processes small colored work pieces that are made out of plastic or metal. Our demonstrator setup (see Figure 5) is composed of two modules, a processing unit and a storage unit.

- The processing unit is built up from a rotating plate with 6 locations, a height probe sensor (which tests the shape of work pieces) and a drilling module (which processes the work pieces).
- The storage unit contains a robot arm used to store and retrieve the object to/from a three-layer rack with six storage locations each.
- Both units are connected over two unidirectional conveyor belts that can deliver one work piece at a time.
- Several levers/rods are allocated on certain positions to move the object between the conveyor belt and the units.

MPS units are usually controlled by PLCs that are directly attached to the units. However, in our demonstrator setup, a global control was needed to execute the generated plan. For this purpose, we wrote a program that interfaces with the FESTO FEC PLCs over Ethernet (UDP/IP) and modified the PLC programs so that every atomic action of the plan can be triggered individually. Since the Siemens LOGO! conveyor belt controllers are attached separately and do not have an appropriate communication interface, we furthermore added a microcontroller that translates incoming requests from the global control 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.

Finally, we designed an automatic script-based process that translates the solution generated from GAVS+ to a C++ program that can be executed on a PC or laptop and acts as global control. The program is linked against a library that implements sending the control codes for each of the atomic operations to the PLCs or the microcontroller. This approach can hence be seen as an interpreter for the solutions generated by GAVS+.

We have created an abstract system configuration (see Figure 8 for components and topology), then tried to model components of each module (together with their predefined actions) using our game extension of PDDL. Table 7 and 8 summarize some predicates and parameterized actions used to specify the system. We then try to specify various specifications and synthesize controllers (in sequentialized-reactive form). Results are in Table 2 (see main text).

1. Our simplest setting is to move an object from position `P1` to the rack. In this example we modify the domain to disable all sensor actions, so it amounts to forward reachability analysis. As the belt moving is only unidirectional (e.g., from `P1` to `P5`), the generated action sequence has 14 steps consisting (a) belt-moving, (b) plate-rotating, and (c) robot-arm processing. The processing time is 1.216 seconds.

[16]Here we list two results: one with FORCE heuristic and the other without; we have an implementation where we tune our heuristic scheme to invoke FORCE when appropriate.

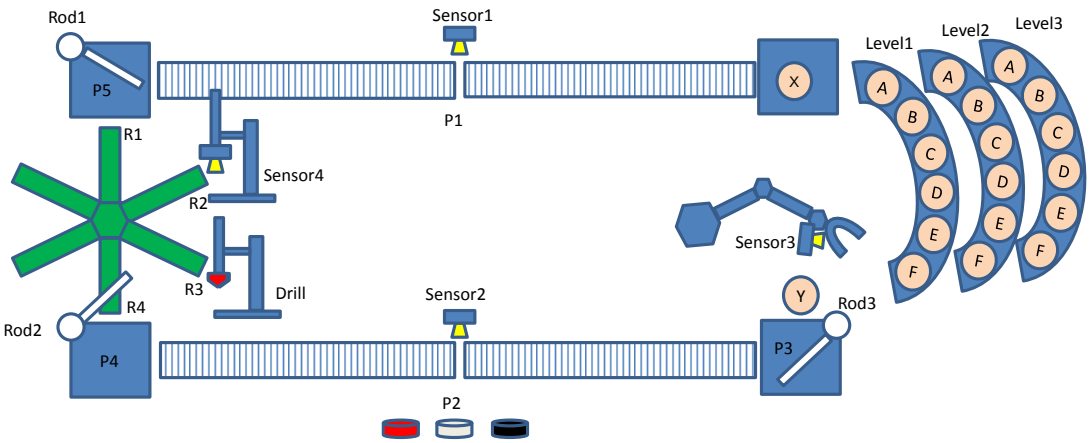

Figure 8: The FESTO MPS demonstrator setup in abstract form.

Table 7: Predicates defined in the PDDL domain of a FESTO MPS system

Predicate	Parameter - Type	Intuitive meaning
in-robot	?pos - robotposition	robot arm is in ?pos
at	?obj - object ?pos - (robotposition, beltposition, plateposition)	object ?obj is at position ?pos
free-hand	?gri - gripper	gripper ?gri has no object
carry	?obj - object, ?gri - gripper	gripper ?gri carries object ?obj
belt-connected	?pos1, ?pos2 - beltposition	transmission belt connected between ?pos1 and ?pos2
rod-located	?pos1, ?pos2 - (robotposition, beltposition, plateposition)	rod can push from ?pos1 to ?pos2
next	?pos1, ?pos2 - plateposition	an object on the plate will move from ?pos1 to the next position ?pos2 when rotating clockwise
drill-position	?pos - plateposition	drill is at position ?pos
have-color-sensor	?pos - (robotposition, beltposition, plateposition)	color sensor is at ?pos
color-sensor-on	?pos - (robotposition, beltposition, plateposition)	color sensor at ?pos is on
color	?obj - object, ?col - colortype	object ?obj has color ?col
have-form-sensor	?pos - (robotposition, beltposition, plateposition)	form sensor is at ?pos
form-sensor-on	?pos - (robotposition, beltposition, plateposition)	form sensor at ?pos is on
form	?obj - object, ?form - formtype	object ?obj has shape ?form
P0TRAN	none	system or environment's move

2. Our second setting is to drill an object and store it on the rack based on the color. Initially the color value is unknown, and shall only be known when triggering the color sensor. Thus in our goal specification, we add an additional constraint specifying that an object shall be of color white, red, or black. Our engine creates sequentialized reactive code-blocks in 30.417 seconds.

3. Our third setting is to modify from 2, and it is required that an object shall only be drilled when it is facing up (formtype: up). When it is placed down, then it shall be returned to position `P1` back to the operator. Currently the engine run out of time (> 500 seconds) when executing such setting. Nevertheless,

   (a) When removing one color and one layer of storage, the engine synthesizes the code in 29.932 seconds.

   (b) When restricting each layer to only two storage positions, the engine synthesizes the code in 86.341 seconds.

   (c) When restricting each layer to only three storage positions, the engine synthesizes the code in 97.473 seconds.

Table 8: Actions defined in the PDDL domain of a FESTO MPS system

Action	Parameter - Type	Intuitive meaning	Change
robot-move	?pos1 ?pos2 - robotposition	robot arm moves from ?pos1 to ?pos2	Sys ⇒ Sys
robot-pick	?gri - gripper ?pos - robotposition, ?obj - object	robot arm ?gri picks object ?obj (both at position ?pos)	Sys ⇒ Sys
robot-check	?pos - robotposition	trigger sensor to check if ?pos is occupied	Sys ⇒ Env
return-check	?pos - robotposition	return whether position ?pos is occupied	Env ⇒ Sys
robot-drop	?gri - gripper ?pos - robotposition, ?obj - object	robot arm ?gri drops object ?obj (both at position ?pos)	Sys ⇒ Sys
belt-move	?obj - object, ?pos1 ?pos2 - beltposition	belt transfers object ?obj from ?pos1 to ?pos2	Sys ⇒ Sys
drill-in	?obj - object, ?pos - plateposition	drill located at position ?pos drills the object	Sys ⇒ Sys
plate-rotate	?obj1 ... ?obj6 - object ?pos1 ... ?pos6 - ?plateposition	rotate the plate to move the object	Sys ⇒ Sys
rod-push	?obj - object ?pos1 ?pos2 - (robotposition, beltposition, plateposition)	push the rod located between ?pos1 and ?pos2 to change the position of object ?obj	Sys ⇒ Sys
trigger-color-sensor	?obj - object ?pos - (robotposition, beltposition, plateposition)	trigger the color sensor located at ?pos	Sys ⇒ Env
return-color-value	?color - colortype ?obj - object ?pos - (robotposition, beltposition, plateposition)	return the color value ?color when the color sensor located at ?pos is on	Env ⇒ Sys
trigger-form-sensor	?obj - object ?pos - (robotposition, beltposition, plateposition)	trigger the shape-detector located at ?pos	Sys ⇒ Env
return-form-value	?form - formtype ?obj - object ?pos - (robotposition, beltposition, plateposition)	return the shape ?form when the shape-detector located at ?pos is on	Env ⇒ Sys

4. The last setting is to modify from 3, and use the object detection sensor to detect whether the rack is full. If it is full, then return object to position `P1` back to the operator.

   (a) When removing one color and one layer while restricting each remaining layer to only two storage positions, the engine synthesizes the code in 231.214 seconds.

   (b) When removing one color and one layer while restricting each remaining layer to only three storage positions, the engine synthesizes the code in 420.061 seconds.

   (c) When restricting each layer to only three storage positions while maintaining three colors, the engine runs out of time (> 500 seconds).

B.4 Game Solving (B): Other Examples

Table 9 summarizes the performance of our synthesis engine on other selected benchmark suites.

- (ROBOTPLANNING modified from MBP [4]) In this domain, a robot is placed in a house with many rooms. The goal varies from simple reachability to repeatedly visit several rooms (generalized Büchi). We also experiment coordination within two robots: the goal is that two robots shall never be in the same room, and in our synthesis framework, we let one robot perform its move freely while the other shall win the safety game.

- (GRIPPER from PDDL4J) In this domain, we model the error of the robot arm, and the purpose is to achieve goal-oriented behavior.

- (ELEVATOR from [23]) In this domain, the request of the user is modeled, and the goal is a GR(1) specification indicating that request shall be responded. See the original paper [23] for details.

- (MODELTRAIN) Lastly, we try to extract from the example shown in Figure 9 a PDDL model for a train system. In this setting, we model the controller with the ability to raise the red light (enforce the train to stop) and to perform change over switches.

  – To create precise modeling over interactions between the train and the track, we have performed case split to partition the set of all tracks into three categories, i.e., in Figure 9 the yellow

Table 9: Experimental results (seconds)

Problem	Execution Time^R	Remark
MBP pb1	0.401	reachability
MBP pb1	1.4	Büchi
MBP pb1	0.399	GR(1)
MBP pb2	6.45	Safety
MBP pb3	7.435	Safety
Gripper pb1	1.48	reachability
Gripper pb2	2.785	reachability
Gripper pb3	6.529	reachability
Gripper pb4	17.963	reachability
Elevator pb1	6.421	GR(1), 8 floors
Elevator pb2	15.557	GR(1), 16 floors
Elevator pb3	31.67	GR(1), 24 floors
Train pb1	1.163	Simple derail prevention (safety)
Train pb2	15.538	Repeated visit (generalized Büchi)
Train pb3	16.616	Collision avoidance (reachability+safety)

[R] Total time includes game creation, game solving, and strategy print-out. No optimization (e.g., variable ordering, sequentialize strategy) is used.

rectangle (`seg`), the red circle (`splitsegP1`), and the mesh rectangle (`splitsegP2`). For example, when a train is on `splitsegP2` moving towards `splitsegP1`, it suffers from derailing when two segments are not connected by switch and when no light signal is on to stop the train.

– The goal is either to perform repeated traversal, to avoid derailing, or to avoid collision (i.e., two trains are never within the same track during their service to destinations).

– We have also experimented our variable reduction techniques under the extension of conditional effects. In this domain, this enables to further reduce the number of declared variables by 30%.

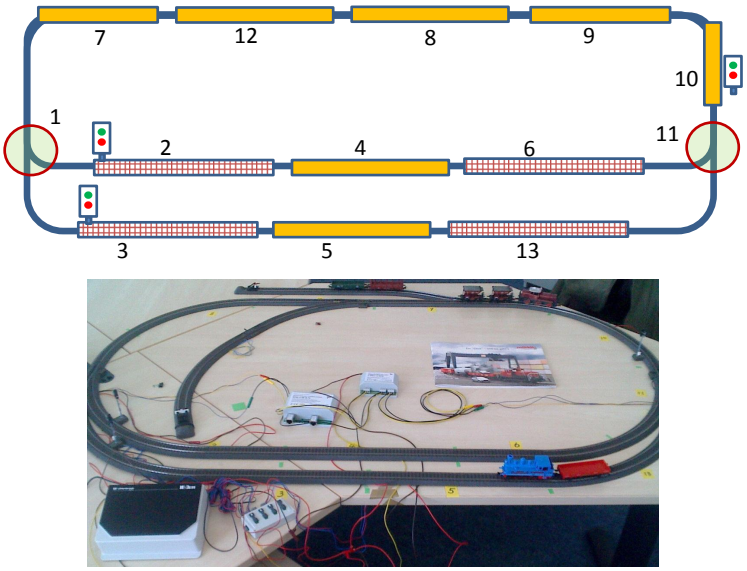

Figure 9: The model train system setup (up) and screenshot of its actual execution (down).