Synthesis of Cost-optimized Controllers from Scenario-based GR(1) Specifications [1]

Daniel Gritzner[2] Joel Greenyer[2]

Abstract: Modern systems often consist of many software-controlled components which must cooperate to fulfill difficult to achieve goals. Trying to reduce the cost of running such a system, e.g., by minimizing total energy consumption, adds additional complexity. To support engineers in the difficult design of such systems we developed a scenario-based specification approach enabling the intuitive modeling of goals and assumptions using short scenarios. These formal specifications allow defects to be detected and fixed early in development. In this paper we present and evaluate an extension to our approach which enables engineers to model costs of processes and thus to synthesize controllers which guarantee that the specified goals are fulfilled in a cost-optimized manner. Our approach even considers the transfer of energy between components to enable the design of systems in which, e.g., the braking energy of moving components can be leveraged to reduce the cost of running a system.

1 Introduction

Modern systems in several domains, e.g., manufacturing, transportation, or health care, often consist of many software-controlled components which must cooperate to fulfill their goals. As users of these systems demand increasingly complex functionality, the processes which need to be performed by the cooperating components become increasingly complex as well. This in turn means engineers face difficult challenges when designing and implementing the behavior required of each component in the system. Processes require the cooperation of multiple components and every components is involved in the implementation of multiple processes, often several processes at the same time. Each component must properly react to external events, e.g., user inputs, as well as internal events, i.e., actions of other components. Determining and implementing the correct behavior for every component is a difficult and error-prone task. This task becomes even more difficult when optimization is a concern, i.e., finding behavior which not only fulfills all requirements but is also optimal according to some well-defined criterion. Typical criteria are reducing the cost of operating a system or reducing its environmental impact, e.g., through reducing the amount of energy or raw materials required for the production of a physical item. A good example of such a system is an automated manufacturing facility in which many robots cooperate to produce cars.

[1] This research is funded by the DFG project EffiSynth.

[2] Leibniz Universität Hannover, Fachgebiet Software Engineering, Welfengarten 1, D-30167 Hannover, Germany daniel.gritzner|greenyer@inf.uni-hannover.de

To support engineers in the difficult design process of such systems we developed a formal, yet still intuitive scenario-based specification approach. In our approach, short scenarios are used to model *guarantees* (goals, requirements, desired behavior) and *assumptions* made about the environment. Scenarios are sequences of events, similar to how engineers describe requirements to each other, e.g., *"When A and B happen, then component C_1 must do D, followed by C_2 doing E."* These sequences are used to describe, in an intuitive way [Al14, GMMS12], when events or actions may, must, or must not occur. The formal nature of scenario-based specifications enables the use of powerful analysis techniques early in the design process. Through simulation and controller synthesis, which, if successful, can prove that the requirements defined in the specification are consistent, defects can be found and fixed early during development. Additionally, these same techniques can be used to directly execute a specification at runtime or to automatically generate executable code [Gr15, GG]. Doing so significantly reduces manual implementation effort, mitigating some of the cost of writing a formal specification and potentially even reducing overall development costs.

In this paper we present and evaluate an extension of our scenario-based specification approach enabling engineers to design systems which behave in a cost-optimized manner. Previously, when synthesizing a controller/strategy, our goal was to find a behavior strategy which allows the system to fulfill all guarantees infinitely often as long as all environment assumptions hold. The extension discussed in this paper enables two things, a) the modeling of costs associated with the behavior defined in a formal specification, and b) the synthesis of controllers which minimize the costs of executing the specified system. In some domains, e.g., manufacturing, negative costs occur: when robots decelerate, their kinetic energy can, through smart electrical design, be leveraged to drive concurrent processes [Gr14]. Usually, this energy is turned into heat and thus is lost to the system. Our proposed extension supports the transfer of energy between processes in order to find a system behavior in which positive and negative costs significantly overlap. Thus the overall energy consumption when running a system can be reduced, i.e., the system's operational costs are reduced. The work in this paper is part of a realization of our proposed vision of generating code for the energy-efficient control of production systems [GG16].

Sect. 2 of this paper introduces a running example. Sect. 3 explains some preliminaries. Then, Sect. 4 discusses the main contribution, cost-optimized synthesis, followed by an evaluation in Sect. 5. The paper finishes with related work and a conclusion in Sect. 6 and 7.

2 Example

As a running example we use a production system as depicted in Fig. 1. While the figure only shows two robot arms, one welding and the other performing a pick-and-place operation, this example can easily be extended to additional robots which perform tasks such as drilling or cutting.

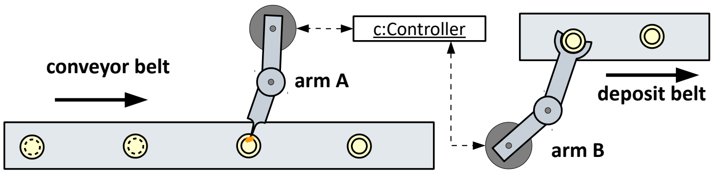

Fig. 1: A typical production system consisting of two robot arms and two conveyor belts. The first arm welds two pieces of a work item firmly together while the second arm picks up the finished items and places them on a different conveyor belt.

This is a typical production process and can be found in many domains, e.g., the manufacturing of cars. Partially built cars move along a conveyor system and once they are in place for the next step, robot arms move in and perform some action, e.g. tightening screws, welding, etc., on the car. Then the robots move back into a holding position and prepare themselves for the next car. These actions are then repeated again and again.

The optimization goal in our example is to find a strategy for controlling the robots such that their movement becomes synchronized in a way that minimizes the overall energy consumption of the system. However, the robots must still fulfill all required tasks. For this purpose, we include positive costs for acceleration and movement at constant velocity for each arm (these must be minimized) and negative costs for deceleration in our specification. Braking energy that cannot be used immediately is turned into heat, i.e., it is lost.

3 Preliminaries

In this section we explain two-player games with a General Reactivity of rank 1 (GR(1)) condition, a formalism to which we map our formal, scenario-based specification. The two players are the specified system and its environment. To synthesize a controller from a formal specification, we try to find a strategy for the system in a GR(1) game such that the system is able to fulfill all guarantees as long as the environment fulfills the assumptions.

3.1 GR(1) Games

A *game* is a directed graph, called *game graph*, $GG = (V, E, v_0)$ with a finite set of vertices V, a set of edges $E \subseteq V \times V$, and an initial state $v_0 \in V$. The set V is partitioned into two disjoint sets V_1 and V_2. Each vertex $v \in V_i$ is controlled by player i, i.e., this player can choose which edge $e = (v, v') \in E$ with $v' \in V$ is traversed to progress the game. Since the players in our case are the system and its environment, we also use V_{sys} and V_{env} instead of V_1 and V_2 respectively. E is left-total, i.e., there are no dead ends in GG. A *play* or *run* is an infinite sequence of vertices and edges $r = v_0 e_0 v_1 e_1 v_2 e_2 v_3 \ldots$ such that $\forall i : e_i = (v_i, v_{i+1}) \in E$.

A *GR(1) game* is a game with an additional winning condition for player sys. To help understanding a GR(1) condition, we start with the simpler *Büchi* condition. A Büchi condition consists of a set of goal states $G \subseteq V$ at least one of which must be visited infinitely often by any run which fulfills this condition. The equivalent Linear Temporal Logic (LTL) [Pn77] formula is $\Box\Diamond g$ with $g =$ "a state $v \in G$ is visited". This condition can be generalized into a *Generalized Büchi* condition with multiple, but finitely many, goal state sets G_i, all of which must be visited independently, i.e., $\bigwedge_i \Box\Diamond g_i$ in LTL. A GR(1) condition consists of two Generalized Büchi conditions, the *assumptions* $A = (A_1, A_2, \dots, A_n)$ and the *guarantees* $G = (G_1, G_2, \dots, G_m)$ with $n, m \in \mathbb{N}$ such that the formula

$$\left(\bigwedge_i \Box\Diamond a_i\right) \implies \left(\bigwedge_j \Box\Diamond g_j\right) \tag{1}$$

holds. This condition is fulfilled iff, whenever all the assumptions hold, i.e., their goal states are visited infinitely often, all the guarantees hold as well.

A *strategy* for player i is a mapping from any possible finite prefix $p = v_0 \dots v_j$ of any possible run r, with v_j controllable by i, to an edge $e = (v_j, v') \in E$. A strategy is *memoryless* if the mapping to edges e only depends on the current state v_j of a game. Player sys wins a run r iff r fulfills the GR(1) condition of the GR(1) game. Player sys wins a GR(1) game iff that player has a strategy such that sys wins every possible run r.

3.2 Scenario-based Modeling

We developed a DSL, the *Scenario Modeling Language* (SML), and an associated tool suite, ScenarioTools, for creating and analyzing formal, scenario-based specifications [Gr15, Gr16, GG]. SML is a textual variant of Live Sequence Charts [DH01, HM03a] which we extended with additional flow control features. Listing 1 shows an excerpt of a specification written in SML. In SML, short scenarios describe in an intuitive way how objects may, must, or must not behave. The excerpt shows an assumption of how robot arm movement works: when the robot is instructed to move, it will immediately accelerate, after that it will eventually move at a constant velocity, eventually decelerate and eventually arrive at its destination. At every step it notifies the system's controller of its current state. *Roles* (lines 2&3) in SML serve the same purpose as lifelines in Message Sequence Charts. Different objects can be bound at runtime to *dynamic roles*, even concurrently. As an example, if two robots move concurrently there will be two instances of RobotMovement (lines 4-10) active concurrently with each instance having different, unique role bindings. Classes of objects can be defined as controllable by the system (line 1). All other objects are uncontrollable, or controllable by the environment, by default.

An SML specification consists of sets of scenarios and the play out algorithm [HM03a, HM03b] defines how they can be interwoven into a play. Basically, the algorithm waits for

```
controllable { Controller }
dynamic role Controller controller
dynamic role Robot robot
assumption scenario RobotMovement {
    controller -> robot.move()
    committed robot -> controller.accelerate()
    eventually robot -> controller.move()
    eventually robot -> controller.decelerate()
    eventually robot -> controller.arrived()
}
```

List. 1: Excerpt from an SML specification of the example shown in Fig. 1

the environment to choose an event and then activates and progresses scenarios accordingly. Whenever at least one event, sent by a system object, is enabled which is flagged by a liveness condition (e.g., committed), play out will execute one of these events, unless it is blocked by another scenario.

The play out algorithm is generally non-deterministic meaning that multiple events may be possible according to the algorithm given a specific set of active scenarios and an object instance model. This induces a state space, in particular a game graph $GG = (V, E, v_0)$. Each vertex represents a system state, i.e., a set of active scenarios and an object model. The initial vertex has no active scenarios but still has an object model. Each edge is labeled with an event that corresponds to the event that caused the changes in the system state between the two connected vertices. Each vertex is controllable by either the system or the environment, i.e., all outgoing edges, also called outgoing transitions, correspond to events in which the senders are all controllable by the same player. To implement the keyword eventually properly, system controlled vertices might have a system-controllable outgoing edge explicitly representing “wait for the environment”.

To turn this game into a GR(1) game, we also extract a GR(1) condition from the specification. Each active scenario with unique role bindings is turned into one assumption or guarantee condition depending on the scenario’s type (assumption or guarantee). The goal states for such an active scenario are all states in which this scenario has no liveness condition to fulfill (in particular, this is true if the scenario is not even active in a given state) and no safety violation of that scenario (tracked via a flag) has occurred in previous states. Additionally, we add a guarantee that contains all environment-controlled states as goal states. This is to ensure that the system will eventually react to external inputs again, assuming the specification is well-separated, which is a desirable property of a good specification [MR16]. In a well-separated specification, the system cannot force a violation of the assumptions by any of its choices, i.e., the system must focus on fulfilling the guarantees.

Active scenarios AS represent subgraphs of this game graph. These subgraphs are characterized by initializing edges, terminating edges, and a set of states $V' \subset V$ in which AS is active. *Initializing edges* represent events which activate AS, i.e., which represent the first event in the associated scenario S. *Terminating edges* represent events which terminate AS, usually the last event in S. The smallest possible such subgraph is just a single edge which is both initializing and terminating an active scenario which merely checks a condition when a certain event occurs. However, for simplicity, we will later only consider subgraphs in which the sets of initializing and terminating edges are disjoint, implying $V' \neq \emptyset$. These subgraphs represent *activities* which last for an extended duration and are characterized by events which indicate the beginning, progress, or end of an activity. The scenario RobotMovement in Listing 1 is an example of such an activity. The set V' of an activity can be partitioned into a set of weakly connected components with exactly one component for each instance of the activity within the game graph.

3.3 Controller Synthesis

Our controller synthesis is an implementation of Chatterjee's attractor-based GR(1) game solving algorithm [Ch16]. It uses the assumptions' and guarantees' goal states to calculate *attractors*. An attractor is dependent on a player and a condition, e.g., a system attractor of guarantee G_j is a state from which the system can ensure to visit a goal state of G_j regardless of the environment's behavior. Chatterjee's algorithm iteratively calculates and removes environment dominions from the game graph. An environment dominion is a set of states all of which are not a system attractor of at least one guarantee. Furthermore, these states are all environment attractors of all assumptions. The states retained after no further environment dominion can be found are known to contain a strategy which allows the system to either fulfill all of its guarantees or to ensure the violation of at least one assumption. If the initial state of the game graph is in this set, the specification is *realizable*, i.e., the system wins the game. We call the set of retained states the *winning states*.

To extract a strategy from the winning states we start with calculating system attractors strategies for all guarantees. This is done via a reverse search starting from the goal states of a guarantee. First, all goal states are marked as attractors. Then, iteratively, all non-attractor states which are directly reachable via traversing an edge from an attractor in reverse direction, are tested whether they are attractors or not, i.e., are they system controllable with an edge leading to an attractor or are they uncontrollable with all edges leading to attractors. All new attractors found this way are also marked as attractors and for each new attractor that is controllable, the edge used to reach it (via reverse direction traversal) is stored as the system's move in that particular state. These mappings of states to edges are a cycle-free strategy to reach a goal state, i.e., this mapping is a system attractor strategy.

Next, for each guarantee, we calculate system dominions, analogously to the environment dominions used in the initial game solving algorithm, in the non-attractor states of that guarantee. We store the strategy to reach and stay in such a dominion as we did before for fulfilling the guarantee. Then we remove it and repeat the process until no further dominion can be found and removed. This way we end up with a strategy for each guarantee to either fulfill it or end up in a subgraph in which at least one assumption is violated.

Finally, a memoryless strategy can be calculated by creating a new game graph which consists of m copies of the original graph's vertices (m = number of guarantees). The states in this new graph are labeled with the label of the state they are a copy of and the label of the guarantee G_j which should be fulfilled next. An arbitrary state, which is a copy of the original initial state, can be chosen as the new initial state. From this state the strategy for G_j is followed (all outgoing edges are added for uncontrollable states) until a goal state is reached. The outgoing edges of this goal state are labeled with a new $G_{j'}$ to fulfill and its strategy is then followed. One after another, all guarantees are fulfilled this way. Most of the states created by this approach will never be reached, thus creating the necessary states on-the-fly while following the current strategy greatly reduces the resulting graph's size. Memoryless strategies make subsequent processes, e.g., code generation, easier as only the system state needs to be tracked at runtime as opposed to the entire event history.

4 Cost-optimized Synthesis

In this section we describe our technique for synthesizing a cost-optimized controller. It is built on top of the GR(1) game solving algorithm described in the previous section, reusing as much of the existing functionality as possible. We followed a divide-and-conquer approach in our design. First, we execute the regular attractor-based game solving algorithm. If the system is realizable, we run our cost optimization algorithm on the winning states. Finally, we perform a modified strategy extraction to extract a cost-optimized memoryless strategy from the winning states, or in other words, synthesize a cost-optimized controller.

4.1 Optimization Goals, Costs and Gains

GR(1) games are infinite games, i.e., the systems represented by these games are intended to be run for an indefinite amount of time. While this is a good analogy for how systems in many domains, e.g., manufacturing, are actually run, this poses a problem for optimization: if every event or activity (cf. end of Sect. 3.2) can, in theory, occur an infinite number of times, how do we model costs in a well-defined, preferably finite, way that can be computed with reasonable effort? We observed that components repeat finite activities an infinite number of times in these games. An analogous pattern in a programming language would be a while-loop whose loop condition is always true and which, of course, has a finite body. Often even the whole system follows such a pattern. Thus, to optimize an infinite game, we need to identify its finite "body", which is a subgraph with a clearly defined beginning and end like that of an activity, and optimize the behavior while in that body.

To identify finite subgraphs during which the system's behavior shall be optimized we added the ability to annotate scenarios in SML as *optimization goals*. The activity represented by the scenario then defines a set of weakly connected components during which optimization shall be performed. A specification may already contain suitable scenarios which merely must be annotated by an expert. But even if not, SML allows writing appropriate scenarios which do not alter the game graph or the GR(1) condition induced by the specification. These can be used to define optimization goals in a clear, separate manner. Through annotations engineers can also choose which parts of the system's behavior are optimized while others, which need not be optimized, e.g., a one time initialization, or which must not be optimized, e.g., a safe shutdown procedure in case of an error or emergency, remain untouched.

We use the same notion of activities we used for defining optimization goals to define *costs* (scenarios annotated with a cost $\in \mathbb{R}^+$) and *gains* (scenarios annotated with a cost $\in \mathbb{R}^-$). By using activities instead of merely assigning cost values to vertices or edges we are able to identify when costs and gains overlap to model effects like the transfer of energy between components mentioned in the introduction. Costs may model whichever property shall be optimized such as the usage of energy, raw materials, or money. Concrete cost values have to be provided by engineers at design time. Measuring or estimating these costs is outside the scope of this paper as it depends on the domain of the system and the type of the cost.

Without loss of generality, for the remainder of this paper we only consider optimization goals, costs, and gains which consist of a single weakly connected component. In practice, we simply split subgraphs consisting of several such components into multiple subgraphs. For optimization goals, we further assume that they do not overlap. Optimization goal instances which, after splitting into single weakly connected components, overlap are merged into one single instance of an optimization goal. This opens up a new issue: circular dependencies of goals. If, e.g., two activities which shall both be optimized overlap which each other such that the end of the first activity overlaps with the beginning of the second activity and vice versa, a simple trick can resolve this issue. The circle becomes a finite subgraph if only one of the two activities is defined as optimization goal. If the impact of the activity, which is not optimized, is sufficiently small or the overlap of the activities is sufficiently large, then the effect on the resulting optimized behavior can be negligible. Assuming the robot arms in Fig. 1 work in such a timing that one arm always starts processing the next work item while the other just finishes its work on its current work item, then only optimizing the movement of one arm yields good results. It will synchronize itself such to the movement of the other arm that the overall system behaves in an optimal way.

4.2 Optimization Objective

Our optimization objective is to find optimal paths through every optimization goal OG of a game. For every system-controllable $v \in V_{OG}$ we want to find the optimal edge to choose for the player sys such that the path $p = ve_0v_1e_1 \dots e_n$ with e_n being a terminating transition incurs the least possible cost. We optimize the worst-case cost, i.e., player env will always choose the edge that leads to the highest overall cost.

The cost of a path $p = ve_0v_1e_1 \dots e_n$ is defined as

$$\text{cost}(p) = \left(\sum_{i=0}^{n} \text{cost}(e_i) \right) + \text{cost}(\text{targetState}(e_n)) \tag{2}$$

with the cost of an edge e_i being the sum of all costs of activities which terminate in e_i and the cost of a state v' being the sum of all costs of activities active in v'. The costs of the target state v' of e_n are included, despite OG already being terminated, because following the path p makes performing the activities active in v' unavoidable. However, to take effects such as transfer of energy into account, we consider the effective cost of a path p, which is

$$\text{effectiveCost}(p) = \text{cost}(p) - \text{optimalGain}(p). \tag{3}$$

The optimal gain is the maximum amount of cost which can be mitigated by gains. Assuming we have a function costTransfer : Costs $\times$ Gains $\rightarrow \mathbb{R}_0^+$ which maps pairs of costs $c \in$ Costs and gains $g \in$ Gains to the amount of c mitigated by g, the optimal gain is defined as

$$\text{optimalGain}(p) = \max_{\text{costTransfer}} \left(\sum_{c \in \text{Costs}(p), g \in \text{Gains}(p)} \text{costTransfer}(c, g) \right), \tag{4}$$

i.e., it is the result of applying the best possible cost transfer function to all relevant cost and gain pairs. A cost, and analogously gain, is only relevant to a path p iff it affects cost(p), i.e., it overlaps p. While every path p may have a different optimal cost transfer function, all such functions must fulfill the following constraints:

- $\forall c \in$ Costs, $g \in$ Gains : $\left(V_c \cap V_g = \emptyset\right) \Rightarrow \text{costTransfer}(c, g) = 0$, with V_c, V_g being states in which c, g are active, i.e., gains can only mitigate concurrently active costs.
- $\forall c \in$ Costs : $\left(\sum_{g \in \text{Gains}} \text{costTransfer}(c, g)\right) \leqslant \text{cost}(c)$, with cost$(c)$ being the cost value assigned to c. This constraint ensures that no cost is "overcompensated".
- $\forall g \in$ Gains : $(\sum_{c \in \text{Costs}} \text{costTransfer}(c, g)) \leqslant |\text{cost}(g)|$, i.e., no gain can be "overused" to mitigate concurrent costs.

A single cost can be mitigated by multiple gains and a single gain can mitigate multiple costs as long as the constraints above are fulfilled. Gains, which do not overlap with sufficiently high costs, are (partially) lost.

4.3 Cost Propagation

Algorithm 1 shows how we apply dynamic programming and depth-first search to build a map M of paths, which minimize the effective cost, through an optimization goal OG.

Algorithm 1 Cost propagation for optimization goal OG

```
Input: terminating transitions $E_T$ of $OG$
Output: map $M$ of cost-optimized paths
 1: initialize empty map $M$, stack $S$, set $Done$, and set $Improved$
 2: $S$.pushAll($e \in E_T$)
 3: while $S$ is not empty do
 4:     $e \leftarrow S$.pop(), $ss \leftarrow e$.sourceState, $ts \leftarrow e$.targetState
 5:     if $M$.contaisKey($ts$) $\wedge$ $ts$ is system-controllable then
 6:         $p \leftarrow M[ts]$.first + $e$
 7:     else if $M$.contaisKey($ts$) $\wedge$ $ts$ is environment-controllable then
 8:         $p \leftarrow M[ts]$.last + $e$
 9:     else
10:         $p \leftarrow e$
11:     effCost $\leftarrow$ calculateEffectiveCost($p$)
12:     if $M[ss]$ = null then
13:         $M[ss] \leftarrow \{p\}$, $Improved \leftarrow Improved \cup \{ss\}$
14:     else
15:         $M[ss]$.addOrdered($p$)
16:         if mappingImproved($M$, $ss$, $p$) then
17:             $Improved \leftarrow Improved \cup \{ss\}$
18:     $Done \leftarrow Done \cup \{e\}$
19:     pushImprovedTransitions($S$, $ss$, $Done$, $Improved$)
20: return $M$
```

The algorithm starts at the terminating transitions of OG and searches optimal paths in the reverse direction. Calculating the effective cost of paths in that order is easier than in a forward search.

For every edge e, the algorithm takes the best path p from e's target state to the end of OG (stored in M), adds e to the front of p (lines 5-10), calculates the effective cost of the new p (line 11) and then finally updates M (lines 12-17), which will eventually store one entry for every outgoing transition of every state in OG. The mappings in M are stored in order from least effective cost to highest effective cost. Mappings with equal cost are stored in the order they are found, as this will cause strategy extraction to favor edges which do not lead to cycles within OG.

To calculate the effective cost (line 11), we maintain an additional map of paths $p' = v'e_0v_1e_1v_2e_2 \ldots e_n$ to a data structure storing the path's (non-effective) cost, which gain beginning in e_0 or later compensates which cost and the degree of cost transfer, and how much cost transfer from gains active in v' could potentially currently occur. From this information the effective cost of p can be easily computed.

To calculate the cost of $p = vev'e_0v_1e_1 \ldots e_n$ (concatenation of e and p'), we add the values of costs terminating in e to the cost of p', determine the optimal cost transfer from gains beginning in e (thus turning potential cost transfer into actual cost transfer), and calculate the new potential cost transfer of gains active in v. When turning potential cost transfers to actual cost transfers we approximate the optimal cost transfer function via a heuristic to avoid expensive combinatorial explosion. Our heuristic is based on the number numOverlaps(g) of costs a gain g could potentially compensate. The idea is to use gains with less opportunities to compensate costs first. When multiple gains begin in the same edge, we process them in the order of their numOverlaps(g) from lowest to highest. Similarly, during the processing of each gain we iterate over costs c based on numOverlaps(c). While doing so, we make sure that the constraints defined in Sect. 4.2 are still satisfied. Computing the potential cost transfer from still active gains follows the same approach.

Algorithm 1 uses the sets $Improved$ and $Done$ to determine which edges to push onto the search stack S (line 19). Whenever S is empty, all incoming edges of states in $Improved$ are pushed onto S and then $Improved$ is cleared. As an optimization, the incoming edges of a state may be pushed onto S if all of the states outgoing edges are in $Done$, even though S is not yet empty. This reduces the number of loop iterations necessary to fully construct M.

The algorithm will eventually terminate because the terminating edges of OG are only pushed onto S once and the incoming edges of every vertex v in OG are pushed onto S at most as often as the number of outgoing edges of v. After termination, M maps every vertex in OG to a list of its outgoing edges ordered from least to highest effective cost.

4.4 Modified Strategy Extraction

Extracting a cost-optimized memoryless strategy works similar to extracting a regular memoryless strategy, though requires some modification to system attractor strategy calculation. Other steps, e.g., merging the strategies for each guarantees into a single strategy, work the same as before. This implies that parts of the game graph, in which the system must ensure that the assumptions are violated, are not optimized. We do not consider this to be a goal of a well-defined system.

The modified system attractor calculation for each guarantee works by iterating over all its attractor vertices v after discarding the system moves determined by the regular system attractor calculation (cf. Sect. 3.3). A forward depth-first search is performed, iterating over outgoing edges of each controllable vertex in the order stored in $M[v]$ (a random order is tried when there is no such $M[v]$). The search starts at an arbitrary controllable attractor vertex v. When a vertex v_{Goal}, which is either a goal state or for which a move is already known, is found, the outgoing transitions used to construct the path from v to v_{Goal} are stored as the systems' moves for each controllable vertex on said path. Then a new search is started from a controllable vertex for which no move has been determined so far. If the search detects a cycle in the current path or encounters a non-attractor state it will backtrack. This way the system will take the most cost-efficient path to fulfill each guarantee. There will be no paths with a lower effective cost that lead to a goal state. The approach terminates as all options are eventually considered, assuming the least cost-effective path is still the best option which still fulfills a given guarantee.

5 Evaluation

In this section we discuss benchmark results of our approach for the synthesis of cost-optimized controllers. We measured the time required to execute the algorithms as well as the size of the game graphs and the synthesized controllers.

5.1 Benchmark Setup

All benchmarks were run on a laptop with an Intel Core i7-5500U, 8 GB RAM, and Windows 10 64-bit. ScenarioTools, the implementation of our DSL and tool-suite, was run using Eclipse Modeling Tools Oxygen and Java 8. During benchmarking no other processes other than system services were running on this system and it was not connected to any network.

We ran benchmarks using the example described in Sect. 2. We used different instances with a varying number of robots for benchmarking. The specification consisted of seven scenarios, one of which was annotated as optimization goal, one was annotated as a cost activity (a robot accelerates and moves) and one was annotated as a gain activity (a robot decelerates; potentially losing kinetic energy as heat). The specified situation was that of new work items, e.g., a car, arriving at each robot's work station simultaneously, followed by each robot moving in from a holding position, performing a task (these were not modeled in detail but kept abstract) and then moving back into the initial holding position waiting for the next work item. The optimization goal was to minimize the loss of kinetic energy as heat during each work item processing cycle, i.e., minimize the total energy cost of the system.

Each example was benchmarked 10 times. Before running the actual benchmarks, five warm up iterations were performed, to prevent the results from becoming tainted by on-demand resource loading or an undefined cache state.

5.2 Benchmark Results

Tables 1 and 2 show our benchmark results. The game graph and controller size comparison shows that only a fraction of the states and transitions from the game graph actually end up being part of the synthesized controller, despite the controller potentially containing multiple copies of each state of the game graph. Also, the state space explosion problem is apparent, as the size of the game graphs increases exponentially with the number of robots. Controllers for instances with five or more robots could not be synthesized as ScenarioTools ran out of memory during game graph creation. Furthermore, there was a single optimization goal instance which spanned nearly the entire game graph.

Table 2 shows that the majority of time was spent creating the game graph, consuming about 95% of the time required to synthesize a controller. The other steps require a comparatively insignificant amount of time, indicating that the algorithms for performing these other steps,

number of robots	**1**	**2**	**3**	**4**
states in game graph	16	279	4010	52877
transitions in game graph	16	379	6310	91477
states in controller	16.0	62.7	1462.7	18954.7
transitions in controller	16.0	67.7	1818.3	26681.3
states in optimization goal	14	275	4002	52861

Tab. 1: Size of the game graphs, synthesized controllers, and the optimization goal subgraph; controller metrics are averaged over 10 iterations.

number of robots	**1**	**2**	**3**	**4**
game graph creation	21.8 ms	326.3 ms	6109.5 ms	117181.8 ms
GR(1) game solving	0.1 ms	1.7 ms	44.5 ms	2193.4 ms
strategy extraction (just GR(1))	0.4 ms	1.5 ms	41.6 ms	1974.1 ms
cost optimization	0.1 ms	1.7 ms	18.8 ms	468.7 ms
strategy extraction (optimized)	1.6 ms	2.7 ms	57.9 ms	2739.1 ms
total (extracting opt. strategy)	23.6 ms	332.4 ms	6230.7 ms	122583.0 ms

Tab. 2: Performance measurements; all measurements are times in milliseconds and averaged over 10 iterations.

including the cost optimization presented in this paper, scale sufficiently well to larger systems. Game graph creating becomes a problem long before these other steps start to run slowly. In particular, the cost optimization, i.e., extracting optimization goals, costs, and gains and propagating effective costs through optimization goals, scales well. For larger systems, it performs significantly better than every other step. Overall, for the instance with four robots, cost optimization and modified strategy extraction added only 1233.7 ms to the process compared to synthesizing a non-optimized controller. This means for that instance the overall synthesis time only increased by 1%.

The modified strategy extraction appears to require significantly more time than the regular strategy extraction for smaller systems (a factor 4 difference for a system with a single robot), but this difference decreases as the size of the system increases (a factor 1.39 difference for a system with four robots). While the effect on the overall synthesis process is already negligible, Table 2 also indicates that the performance difference between the two strategy extraction approaches themselves becomes negligible for sufficiently large game graphs.

6 Related Work

There are many approaches for synthesizing optimized controllers from a formal specification, e.g., by Smith et al. [Sm10], Karaman et al. [KF11, KSF08], Wolff et al. [WTM12], and Jing et al. [JEKG13]. The last one even offers an extensible GR(1) synthesis tool [ER16]. These approaches use temporal logic, usually LTL, for their formal specifications and transition-based costs. Our approach, instead, uses intuitive scenario-based specifications and optimizes based on activities. Activity-based costs are necessary to leverage effects such as the transfer of braking energy to drive concurrently active components.

Also related is the work done on energy games. Introduced by Bouyer et al. [Bo08], there are synthesis approaches proposed by Chatterjee et al. [Ch10, CRR14], Maoz et al. [MPR16], and Brim et al. [Br11]. Actions performed by the players, i.e., transitions traversed in the game graph, cause an energy level to rise and fall throughout the game, thus modeling costs and gains. The first player's objective is to keep this level non-negative. Again, transition-based costs and gains are insufficient for our optimization objective (cf. Sect. 4.2).

Mean pay-off games offer the ability to model vertex-based rewards [CHJ05]. Maximizing rewards can be interpreted as minimizing costs. However, vertex-based costs/rewards have the same limitations as transition-based rewards.

Pellicciari et al. [Pe13] try to utilize idle times of individual robot trajectories. Wigström et al. [Wi13] use dynamic programming to determine an optimal task schedule for a multi robot cell. However, neither approach is able to automatically find optimization opportunities outside of isolated time windows or pre-defined energy exchange opportunities between components. Our approach is able to consider all optimization opportunities.

7 Conclusion

In this paper we presented an approach for synthesizing cost-optimized controllers from scenario-based specifications, an intuitive method for creating formal specifications. We evaluated the performance of this optimization technique and found that it works very fast, especially for larger systems. It is able to handle optimization goals which leverage the transfer of energy between components to reduce the overall energy consumption of a system, a goal difficult or even impossible to model in many existing approaches.

In future work, we want to extend SML with the ability to model time-based constraints and costs. This enables not only the modeling of safety properties in which timing plays an important role but also opens up new avenues for optimization. In robotics, moving a component the same distance at different speeds causes different energy consumptions. To take advantage of this, we need to know how much time a component has to fulfill its goal without slowing down the overall system. Furthermore, knowing how long processes take would also allow for a more accurate model of cost compensation. If we know by how much the deceleration and movement of different components overlap we can more accurately determine how much of the braking energy can actually be recouped.

References

[Al14] Alexandron, G.; Armoni, M.; Gordon, M.; Harel, D.: Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In: Proc. 36th Int. Conf. on

Software Engineering (ICSE). pp. 311–320, 2014.

[Bo08] Bouyer, Patricia; Fahrenberg, Uli; Larsen, Kim G; Markey, Nicolas; Srba, Jiří: Infinite runs in weighted timed automata with energy constraints. In: International Conference on Formal Modeling and Analysis of Timed Systems. Springer, pp. 33–47, 2008.

[Br11] Brim, Lubos; Chaloupka, Jakub; Doyen, Laurent; Gentilini, Raffaella; Raskin, Jean-François: Faster algorithms for mean-payoff games. Formal methods in system design, 38(2):97–118, 2011.

[Ch10] Chatterjee, Krishnendu; Doyen, Laurent; Henzinger, Thomas A; Raskin, Jean-François: Generalized mean-payoff and energy games. arXiv preprint arXiv:1007.1669, 2010.

[Ch16] Chatterjee, Krishnendu; Dvorák, Wolfgang; Henzinger, Monika; Loitzenbauer, Veronika: Conditionally Optimal Algorithms for Generalized Büchi Games. In (Faliszewski, Piotr; Muscholl, Anca; Niedermeier, Rolf, eds): 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016). volume 58 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 25:1–25:15, 2016.

[CHJ05] Chatterjee, Krishnendu; Henzinger, Thomas A; Jurdzinski, Marcin: Mean-payoff parity games. In: Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on. IEEE, pp. 178–187, 2005.

[CRR14] Chatterjee, Krishnendu; Randour, Mickael; Raskin, Jean-François: Strategy synthesis for multi-dimensional quantitative objectives. Acta Informatica, 51(3-4):129–163, 2014.

[DH01] Damm, Werner; Harel, David: LSCs: Breathing Life into Message Sequence Charts. In: Formal Methods in System Design. volume 19, pp. 45–80, 2001.

[ER16] Ehlers, Rüdiger; Raman, Vasumathi: Slugs: Extensible gr (1) synthesis. In: International Conference on Computer Aided Verification. Springer, pp. 333–339, 2016.

[GG] Gritzner, Daniel; Greenyer, Joel: Controller Synthesis and PCL Code Generation from Scenario-based GR (1) Robot Specifications. In: Proceedings of the 4th Workshop on Model-Driven Robot Software Engineering (MORSE 2017), co-located with Software Technologies: Applications and Foundations (STAF 2017) (to appear).

[GG16] Greenyer, Joel; Gritzner, Daniel: An Approach for Synthesizing Energy-Efficient Controllers for Production Systems from Scenario-Based Specifications. In: Proc. of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2016). volume 1725. CEUR Workshop Proceedings, pp. 87–93, 2016.

[GMMS12] Gordon, M.; Marron, A.; Meerbaum-Salant, O.: Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In: Proc. 17th Conf. on Innovation and Technology in Computer Science Education (ITICSE). pp. 198–203, 2012.

[Gr14] Greenyer, Joel; Hansen, Christian; Kotlarski, Jens; Ortmaier, Tobias: Towards Synthesizing Energy-efficient Controllers for Modern Production Systems from Scenario-based Specifications. Procedia Technology (Proceedings of the 2nd International Conference on System-Integrated Intelligence (SysInt 2014)), 15(0):388–397, 2014.

[Gr15] Greenyer, Joel; Gritzner, Daniel; Gutjahr, Timo; Duente, Tim; Dulle, Stefan; Deppe, Falk-David; Glade, Nils; Hilbich, Marius; Koenig, Florian; Luennemann, Jannis; Prenner, Nils; Raetz, Kevin; Schnelle, Thilo; Singer, Martin; Tempelmeier, Nicolas; Voges, Raphael: Scenarios@run.time – Distributed Execution of Specifications on IoT-Connected Robots. In: Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015. CEUR Workshop Proceedings, 2015.

[Gr16] Greenyer, Joel; Gritzner, Daniel; Katz, Guy; Marron, Assaf: Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In: Proc. of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2016). volume 1725. CEUR, pp. 16–32, 2016.

[HM03a] Harel, D.; Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, 2003.

[HM03b] Harel, David; Marelly, Rami: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. SoSyM, 2:82–107, 2003.

[JEKG13] Jing, Gangyuan; Ehlers, Rüdiger; Kress-Gazit, Hadas: Shortcut through an evil door: Optimality of correct-by-construction controllers in adversarial environments. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, pp. 4796–4802, 2013.

[KF11] Karaman, Sertac; Frazzoli, Emilio: Linear temporal logic vehicle routing with applications to multi-UAV mission planning. International Journal of Robust and Nonlinear Control, 21(12):1372–1395, 2011.

[KSF08] Karaman, Sertac; Sanfelice, Ricardo G; Frazzoli, Emilio: Optimal control of mixed logical dynamical systems with linear temporal logic specifications. In: Decision and Control, 2008. CDC 2008. 47th IEEE Conference on. IEEE, pp. 2117–2122, 2008.

[MPR16] Maoz, Shahar; Pistiner, Or; Ringert, Jan Oliver: Symbolic BDD and ADD Algorithms for Energy Games. arXiv preprint arXiv:1611.07622, 2016.

[MR16] Maoz, Shahar; Ringert, Jan Oliver: On well-separation of GR (1) specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 362–372, 2016.

[Pe13] Pellicciari, M; Berselli, G; Leali, F; Vergnano, A: A method for reducing the energy consumption of pick-and-place industrial robots. Mechatronics, 23(3), 2013.

[Pn77] Pnueli, Amir: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. IEEE, pp. 46–57, 1977.

[Sm10] Smith, Stephen L; Tůmová, Jana; Belta, Calin; Rus, Daniela: Optimal path planning under temporal logic constraints. In: Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on. IEEE, pp. 3288–3293, 2010.

[Wi13] Wigstrom, Oskar; Lennartson, Bengt; Vergnano, Alberto; Breitholtz, Claes: High-level scheduling of energy optimal trajectories. IEEE Transactions on Automation Science and Engineering, 10(1):57–64, 2013.

[WTM12] Wolff, Eric M; Topcu, Ufuk; Murray, Richard M: Optimal control with weighted average costs and temporal logic specifications. Proc. of Robotics: Science and Systems VIII, 2012.