Dynamic Update for Synthesized GR(1) Controllers

Gal Amram
Tel Aviv University

Shahar Maoz
Tel Aviv University

Itai Segall
Nokia Bell Labs

Matan Yossef
Tel Aviv University

ABSTRACT

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. GR(1) is an expressive fragment of LTL that enables efficient synthesis and has been recently used in different contexts and application domains. In this paper we investigate the dynamic-update problem for GR(1): updating the behavior of an already running synthesized controller such that it would safely and dynamically, without stopping, start conforming to a modified, up-to-date specification. We formally define the dynamic-update problem and present a sound and complete solution that is based on the computation of a bridge-controller. We implemented the work in the Spectra synthesis and execution environment and evaluated it over benchmark specifications. The evaluation shows the efficiency and effectiveness of using dynamic updates. The work advances the state-of-the-art in reactive synthesis and opens the way to its use in application domains where dynamic updates are a necessary requirement.

ACM Reference Format:
Gal Amram, Shahar Maoz, Itai Segall, and Matan Yossef. 2022. Dynamic Update for Synthesized GR(1) Controllers. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510054

1 INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [35]. Rather than manually constructing an implementation of a reactive controller and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation is automatically obtained for a given specification, if such an implementation exists.

GR(1) is a fragment of Linear Temporal Logic (LTL) that has an efficient symbolic synthesis algorithm [7] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [10, 24]. GR(1) specifications include assumptions and guarantees about what needs to hold on all initial states, on all states and transitions (safety), and infinitely often on every run (justice). GR(1) has been used in several application domains, e.g., to specify and implement autonomous robots [20, 25], control protocols for smart camera networks [33], distributed control protocols for aircraft vehicle management systems [32], and device drivers [36]. Several tools support GR(1) synthesis [6, 11, 26, 38].

Many reactive systems are very difficult or very expensive to stop. Examples include mission and business critical systems whose continuous operation is one of their key requirements. Thus, when updates in functionality are required, the behavior of such a system has to be updated while it is executing. Indeed, our main motivation for the present work comes from a proof-of-concept project we have conducted, on using reactive synthesis in several application domains, as a collaboration between Tel Aviv University and Nokia Bell Labs. Dynamic update was presented by the industry partner as a necessary requirement for the use of reactive synthesis in the target application domain.

In this work we present a formulation and solution of the dynamic-update problem for synthesized GR(1) reactive systems. Specifically, given an already executing controller C_1, synthesized from some GR(1) specification $Spec_1$, and given a new GR(1) specification $Spec_2$, we formalize a notion of an update-strategy, a strategy for the system to force the execution on a path from the current state of C_1 to some state of C_2, a controller synthesized from $Spec_2$, while taking some steps in compliance with $Spec_1$ and from some point on start taking steps in compliance with $Spec_2$. Note that an update-strategy execution starts from the current state of C_1, which is a moving target. The dynamic-update problem is to compute and then execute such an update-strategy, if one exists.

We solve the dynamic-update problem for GR(1) by showing how to construct a bridge controller, which implements an update-strategy, and apply it to C_1 dynamically, while it is executing. We prove that our solution is correct and complete, i.e., that when an update is possible we find one and execute it, and when an update is not possible, we report so. We further prove that our solution is optimal, i.e., that it minimizes the maximal possible bridge length. Finally, we present a heuristic optimization based on the early detection of a successful update, which aims to save computation time and to shorten the length of the *actual* bridge that is executed. This optimization is one of the unique features of our work, taking advantage of the dynamic nature of the problem.

An important characteristics of our setup and algorithms is that they are symbolic. Thus, the update-strategy we compute is a symbolic representation of all possible correct and shortest bridges. Then, the actual bridge that will be executed depends on the state in which the running controller might be in when the bridge is ready, and of course on the behavior of the environment.

Note that the dynamic-update problem is relevant for setups where the synthesized controller is implemented in software, as in, e.g., various robotics setups [20, 25, 26], and not in setups where it is expected to be implemented as a hardware circuit, e.g., in the AIGER format [5] used in the SYNTCOMP competition series [3].

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510054

Indeed, our solution takes advantage of the recently presented just-in-time reactive synthesis [28], which is applicable only to software implementations of synthesized controllers.

We have implemented dynamic updates on top of the open-source Spectra synthesis and execution environment [1, 26]. Given an already executing synthesized system, the implementation allows the engineer to define a new specification and dynamically update the executing system to comply with it. We deliberately divide the implementation between two machines that communicate over TCP, the machine that runs controller and bridge synthesis, and the machine that executes the controller (e.g., a robot). Thus, our implementation supports a truly remote dynamic deployment as is indeed required in the target application domain defined by Nokia Bell Labs.

We evaluated our work using three sets of benchmark specifications. The evaluation shows the efficiency and effectiveness of our work. All examples and experiment results we report on in this paper are implemented in Spectra and available in supporting materials [2].

The dynamic-update problem for synthesized controllers has been studied before. In particular, Nahabedian et al. studied dynamic updates in the context of event-based controller synthesis [30]. Our work is distinctive in several ways, most notably (A) in its applicability and expressiveness - support for the GR(1) fragment of LTL, including safeties and justices, with a computation model that is not specific to event-based specifications; and (B) in its efficiency and scalability - thanks to the symbolic nature of the algorithms and implementation. We discuss related work in detail in Sect. 8.

1.1 An Illustrative Running Example

To demonstrate the dynamic-update problem and our solution, we consider a variant of the obstacle evasion specification, which is inspired by robotic motion planning and was recently used as a benchmark in the synthesis literature, see, e.g. [11, 14, 31]. Moreover, this problem is very similar in nature, size, and expected updates, to the target application developed in Nokia Bell Labs, a general platform for scheduling the activities of several robots while they patrol and perform different tasks in an environment with several target locations and obstacles. The target locations, the specific tasks, and the location of most obstacles are fixed during most normal execution but do change dynamically from time to time. Scalability in terms of the number of locations and dynamic update without stopping are considered critical requirements.

Consider a single cell sized robot and a 2×2 cells sized obstacle, both moving on an $n \times n$ grid. Initially, the robot and the obstacle start in two opposite corners of the grid. Both the robot and the obstacle can move to any adjacent cell. The robot moves twice upon every step of the obstacle. Moreover, the obstacle chases the robot and always tries to get closer to it. The problem is to synthesize a strategy for the robot so that collision never occurs, regardless of the obstacle's behavior. We model the robot's and obstacle's locations via two coordinates $robX, robY \in \{1, \ldots, n\}$, and $obsX, obsY \in \{1, \ldots, n-1\}$.[1] We provide the complete specification with simulation videos in supporting materials [2].

[1]The coordinates specify the location of the obstacle's upper left corner.

```
1  . . .
2  pred robotLoc(x,y): robX = x & robY = y;
3
4  // locations the robot must never visit
5  gar Avoid1: G !robotLoc(2,1);
6  gar Avoid2: G !robotLoc(6,4);
7  gar Avoid3: G !robotLoc(7,7);
8  gar Avoid4: G !robotLoc(7,4);
9
10 // locations the robot must visit infinitely often
11 gar AlwEventuallyVisit1: GF robotLoc(3,3);
12 gar AlwEventuallyVisit2: GF robotLoc(1,7);
13 gar AlwEventuallyVisit3: GF robotLoc(5,2);
14
15 // a switching condition
16 switch robotLoc(2,8);
```

Listing 1: Examples of safety (avoid) guarantees, justice (always-visit) guarantees, and a switching condition, using Spectra syntax. These are the parts of the specification that may be updated from time to time in our example.

In addition to evading the moving obstacle, in our specification, the robot has some safety guarantees, to avoid certain locations (avoid guarantees) and some justice guarantees, to visit some other locations infinitely often (always-visit guarantees).[2] We provide an example of such guarantees in Lst. 1. Fig. 1 illustrates our example grid world.

Most importantly for the dynamic update context, the avoid and always-visit guarantees may change from time to time. Specifically, we consider that during the controller's execution, the system requirements may change, so the engineers have to update the robot's avoid and always-visit guarantees, while the robot is running and as it continues to evade the moving obstacle. Moreover, we include an optional switching condition, *cond*, specified by the declaration **`switch`**. The switching condition requires the robot to switch to the new specification only in a state where the assertion *cond* holds. Again, the need for supporting an optional switching condition, as part of the dynamic-update problem, is a requirement defined by Nokia Bell Labs.

Note that this example synthesis problem is not trivial. As the robot evades the obstacle that chases it, it has not only to avoid certain locations, but also to always eventually visit some other locations. Since the obstacle is chasing it, in order to make sure it can visit a location, the robot sometimes has to first "lure" the obstacle to move farther from the target location in point and then, when the obstacle is far enough, race to get to the target in point before the obstacle gets there or blocks the way. On top of this non-trivial behavior, we add here the dynamic update of the avoid and always-visit locations.

We now demonstrate the dynamic-update use case. The engineers have synthesized a controller from an obstacle evasion specification and started executing it on a robot. Using this controller, the robot evades the moving obstacle that chases it while avoiding some cells and always eventually visiting some other cells, as specified in the specification. After some time, the requirements change, so the engineers write a new specification that includes up-to-date avoid and always-visit guarantees (and optionally a switching condition). Thus, while the first controller is running, the engineers use our tool to synthesize a controller for the new

[2]This is an instance of the *patrolling* pattern from [29].

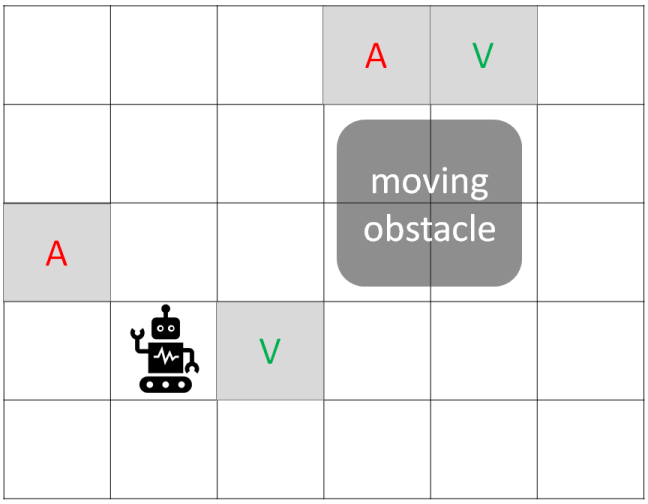


Figure 1: An illustration of our example in a grid world with a robot that has to avoid the A cells and always-visit the V cells, all while evading a 2x2 moving obstacle chasing it.

specification and, most importantly, a bridge-controller. Then, if the bridge covers the state in which the first controller happens to be at when it is ready, the bridge forces the execution of the robot to take some steps in compliance with the first specification, reach a state that satisfies the switching condition (e.g., `robotLoc(2,8)` (see Lst. 1)), and from that point on start taking steps in compliance with the new specification, eventually starting to execute using the new controller.

2 PRELIMINARIES

2.1 Reactive Systems

Given a set of Boolean variables $\mathcal{V}$, a *state* s over $\mathcal{V}$ is a subset of $\mathcal{V}$, i.e., $s \in 2^{\mathcal{V}}$. If $s_1, \ldots, s_n$ are states over the disjoint sets $\mathcal{V}_1, \ldots, \mathcal{V}_n$, resp., we write $(s_1, \ldots, s_n)$ as an abbreviation for $s_1 \cup \cdots \cup s_n$. An *assertion* over $\mathcal{V}$ is a Boolean formula over $\mathcal{V}$. For an assertion ρ and a state s, we write $s \models \rho$, if ρ is evaluated to **`true`** by assigning **`true`** to all variables in s and **`false`** to variables in $\mathcal{V} \setminus s$. If $s \models \rho$, we say that s is a ρ-state. For a set of variables $\mathcal{V}$, $\mathcal{V}'$ is the set of variables obtained by replacing each $v \in \mathcal{V}$ with its *primed version*, $v' \in \mathcal{V}'$. Furthermore, for a state s over $\mathcal{V}$, s' is a state over $\mathcal{V}'$ defined by $s' = \{v' : v \in s\}$.

A reactive system repeatedly reacts to inputs coming from its environment. This interaction is modeled as a two-player game, played between the environment player and the system player. At each turn the environment provides an input, and the system replies with an output. This interaction is formally captured through a game structure.

A *game structure* is a tuple $GS = (\mathcal{X}, \mathcal{Y}, \theta^e, \theta^s, \rho^e, \rho^s)$ [3] where (1) $\mathcal{X}$ is the set of variables owned by the environment; (2) $\mathcal{Y}$ is the set of variables owned by the system, and it is disjoint to $\mathcal{X}$; (3) θ^e is an assertion over $\mathcal{X}$, which constitutes the *initial assumptions* of the environment; (4) θ^s is an assertion over $\mathcal{X} \cup \mathcal{Y}$, which constitutes the *initial guarantees* of the system; (5) ρ^e is assertion over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$ which constitutes the *safety assumptions* of the environment; and (6) ρ^s is an assertion over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$ which constitutes the *safety guarantees* of the system.

Intuitively, a game structure models the two-player game described above as follows. First, the environment chooses an input $s_x^0 \in 2^{\mathcal{X}}$ such that $s_x^0 \models \theta^e$, and the system replies with an output $s_y^0 \in 2^{\mathcal{Y}}$ such that $(s_x^0, s_y^0) \models \theta^s$. (s_x^0, s_y^0) forms the play's first state, and then the players repeatedly construct the next states by choosing inputs and outputs. From state s, the environment can choose an input $s_x \in 2^{\mathcal{X}}$ if $(s, s_x') \models \rho^e$, and the system can respond with $s_y \in 2^{\mathcal{Y}}$ if $(s, s_x', s_y') \models \rho^s$. Game structures naturally extend to finite type variables, which are not necessarily Boolean.

Example 2.1. For the obstacle evasion example (see Sect. 1.1), $\mathcal{X}$={*obsX*, *obsY*} and $\mathcal{Y}$={*robX*, *robY*}. The initial assumptions and guarantees are: θ^e=($obsX = 7 \wedge obsY = 7$); θ^s=($robX = 1 \wedge robY = 1$). The safety assumptions model the way the obstacle moves (to an adjacent cell at each step). The safety guarantees are a conjunction of movement restriction (two cells at most at each step) and the avoid guarantees from Lst. 1, e.g., $robX \neq 2 \vee robX \neq 1$.

Let $GS = (\mathcal{X}, \mathcal{Y}, \theta^e, \theta^s, \rho^e, \rho^s)$ be a game structure. The game structure's states are $2^{\mathcal{X} \cup \mathcal{Y}}$. A transition from state s to state t is *consistent* with ρ^e (resp. ρ^s, (ρ^e, ρ^s)) if $(s, t') \models \rho^e$ (resp. $\models \rho^s$, $\models \rho^e \wedge \rho^s$). Consistency with ρ^e is defined also when $t \in 2^{\mathcal{X}}$. A transition that is consistent with ρ^e (resp. ρ^s, (ρ^e, ρ^s)) is called a ρ^e-transition (and resp. for the others). A sequence of states is consistent with ρ^e (resp. ρ^s, (ρ^e, ρ^s)) if any pair of consecutive states in it is consistent with ρ^e (resp. ρ^s, (ρ^e, ρ^s)). A state s is a *deadlock* for the environment if $\forall s_x \in 2^{\mathcal{X}}((s, s_x') \not\models \rho^e)$. s is a deadlock for the system if $\exists s_x \in 2^{\mathcal{X}}((s, s_x') \models \rho^e \wedge \forall s_y \in 2^{\mathcal{Y}}((s, s_x', s_y') \not\models \rho^s))$. A *play* is a sequence of states such that (1) the first state in the sequence $s_0 \models \theta^e \wedge \theta^s$, (2) any pair of consecutive states in the sequence is consistent with (ρ^e, ρ^s), and (3) the sequence is either infinite, or ends in a deadlock.

The system employs a strategy to repeatedly choose the next output. This output depends on the given input and on the states traversed so far. A *strategy* is a partial function $\sigma : (2^{\mathcal{X} \cup \mathcal{Y}})^+ \times 2^{\mathcal{X}} \rightarrow 2^{\mathcal{Y}}$. A state s is *consistent* with σ if for any $s_x \in 2^{\mathcal{X}}$ such that $(s, s_x') \models \rho^e$, (s, s_x) is in the domain of σ. We further require that a strategy satisfies the following: if $s_0, \ldots, s_i \in (2^{\mathcal{X} \cup \mathcal{Y}})^+$ is consistent with σ, and $(s_i, s_x') \models \rho^e$ for $s_x \in 2^{\mathcal{X}}$, then (1) $s_0, \ldots, s_i, s_x$ is in the domain of σ, (2) for $s_y = \sigma(s_0, \ldots, s_i, s_x)$, $(s_i, (s_x', s_y')) \models \rho^s$, and (3) we say that $(s_0, \ldots, s_i, (s_x, s_y))$ is consistent with σ. A play is consistent with σ if every prefix of it is consistent with σ. We say that σ is from $S \subseteq 2^{\mathcal{X} \cup \mathcal{Y}}$ if every $s \in S$ is consistent with σ.

A controller is an algorithm that implements a strategy. Therefore, a controller receives a state and an input as arguments, and returns an output. During a play, the controller is repeatedly applied and reacts to the sequence of inputs produced by the environment. Hence, a controller retains memory from one invocation to another.

Remark 1. *For the study of dynamic updates, we consider game structures with no initial constraints, since an update occurs from an "intermediate" state, which depends on the execution of another, running controller. All notations defined above are transferred to game*

[3] Some presentations use a game structure that includes a graph and a winning condition formula. In our notation they are separate. The difference is only notational.

structures with no initial constraints with a minor exception: we do not set rules for the initial state of a play.

2.2 The GR(1) Winning Condition

A GR(1) formula over a set of variables $\mathcal{V}$ is a formula of the form $\wedge_{i=1}^{m} GF(J_i^e) \rightarrow \wedge_{j=1}^{n} GF(J_j^s)$, where $J_1^e, \ldots, J_m^e, J_1^s, \ldots, J_n^s$ are assertions over $\mathcal{V}$. G and F are the Linear Temporal Logic (LTL) [34] operators, *Globally* and *Finally*, resp. Therefore, a GR(1) formula states that if the assertions $J_1^e, \ldots, J_m^e$ (the *justice assumptions*) hold infinitely often, then $J_1^s, \ldots, J_n^s$ (the *justice guarantees*) hold infinitely often as well.

Example 2.2. For the obstacle evasion example (see Sect. 1.1), the justice guarantees are the *Always Eventually Visit* guarantees in Lst. 1, e.g., $GF(robX = 3 \wedge robY = 3)$.

For a game structure GS and a GR(1) formula φ over $\mathcal{X} \cup \mathcal{Y}$, (GS, φ) is a *GR(1) specification*. For an infinite play $\pi = s_0, s_1, \ldots$ we write $\pi \models \varphi$ if (1) there exist $1 \leq k \leq m$ and $i \geq 0$ such that $\forall j \geq i (s_j \not\models J_k^e)$, or (2) for each $1 \leq k \leq n$, there are infinitely many states s_i in π such that $s_i \models J_k^s$. Given a specification (GS, φ), the play π wins for the system if (1) π ends in a deadlock for the environment, or (2) π is infinite and $\pi \models \varphi$. A strategy wins from state s if any play from s, consistent with it, wins for the system. The winning region of a specification is the set of all states from which a winning strategy exists. A strategy is a winning strategy if it wins from every state in the specification's winning region.

3 PROBLEM DEFINITION

Consider a situation in which an engineer synthesized a controller C_1 w.r.t. some GR(1) specification, $Spec_1$, and during an execution of C_1, new up to date system requirements are defined, formulated in a new specification, $Spec_2$. Apparently, the engineer can use $Spec_2$ to synthesize a new controller C_2, stop C_1 execution, and start executing C_2. However, the execution of the system we deal with must not be stopped; it should be updated to behave according to $Spec_2$ while it is running. Hence, from a user perspective, once $Spec_2$ is provided to the synthesizer, the controller can take several steps in compliance with $Spec_1$ and, from some point on, should satisfy $Spec_2$.

Below we elaborate on the requirements that an update-strategy should satisfy. Then, we formalize these to obtain a precise definition of the dynamic-update problem.

Timing of change in environment behavior. We assume that once $Spec_2$ is available, the environment starts behaving according to its safety assumptions. This assumption is reasonable when the environment is independent and uncontrollable. In these cases we adapt $Spec_2$ immediately, as it provides an up-to-date perspective on the environment's behavior. However, minor modifications in the solution we shall provide capture other possibilities one may consider, as we elaborate later in Remark 5.

A switching condition. The dynamic-update problem may include a switching condition. That is, an assertion that must hold at the state in which the controller starts obeying $Spec_2$'s guarantees. For example, we may require that the switch will occur only when the robot is located at some specified region (i.e., as we have done in Lst. 1), or when the distance from the moving obstacle exceeds some threshold etc. We remark that the requirement to allow the definition of a switching condition as part of the problem definition came from the concrete case study we work on with Nokia Bell Labs.

Note that the switching condition is optional. If the engineer does not provide a switching condition, we consider it to be `true`.

A bounded switching phase. We require that the system update (unlike, e.g., satisfaction of a justice guarantee) occurs within a bounded number of steps. Consequently, the dynamic-update problem formulation excludes the possibility of an unbounded switching phase. We thus require that reaching a state that satisfies the switching condition and from which $Spec_2$'s guarantees hold will occur within a bounded number of steps. Note, however, that we do not assume a given, fixed bound, as input, but only require that a bound on the length of the switching phase exists.

We now formalize all the above. Note that for the simplicity of presentation, in the definition below we consider that the two specifications are defined over the same variables. This is justified as follows: if $Spec_1$ is defined over $(\mathcal{X}_1, \mathcal{Y}_1)$, and $Spec_2$ over $(\mathcal{X}_2, \mathcal{Y}_2)$, we may regard both as specifications over $(\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{Y} = \mathcal{Y}_1 \cup \mathcal{Y}_2)$, since variables that are defined but do not appear in a specification have no effect on its semantics.

Definition 3.1 (update-strategy). Let $Spec_1 = (GS_1 = (\mathcal{X}, \mathcal{Y}, \theta_1^e, \theta_1^s, \rho_1^e, \rho_1^s), \varphi_1)$ and $Spec_2 = (GS_2 = (\mathcal{X}, \mathcal{Y}, \rho_2^e, \rho_2^s), \varphi_2)$[4] be GR(1) specifications, and let *cond* be an assertion over $\mathcal{X} \cup \mathcal{Y}$ (the switching-condition). A strategy σ is an update-strategy from state s_0 if, assuming the environment takes actions in compliance with ρ_2^e, there exists $k \geq 0$ such that for every prefix of a play, $s_0, s_1, \ldots, s_k$, consistent with σ, there exists s_i with $i \leq k$, for which:

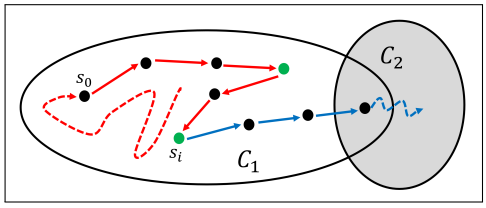
(1) the switching phase prefix $s_0, \ldots s_i$ is consistent with (ρ_2^e, ρ_1^s);
(2) the switching condition holds: $s_i \models cond$;
(3) the switching phase future $s_i, s_{i+1}, \ldots$ is consistent with (ρ_2^e, ρ_2^s) and $s_i, s_{i+1}, \ldots \models \varphi_2$.

Intuitively, the state s_i in Def. 3.1 is the state in which the switch actually occurs. (1) The system takes ρ_1^s-transitions until reaching s_i, (2) s_i satisfies the switching condition, and (3) the system takes ρ_2^s-transitions from s_i. If there is no switching condition, we set $cond$ = `true` (and then item (2) holds vacuously).

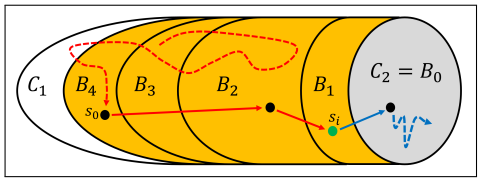
Definition 3.2 (dynamic-update problem). Given game structures and a switching condition as in Def. 3.1, compute a strategy σ and a set W, such that σ is an update-strategy from each $s \in W$ (soundness), and there exists an update-strategy from a state s iff $s \in W$ (completeness), iff such a set and strategy exist.

Illustration 1. *Figure 2a illustrates the dynamic-update problem. The red dashed arrow denotes a C_1 computation that reaches state s_0, from which we apply an update-strategy. The update-strategy takes several (ρ_2^e, ρ_1^s)-transitions (red arrows) until reaching state s_i ($i = 5$ in our example) that satisfies cond (marked in green). From s_5, only (ρ_2^e, ρ_2^s)-transitions are taken (blue arrows). The suffix of that computation is performed by the controller C_2, which ensures satisfaction of $Spec_2$. Note that s_3 also satisfies cond, but it is* not *our*

[4] $Spec_2$ does not include initial constraints as such constraints have no role in the dynamic-update context, see Sect. 2, Rem. 1



(a) Update-strategy



(b) Bridge construction and its execution

Figure 2: Illustrations of an update-strategy and our solution. Red arrows denote $Spec_1$-outputs, blue arrows denote $Spec_2$-outputs, and green states satisfy the assertion *cond*

"switching state", since two additional ρ_1^s-transitions were taken after visiting it.

Example 3.3. In our obstacle evasion example, the required update-strategy is not trivial. The robot must consider the old and new avoid guarantees, which may block some paths to and from the switching cell, do it while observing the obstacle's locations and reacting to its moves, and eventually end up in a state from which the new controller can take over.

4 SOLVING DYNAMIC UPDATE

To solve the dynamic-update problem we need to synthesize an update-strategy or state that such a strategy does not exist. We solve the problem in two stages. First, we compute the system's winning region w.r.t. $Spec_2 = (GS_2, \varphi_2)$, W_2, and synthesize a matching controller C_2. If W_2 is empty, we already know that an update-strategy does not exist.

Second, if W_2 is not empty, we move on to synthesize a bridge-controller B. The bridge forces the play from the current (but yet unknown) state in C_1 into W_2, in finitely many steps. On its way, the bridge makes sure passing a state that satisfies *cond* after which violations of the safety guarantees of $Spec_2$ (i.e., ρ_2^s) are not allowed.

Note that the constructions of C_2 and the bridge-controller B are independent of one another. C_2 is constructed from $Spec_2$, and the bridge is constructed using a reachability game, based on the old and new specifications, as we elaborate next. Furthermore, importantly, both constructions are independent of the controller C_1, which keeps running (typically on a separate machine) while C_2 and the bridge-controller B are constructed.

The computation of W_2 and C_2 is done by a standard GR(1) synthesis and we omit its details. We turn to describe the construction of the bridge-controller and the execution of the update. Finally, we discuss correctness, optimality, and complexity.

Remark 2. *Note that the bridge-controller that we compute in Sect. 4.1 below does not represent a single possible update. Rather, it is a symbolic representation of all possible correct and shortest bridges. Then, in Sect. 4.2, we show how a concrete bridge is executed, depending on the state in which the running controller might be in when the bridge is ready, and on the behavior of the environment in every step of its execution.*

4.1 Bridge-Controller Construction

To synthesize a bridge-controller we extend $\mathcal{Y}$ with two variables, switch and allowed. Intuitively, switch is turned on by the system to mark that we switched to ρ_2^s-transitions. Unlike switch, allowed is an auxiliary variable, i.e., a variable whose value is uniquely determined by its previous value and by the valuation of all other variables. allowed monitors the play-states, and evaluates to **`true`** only when the system is allowed to switch: if only ρ_2^s-transitions have been taken since visiting a *cond*-state, i.e., a state where the assertion *cond* holds.

We consider the game structure $\widehat{GS} = (\mathcal{X}, \mathcal{Y} \cup \{\text{switch}, \text{allowed}\}, \rho_2^e, \rho^s)$ where ρ^s is the conjunction of the formulas listed below, divided into three groups. *Transitions* (T1,T2) set rules for an appropriate use of ρ_1^s and ρ_2^s. *Switch* (S1-S3) set rules for the valuation of switch. *Permission* (P1) defines how allowed is updated in each step. Hence,

$$\rho^s = \bigwedge \{\text{T1,T2,S1,S2,S3,P1}\}, \text{where}$$

T1:= $\rho_1^s \vee \rho_2^s$. The system must always follow $Spec_1$ or $Spec_2$ safety guarantees.

T2:= $\text{switch}' \rightarrow \rho_2^s$. When switch is true, only ρ_2^s-transitions are allowed.

S1:= $(\neg\text{switch} \wedge \text{switch}') \rightarrow \text{allowed}'$. We turn on switch for the first time only if we are allowed to.

S2:= $\text{switch} \rightarrow \text{switch}'$. Once switch is turned on, it remains true.

S3:= $\neg\rho_1^s \rightarrow \text{switch}'$. When we take a $\neg\rho_1^s \wedge \rho_2^s$-transition, we must switch.

P1:= $\text{allowed}' \leftrightarrow ((cond \wedge \rho_2^s) \vee (\text{allowed} \wedge \rho_2^s))$. allowed is valuated to **`true`** iff (1) we take a ρ_2^s-transition from a *cond*-state, or (2) allowed holds, and we continue with a ρ_2^s-transition.

Let $\widehat{\mathcal{Y}} = \mathcal{Y} \cup \{\text{switch}, \text{allowed}\}$, $\widehat{\mathcal{V}} = \mathcal{X} \cup \widehat{\mathcal{Y}}$, and $\widehat{W_2} = \{s \in 2^{\widehat{\mathcal{V}}} : s|_{\mathcal{X}\cup\mathcal{Y}} \in W_2\} = W_2 \times 2^{\{\text{switch,allowed}\}}$, where for $\mathcal{U} \subseteq \mathcal{V}$, $s|_{\mathcal{U}}$ denotes the state $s \cap \mathcal{U}$ over $\mathcal{U}$. Let $\widehat{W} \subseteq 2^{\widehat{\mathcal{V}}}$ be the set of states from which the system can force reaching $\widehat{W_2} \wedge \text{switch}$, and let B be a matching controller. That is, B wins the reachability game with target set $\widehat{W_2} \wedge \text{switch}$ [17, Ch. 2].

$\widehat{W}$ and B are constructed by the procedure presented in Alg. 1. The algorithm employs the modal μ-calculus [19] *controllable-predecessor* operator $\diamondsuit$ [7], which is defined as follows. Let $\mathcal{X}$ and $\mathcal{Y}$ be the environment and system variables, resp., and ρ^e and ρ^s be matching safety assumptions and guarantees, resp. For a set of states W,

$$\diamondsuit_{(\rho^e,\rho^s)}(W) := \{s \in 2^{\mathcal{X}\cup\mathcal{Y}} : \forall s_x \in 2^{\mathcal{X}} ((s, s_x') \models \rho^e \rightarrow \exists s_y \in 2^{\mathcal{Y}} ((s, s_x', s_y') \models \rho^s \wedge (s_x, s_y) \in W))\}.$$

That is, $\diamondsuit_{(\rho^e,\rho^s)}(W)$ includes all states from which the system can force reaching W in a single step: for each legal input by the environment, the system has a legal output to reach W.

Consider Alg. 1, and observe that there exists $k \geq 0$ such that, for $s \in \widehat{W}$, B forces reaching $\widehat{W_2} \wedge \text{switch}$ from s within k steps. Indeed, k is the number of fixed-point iterations performed in the

Algorithm 1 Computing $\widehat{W}$ and a matching bridge-controller B

$j = 0, Z_0 = \widehat{W_2} \wedge \text{switch}, B_0 = \emptyset$
repeat
  j++
  $Z_j = Z_{j-1} \cup \bigcirc_{(\rho_2^e, \rho^s)} Z_{j-1}$
  $B_j = B_{j-1} \cup \{(s,t) \in (Z_j \setminus Z_{j-1}) \times Z_{j-1} : (s,t') \models \rho^s\}$
until $Z_j = Z_{j-1}$
$\widehat{W} = Z_j, B = B_j$, **return** $\widehat{W}, B$

computation of $\widehat{W}$. In the next subsection we prove that B is the bridge-controller we are interested in.

Remark 3. *To construct B, one may suggest an alternative, simpler, naive approach to the problem, using two reachability strategies: a (ρ_2^e, ρ_1^s)-reachability strategy with target cond, from which we can apply a (ρ_2^e, ρ_2^s)-reachability strategy with target W_2. This suggestion is sound, but incomplete, i.e., it may not find all states from which an update-strategy exists. For example, consider a state s such that (1) for some ρ_2^e-inputs from s, the system has a ρ_1^s-response that leads the play to a state from which an update-strategy exists, and (2) for all other ρ_2^e-inputs from s, the system has a ρ_2^s-response that leads to a state from which an update-strategy exists. Hence, an update-strategy exists from s, but the two-phase reachability suggestion above fails to identify s.*

Remark 4. *We present* switch *and* allowed *as new variables only for the presentation of the bridge construction. In practice, we do not add them to the game structure and they only serve as memory variables for the bridge-controller: when activated from $W = \widehat{W}|_{\mathcal{X} \cup \mathcal{Y}}$, B forces reaching W_2 within k steps, for some $k > 0$, so that the memory variable* switch *is* **`true`**. *Recall that if* switch *is* **`true`**, *then only ρ_2^s-transitions have been taken since visiting a cond-state.*

Remark 5. *We construct the bridge while considering the safety assumptions ρ_2^e, as we made a design decision, to immediately adapt the new environment assumptions (see the considerations we presented before Def. 3.1). Other options are easily supported via minor modifications in the game structure $\widehat{GS}$. As an example, we can let the controller choose when the environment switches from ρ_1^e to ρ_2^e, by adding a fresh system variable and constraining the environment's behavior on its value. Likewise, we can decide that the change in assumptions will occur in a cond-state, when the system switches to ρ_2^s etc. These alternatives are relevant for setups where the timing of the change in the environment can be controlled by the system or its operator.*

4.2 Execution of the Update

We are ready to present the execution of the update. Given $Spec_2$'s winning region W_2, a matching controller C_2, a bridge-controller B, and its winning region W, Alg. 2 shows the execution of the update U as follows. First, if $W = \emptyset$, then an update-strategy does not exist and we announce it (line 0). Otherwise, we look at the current play-state s_0, and check whether $s_0 \in W$ (initial activation (I)). In case $s_0 \in W$, we proceed to (II), the code that we activate from this point on. In (II), we iteratively generate a response s_y given the current state s and the input s_x. We apply B from s_0, until reaching

Algorithm 2 The execution of the update U

0: **if** $W = \emptyset$ **then return** "switching cannot be forced from any state"
  // (I). initial activation from state s_0
1: **if** $s_0 \notin W$ **then return** "switching cannot be forced from the current state"
  // (II). a response from state s and input s_x
1: **if** $(s \notin W_2) \vee (\text{switch} == \texttt{false})$ **then** $s_y = B(s, s_x)$
2: **else** $s_y = C_2(s, s_x)$
3: **return** s_y

W_2 while switch is **`true`** (line (II).1). Afterwards, we apply C_2 for the remainder of the play (line (II).2).

In case $s_0 \notin W$, we report to the engineer that an update-strategy does not exist from the current state (line (I).1). Note that if $s_0 \notin W$, the engineer can try and execute Alg. 2 again, as the play may traverse into W.

Illustration 2. *Figure 2b illustrates a bridge controller B, as computed by Alg. 1, and how it is used. The dashed red arrow represents a play prefix induced by the $Spec_1$ controller, C_1. In parallel to this play prefix, we compute the reachability bridge controller B, following Alg. 1. Fortunately, the computation of B ends when the play prefix reaches state s_0, covered by B. Hence, the bridge takes control, takes two (ρ_2^e, ρ_1^s)-transitions (red arrows), reaches a state s_i which satisfies cond ($i = 2$ in our example, green states satisfy cond), takes an additional (ρ_2^e, ρ_2^e)-transition (blue arrow) and passes control to the $Spec_2$ controller C_2 for the remainder of the play (dashed blue arrow).*

4.3 Correctness, Optimality, Complexity

Our solution is sound and complete. If there is an update-strategy we find it and execute it; if there is not any, we report so. Formally:

Theorem 4.1 (construction-correctness). *Alg. 2 is sound and complete:*

(1) The algorithm implements an update-strategy from W.
(2) There exists an update-strategy from s iff $s \in W$.

Beyond correctness, we mention that the reachability strategy minimizes the number of steps taken until the activation of the new controller. To elaborate, the bridge not only minimizes the longest possible switching phase length (denoted k), but also minimizes the number of steps actually executed (denoted j), see Def. 3.1. Both results follow from the next theorem.

Theorem 4.2 (bridge-optimality). *For $s \in W$, let $k(s)$ be the minimal integer s.t. there exists an update-strategy from s that satisfies the conditions of Def. 3.1 with the parameter $k(s)$. Then, the bridge B forces reaching $\widehat{W_2} \wedge$ switch from s within, at most, $k(s)$+1 steps.*[5]

The proofs of Thms. 4.1 and 4.2 appear in [2].

Finally, note that, asymptotically, the update controller synthesis does not add computational cost to the GR(1) synthesis of the new controller. The construction of the bridge involves a single fixed-point loop and is thus computed in $O(N)$ symbolic operations,

[5]The possible single redundant step reported in the theorem can be avoided by a minor modification of $\widehat{GS}$. Mainly, turning on *allowed* when *cond* is reached. We decided to give the "single-redundant-step" version as it slightly simplifies the correctness proof.

Algorithm 3 Optimized computation of $Z \subseteq \widehat{W}$ and B (replacing Alg. 1)

1: $j = 0, Z_0 = \widehat{W_2} \wedge \text{switch}, B_0 = \emptyset$
2: **repeat**
3: $s = C_1$'s current state
4: **if** $(s, \neg\text{switch}, \neg\text{allowed}) \in Z_j$ **then**
5: stop C_1
6: $Z = Z_j, B = B_j$, **return** Z, B
7: **end if**
8: j++
9: $Z_j = Z_{j-1} \cup \langle\!\langle (\rho_2^e, \rho^s) \rangle\!\rangle Z_{j-1}$
10: $B_j = B_{j-1} \cup \{(s,t) \in (Z_j \setminus Z_{j-1}) \times Z_{j-1} : (s,t') \models \rho^s\}$
11: **until** $Z_j = Z_{j-1}$
12: $\widehat{W} = Z_j, B = B_j$, **return** $\widehat{W}, B$

where N is the size of the state space. That said, the number of fixed-point iterations is just the length of the maximal possible bridge, and in practice, the length of the bridge is orders of magnitude smaller than the state space, see our evaluation in Sect. 7.

Remark 6 (no overhead accumulation). *Our solution enables the seamless application of a sequence of dynamic updates. That is, once the update completes and Alg. 2 returns an output from C_2, C_2 controls the execution, and another dynamic update can be applied when necessary. Applying a sequence of updates comes at no cost in terms of additional variables and thus no overhead is accumulated.*

5 EARLY DETECTION OF SUCCESS

We propose an optimization for the execution of the update U, based on a modified construction of the bridge-controller. The optimization we describe can lead to a shorter switch, i.e., reduce the number of steps we take until switching to ρ_2^s-transitions and thus to a (GS_2, φ_2)-strategy. Furthermore, the optimization can reduce the computation time of the bridge.

The optimization is based on a simple idea: rather than computing the complete bridge B and only then checking whether the current state of the executing controller is covered by it, we suggest to check before every loop iteration of Alg. 1 whether the current state of the executing controller was already covered by the set B computed so far. In case B already covers the current state, we can immediately apply Alg. 2, i.e., apply B until reaching its target, and then switch to C_2. In other words, instead of computing all states from which an update can be executed, the optimization stops as soon as the current state of the controller is within the part of the bridge that was computed so far. This early detection of success, allows to stop the computation of B and thus to reduce computation time. We present this optimized construction in Alg. 3.

Illustration 3. *Figure 3 illustrates an optimized bridge computation of Alg. 3. The red dashed arrow represents a play prefix induced by the $Spec_1$ controller C_1. After the computation of each bridge component, we check if the current play state has been covered. Fortunately, after the computation of B_3, the play reaches s_0 which is covered by B as computed so far. Hence, we do not compute the remaining (semi-transparent) set B_4 that Alg. 1 would have computed. We let B take control and lead the play to B_0, from which we activate C_2.*

Note that the optimization requires more communication between the synthesizer and C_1 as well as multiple checks on whether its current state allows to switch (Alg. 3, line 4). This additional computation is however, negligible, as it involves only a small constant number of symbolic operations.

Finally, note that the heuristic optimization does not compromise soundness and completeness. It potentially reduces the number of steps during the bridge construction and it never adds steps. In our evaluation we present evidence that this optimization is very effective in reducing the bridge's computation time and actual length when executed.

6 IMPLEMENTATION

We have implemented and integrated the bridge construction of Alg. 1, the proposed dynamic-update strategy construction of Alg. 2, and further the optimized bridge construction of Alg. 3, in the Spectra specification language and synthesis environment [1, 26]. The implementation consists of two controller constructions: a just-in-time controller [28] for $Spec_2$, C_2, and a bridge controller B following Alg. 1 or Alg. 3. A just-in-time controller stores an efficient symbolic representation and computes concrete next states only when they are required. The separation between C_2 and B allows us to consider the added variables, switch and allowed, only during the bridge phase. C_2 does not use these variables, neither as game variables, nor as memory variables, and thus the just-in-time C_2 is synthesized from the new specification by the standard synthesis procedure of Spectra (including the heuristics suggested in [13]). The added cost of the dynamic-update synthesis is due solely to the construction of B.

In our implementation, the bridge-controller B minimizes the number of steps taken until we can activate C_2. Specifically, at each step, from the current state s, given input in, we apply a binary search on the array $B_0, \ldots, B_j$,[6] to find a valid output out, such that (in, out) is of lowest index, i.e., closest as possible to the target B_0. Note that the array is strictly monotonic, i.e., for all m, the set of transitions represented by B_m is a strict subset of the set of transitions represented by B_{m+1}. The search for the output out takes $\log(|B|)$ steps, where $|B|$ denotes the length of the array. Clearly, $\log(|B|)$ is bounded by the number of variables $\mathcal{X} \cup \mathcal{Y} \cup \{\text{switch}, \text{allowed}\}$. In practice, it tends to be much smaller.

[6] For efficiency, the actual implementation uses the array of winning regions $Z_0, \ldots, Z_j$ rather than the array of controllers $B_0, \ldots, B_j$. We use the B notation for the simplicity of the presentation.

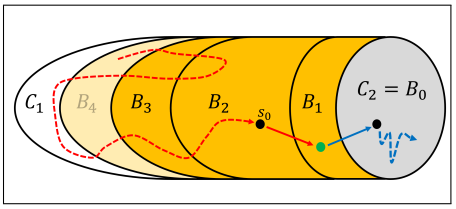


Figure 3: Optimized bridge construction. Red arrows denote $Spec_1$-outputs, blue arrows denote $Spec_2$-outputs, green states satisfy the assertion *cond*. The semi-transparent ellipse B_4 is a basic bridge component that the optimized bridge did not compute.

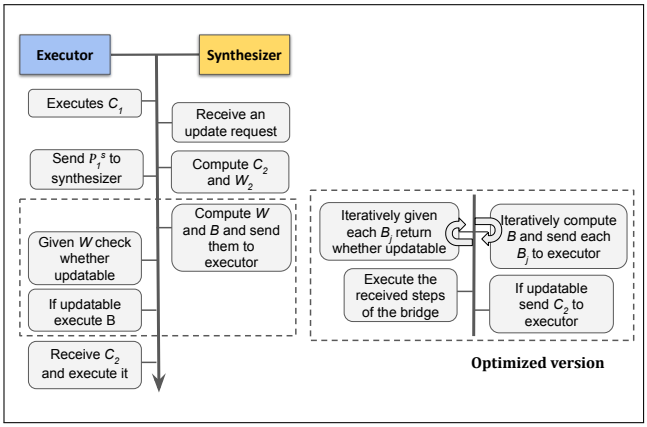


Figure 4: Dynamic-update communication timeline

The implementation is divided between an executor and a synthesizer, which communicate over TCP. It thus supports a truly remote communication between the machine that runs synthesis and the machine that executes the controller (e.g., a robot).

Figure 4 shows the timeline of communication during an update. While the executor is executing C_1, the synthesizer receives an update request for $Spec_2$ and synthesizes C_2. Then, the synthesizer uses Alg. 1 to compute B and W, the bridge and the set of states from which the system can force an update, and sends them to the executor. The executor checks whether an update is possible from its current state using W and returns the answer to the synthesizer. If the answer is positive, the executor starts executing B immediately. While B is executed, the executor receives C_2 from the synthesizer, and it starts executing C_2 right after the bridge execution is completed according to Alg. 2.

In the optimized version, where Alg. 1 is replaced by Alg. 3, the bridge is computed and sent to the executor iteratively until reaching a B_j from which the system can force an update. Then, the bridge B_j constructed so far and C_2, are executed using Alg. 2.

7 PRELIMINARY EVALUATION

We provide all the specifications, raw results, and means to reproduce the experiments described below in supporting materials [2].

7.1 Research Questions

We consider the following research questions.

RQ1. How do update-strategy synthesis and standard GR(1) synthesis compare w.r.t. running times? What is the added cost of supporting dynamic update?

RQ2. How many steps does it take for the bridge-controller to complete and activate C_2?

RQ3. Does our optimization improve performance, in terms of (a) the bridge construction running time and (b) number of steps taken to complete it?

7.2 Corpus and Applied Updates

Our corpus includes three sets of specifications: (1) instances of our running example, obstacle evasion, which we use to dynamically update the avoids and always-visit guarantees, (2) instances of a prioritized arbiter specification, a variant of a benchmark from the reactive synthesis competition SYNTCOMP [3], which we use to dynamically update the identity of the prioritized client, and (3) instances of a job scheduler specification, which we use to dynamically update a subset of the servers that goes down for maintenance. For each of these sets, we created many pairs of specifications where each pair is an example of an update from the first specification in the pair to the second specification in the pair. We describe them below.

Configurations based on the obstacle evasion problem. For every $n = 8, 16, 32, 64$ and $m = 3, 5, 7, 9$, we created 30 pairs of realizable specifications by adding to the obstacle evasion specification over an $n \times n$ grid: (1) m safety guarantees: randomly chosen m cells that the robot must avoid; (2) m justice guarantees: randomly chosen m cells that the robot must visit infinitely often; and (3) one randomly generated switching condition of the form $cond = (robX{=}i \wedge robY{=}j)$, i.e., a condition that instructs the robot to visit a randomly chosen designated cell as part of the update-strategy (see Def. 3.1). We denote these pairs by pairs(n, m).

Configurations based on the prioritized arbiter problem. The arbiter (the system) grants requests raised by n clients (the environment). It must never grant two requests simultaneously, it can grant client i's request only when client i raises a request, and it must make sure that each request is eventually granted. We assume that a client lowers a request iff it was just granted. Finally, prioritization means that one of the clients is prioritized: a pending request by the prioritized client must be granted before any other client's request is granted. We considered variants with $n = 70, 80, 90, 100$ clients. For each of these, we created 30 pairs of specifications with n clients, that differ in the identity of the prioritized client. Our switching condition is the assertion: "no pending request by the prioritized client". We denote these pairs by pairs(n).

Configurations based on the job scheduler problem. The scheduler (the system) assigns jobs of different lengths (max length k) to one of n servers. The scheduler must eventually assign every incoming job. Once a job is assigned to a server, it cannot be reassigned. When a server is assigned a job of length $l \leq k$, it works on it for l consecutive states, after which it is available for new assignments if any. A new job must not be assigned to a server that is currently down for maintenance. For every $n = 4, 6, 8, 10$ and $k = 4, 8, 12, 16$, we created 30 pairs of realizable specifications, where updates specify which server should now go down for maintenance. Correct update strategies ensure that existing jobs that are already assigned to a server complete before the server goes down for maintenance. We denote these pairs by pairs(n, k).

Overall, our corpus includes 1080 configurations: 16 sets of pairs for the obstacle evasion problem, 4 sets of pairs for the prioritized arbiter problem, and 16 sets of pairs for the job scheduler problem, each consisting of 30 configurations.

7.3 Validation

To validate our implementation we defined assertions over the behavior of the system execution during and after the update: assertions on the bridge phase, an assertion stating that the switching

state satisfies *cond*, and an assertion stating that after the bridge, the execution should match the behavior specified by $Spec_2$.

We verified these assertions automatically using hundreds of execution logs of dynamic updates over our corpus. For example, for the obstacle evasion, we programmatically verified that in the logs, the robot visits the specified *cond* cell and from then on does not visit the cells it should avoid according to $Spec_2$.

7.4 Experiment Setups and Results

Our corpus includes pairs of realizable specifications, where the second specification also includes a switching condition. For each pair we performed the following experiment. First, we constructed and executed a controller for the first specification in the pair. Then, we performed a dynamic-update with the second specification as the new specification. For each of the dynamic updates performed we measured the time it took to synthesize the new controller (the GR(1) just-in-time synthesis cost in Spectra), the time it took to synthesize the bridge controller (the cost that the dynamic update task adds), and the number of steps taken by the bridge controller when the update was executed.

We ran all experiments on an ordinary laptop PC, Intel Core i7 CPU 1.8GHz, 8GB RAM with Ubuntu 18.04 64-bit OS, Java 13 64Bit, and CUDD 3 compiled for 64Bit. Times we report are median values of 10 runs, measured by Java in milliseconds. Although the algorithms we deal with are deterministic, we performed 10 runs since JVM garbage collection and BDD dynamic-reordering add variance to running times.

To address RQ1 and RQ3(a), for each set of pairs, we report the median times we obtained. In Tbl. 1, Tbl. 2, and Tbl. 3, we report median values for dynamic updates between obstacle evasion specifications, between prioritized arbiter specifications, and between job scheduler specifications. For example, for the obstacle evasion configurations, for a grid of size 32×32 and number of safeties and justice guarantees $m = 7$ (i.e., for pairs(32, 7)), the median synthesis time of a new controller is 25.6 sec, the median bridge controller synthesis time is 23.5 sec, and the median optimized bridge controller synthesis time is 10.5 sec. Hence, in Tbl. 1, the cell on row '32×32' and column '7/new controller' reads 25.6, on row '32×32/No' and column '7/bridge' reads 23.5, and on row '32×32/Yes' and column '7/bridge' reads 10.5.

To address RQ2 and RQ3(b), for each performed dynamic update, we measured the actual bridge length as executed, i.e., the number of steps taken by the bridge controller. For each set of pairs, pairs(n, m), pairs(n), and pairs(n, k), in Tbl. 1, Tbl. 2, and Tbl. 3 respectively, we report the median actual length as executed. For example, for the obstacle evasion configuration pairs(64, 7), the median actual execution length of the non-optimized bridge is 27 steps, and of the optimized bridge is 11. Hence, in Tbl. 1, the cell on row '64×64/No' and column '7/length' reads 27, and on row '64×64/Yes' and column '7/length' reads 11.

To answer RQ1, we observe that the time that the dynamic update adds, i.e., synthesis time of the (non-optimized) bridge, is approximately of the same magnitude as the synthesis time of the new controller. In particular, this means that as long as the synthesis of $Spec_2$ is feasible, the dynamic update remains feasible as well.

To answer RQ2, the length of the actual bridges as executed is small, in particular orders of magnitudes shorter relative to the worst case length (see the last paragraph of Sect. 4.3). In the arbiter's case, bridge length is always 1 since prioritized requests are answered immediately and thus we are always at most a step away from a switching condition state.

To answer RQ3, we see that the optimization is efficient and effective. It reduces the bridge synthesis time and improves the quality of the update as it shortens its actual length. Moreover, the more challenging the case, the higher the factor of improvement.

7.5 Threats to Validity

We consider the following threats to the validity of our evaluation. First, our implementation may have bugs. To mitigate this, we performed extensive validation (see Sect. 7.3).

Second, running time measurements have variance due to the random generation of updates, the BDD libraries, and the Java garbage collector. To mitigate, we performed every experiment multiple times and report median values (see Sect. 7.4).

Third, running time measurements of our optimized version may be affected by the time between steps of the controller executor, which is configured in our testing environment. We used a constant step time for each set of specifications. Choosing a different step time could have changed our results. In reality, every system may have a different step time.

Fourth, we used only three sets of specifications with specific kinds of updates. We took the specifications from well-known benchmarks, but we do not know if these specifications and updates are representative of specifications and updates engineers will apply in practice. That said, we chose the obstacle evasion specification because it is very similar to the target application developed by Nokia Bell Labs, and we used hundreds of randomly generated updates involving changes in safeties and justices on all three sets of specifications.

Finally, dynamic update requires communication between the synthesizer and the executor. Our implementation of this communication is based on TCP. The communication time is strongly affected by the connection between the synthesizer and the executor. For example, a synthesizer and an executor that are connected via WiFi may communicate slower than a synthesizer and an executor that are running on the same machine. Thus, our results in this regard are implementation specific. One may obtain different results when using different means of communication.

m		3			5			7			9		
grid size	optimized?	new controller	bridge	length	new controller	bridge	length	new controller	bridge	length	new controller	bridge	length
8×8	No	0.3	0.9	8	0.5	1.3	8	0.7	1.4	8	1.0	1.6	8
	Yes		0.8	5		0.8	6		1.1	5		1.1	4
16×16	No	1.2	3.2	9	2.3	4.0	8	3.4	4.9	9	6.1	6.1	8
	Yes		2.1	6		3.1	6		3.7	5		4.9	5
32×32	No	4.9	12.8	15	13.9	16.2	12	25.6	23.5	14	41.8	30.3	13
	Yes		4.6	9		5.9	7		10.5	6		18.1	8
64×64	No	29.2	70.8	28	83.1	110.5	26	122.6	148.7	27	185.2	189.7	25
	Yes		18.1	15		33.7	11		61.8	11		71.8	11

Table 1: Obstacle evasion synthesis times (sec), bridge computation times (sec), and actual length as executed

#clients	optimized?	new controller	bridge	length
70	No	5.4	6.1	1
	Yes		2.9	1
80	No	7.7	12.5	1
	Yes		3.6	1
90	No	13.8	21.8	1
	Yes		5.5	1
100	No	24.7	34.5	2
	Yes		6.3	1

Table 2: Prioritized arbiter synthesis times (sec), bridge computation times (sec), and actual length as executed

8 RELATED WORK

Many authors have discussed dynamic adaptation in general and the dynamic-update problem in the context of synthesized controllers in particular, see, e.g., [4, 30, 40].

Baresi and Ghezzi [4] present a broad discussion on the motivation for dynamic updates, that is, for changes to occur as the software is running and while assuring continuous dependability. The combination of synthesis and dynamic updates goes in this direction.

Zhang and Cheng [40] present A-LTL, an approach to formally specify adaptation requirements in temporal logic. They consider three adaptation variants (one point, guided, and overlap). One may view our work as implementing a case of guided adaptation, where the synthesized update-strategy plays the role of the guide. In another work, the same authors [39] present a model-based approach to adaptation, separating the adaptation behavior and non-adaptive behavior specifications, and demonstrated using an adaptive GSM-oriented audio streaming protocol for a mobile computing application. The two works do not use synthesis for the construction of an update-strategy.

La Manna et al. [23] studied a notion of "updatable states". A state is updateable if the concatenation of *some* executions that lead into the state, with *any* future behavior induced by the new controller, satisfies the new specification. Their work is specific to assume-guarantee specifications given as universal Modal Sequence Diagrams (MSDs) [18]. They describe a related tool in [15] but do not provide an empirical evaluation or complexity analysis.

More recently, Nahabedian et al. [30] proposed a technique for dynamic update in the context of event-based controller synthesis. The work supports updates of labeled transition systems (LTSs) specified using LTL safeties and fluents [16]. It presents correctness criteria and a sound and complete solution. It considers various complexities of updates, e.g., planned updates, and allowing one to specify the requirements for a transition period where both the old and the new specification may not hold. It supports the specification of a transition requirement which is similar to our optional switching condition. It is implemented as an extension of the MTSA tool [9] and presents validation through several case studies.

Our work is distinctive in several ways, most notably (A) in its applicability and expressiveness - support for the GR(1) fragment of LTL, including safeties and justices, with a general computation model that is not specific to event-based specifications; and (B) in its efficiency and scalability - thanks to the symbolic nature of the algorithms and implementation. Note that the work in [30] uses concrete algorithms which in this sense do not scale. Indeed, the evaluation in [30] presents transition systems of up to 3000 states and maximum synthesis times of 5 minutes, while our evaluation uses controllers of up to about 120000 reachable states (from state spaces of 2^{20} to 2^{110} states) and synthesis times of up to 3 minutes but typically much less. Importantly, due to the difference in computation models, our main motivating example of obstacle evasion while patrolling between locations on a grid (and the other systems we present in our evaluation), cannot be modeled and solved using the approach of [30].

Livingston and Murray [22] suggested a dynamic-update technique for GR(1), restricted to adding or removing justice guarantees. Their aim is to use the existing controller to reduce the synthesis time of the modified specification, but their solution also works at runtime, and thus forms a restricted dynamic-update technique. Our work is not restricted to updates in justice guarantees, but covers the complete GR(1), including safeties and justices.

Another restricted dynamic-update technique for GR(1) is proposed by Shi et al. [37], who suggest techniques for several types

k		4			8			12			16		
# servers	optimized?	new controller	bridge	length	new controller	bridge	length	new controller	bridge	length	new controller	bridge	length
4	No	0.1	0.1	1	0.2	0.7	3	0.3	1.8	4	0.7	4.3	7
	Yes		0.1	1		0.4	3		1.2	5		2.1	4
6	No	0.2	0.5	1	1.0	4.7	2	2.0	8.7	3	4.9	19.8	5
	Yes		0.4	1		1.6	2		3.1	3		4.6	1
8	No	0.5	1.3	1	2.4	9.3	1	7.5	29.8	1	23.2	76.5	4
	Yes		0.8	1		2.5	1		9.0	1		15.6	1
10	No	1.3	2.6	1	5.9	34.8	1	24.6	59.8	2	56.8	226.4	4
	Yes		1.6	1		6.9	1		21.7	1		42.7	1

Table 3: Job scheduler synthesis times (sec), bridge computation times (sec), and actual length as executed

of controller synthesis and update in runtime, to overcome unexpected accidents caused, e.g., by a malfunction device. The work does not provide correctness criteria or an evaluation.

Finally, most recently, Finkbeiner et al. [12] present a form of a live update for controllers synthesized from LTL specifications, with a different semantics than ours. Moreover, due to the use of LTL, unlike our use of GR(1), their complexity of checking whether an update is possible is double exponential.

9 CONCLUSION

We formulated and solved the dynamic-update problem for synthesized GR(1) reactive systems: updating the behavior of an already running synthesized controller such that it would safely and dynamically, without stopping, start conforming to a modified, up-to-date specification. We formally defined the dynamic-update problem in the context of GR(1) and presented a sound and complete solution. The solution is based on the symbolic computation of an optimal bridge-controller, which guarantees a shortest switching phase, if one exists. Finally, we presented a heuristic optimization based on the early detection of a successful update, which may reduce both bridge synthesis times and actual bridge steps when executed.

We implemented our ideas in the Spectra synthesizer. Our evaluation shows the efficiency and effectiveness of our work. The work opens the way for the use of GR(1) synthesis in application domains where dynamic updates are a necessary requirement.

We suggest the following future work directions. First, we consider developing means to deal with the case of an unrealizable dynamic update, i.e., where switching cannot be forced.[7] Currently, our technique is limited to detecting unrealizable dynamic updates, but not to further deal with them. When the system cannot force a strategy update, it is still possible that an update can be done with the *cooperation* of the environment. That is, there may exist a sequence of inputs by the environment that enables a strategy switch. This calls for synthesizing a cooperative bridge-controller.

Second, we hope to complete the case study of deploying our implementation in Nokia Bell Labs system and report on our experience in a follow-up paper.

[7] Note that this is different than the case where the new specification is unrealizable by itself, which may be addressed in various ways, e.g., using cores, counter-strategies, or repairs [8, 21, 27].

ACKNOWLEDGEMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

REFERENCES

[1] [n.d.]. Spectra Website. http://smlab.cs.tau.ac.il/syntech/spectra/.
[2] [n.d.]. Supporting Materials Website. http://smlab.cs.tau.ac.il/syntech/dupdate/.
[3] [n.d.]. SYNTCOMP Website. http://www.syntcomp.org/.
[4] Luciano Baresi and Carlo Ghezzi. 2010. The disappearing boundary between development-time and run-time. In *Proc. of the Workshop on Future of Software Engineering Research (FoSER)*. ACM, 17–22. https://doi.org/10.1145/1882362.1882367
[5] Armin Biere, Keijo Heljanko, and Siert Wieringa. 2011. *AIGER 1.9 And Beyond*. Technical Report 11/2. Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria.
[6] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. 2010. RATSY - A New Requirements Analysis Tool with Synthesis. In *CAV (LNCS)*, Vol. 6174. Springer, 425–429. http://dx.doi.org/10.1007/978-3-642-14295-6_37
[7] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.* 78, 3 (2012), 911–938. http://dx.doi.org/10.1016/j.jcss.2011.08.007
[8] Davide G. Cavezza and Dalal Alrajeh. 2017. Interpolation-Based GR(1) Assumptions Refinement. In *TACAS (LNCS)*, Vol. 10205. 281–297. https://doi.org/10.1007/978-3-662-54577-5_16
[9] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. 2008. MTSA: The Modal Transition System Analyser. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 475–476. https://doi.org/10.1109/ASE.2008.78
[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *ICSE*. ACM, 411–420.
[11] Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) Synthesis. In *CAV (LNCS)*, Vol. 9780. Springer, 333–339.
[12] Bernd Finkbeiner, Felix Klein, and Niklas Metzger. 2021. Live Synthesis. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings (Lecture Notes in Computer Science)*, Zhe Hou and Vijay Ganesh (Eds.), Vol. 12971. Springer, 153–169. https://doi.org/10.1007/978-3-030-88885-5_11
[13] Elizabeth Firman, Shahar Maoz, and Jan Oliver Ringert. 2020. Performance heuristics for GR(1) synthesis and related algorithms. *Acta Informatica* 57, 1-2 (2020), 37–79. https://doi.org/10.1007/s00236-019-00351-9
[14] Enric Galceran and Marc Carreras. 2013. A survey on coverage path planning for robotics. *Robotics Auton. Syst.* 61, 12 (2013), 1258–1276. https://doi.org/10.1016/j.robot.2013.09.004
[15] Carlo Ghezzi, Joel Greenyer, and Valerio Panzica La Manna. 2012. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, 145–154. https://doi.org/10.1109/SEAMS.2012.

6224401

[16] Dimitra Giannakopoulou and Jeff Magee. 2003. Fluent model checking for event-based systems. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, Jukka Paakki and Paola Inverardi (Eds.). ACM, 257–266. https://doi.org/10.1145/940071.940106

[17] Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Lecture Notes in Computer Science, Vol. 2500. Springer. https://doi.org/10.1007/3-540-36387-4

[18] David Harel and Shahar Maoz. 2008. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling* 7, 2 (2008), 237–252. https://doi.org/10.1007/s10270-007-0054-z

[19] Dexter Kozen. 1982. Results on the Propositional μ-Calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK, UK, 348–359. http://dl.acm.org/citation.cfm?id=646236.682866

[20] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Trans. Robotics* 25, 6 (2009), 1370–1381. http://dx.doi.org/10.1109/TRO.2009.2030225

[21] Aviv Kuvent, Shahar Maoz, and Jan Oliver Ringert. 2017. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 362–372. https://doi.org/10.1145/3106237.3106240

[22] Scott C. Livingston and Richard M. Murray. 2014. Hot-swapping robot task goals in reactive formal synthesis. In *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. IEEE, 101–107. https://doi.org/10.1109/CDC.2014.7039366

[23] Valerio Panzica La Manna, Joel Greenyer, Carlo Ghezzi, and Christian Brenner. 2013. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, 63–72. https://doi.org/10.1109/SEAMS.2013.6595493

[24] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) synthesis for LTL specification patterns. In *ESEC/FSE*. ACM, 96–106. http://doi.acm.org/10.1145/2786805.2786824

[25] Shahar Maoz and Jan Oliver Ringert. 2015. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015 (EPTCS)*, Vol. 202. 58–72.

[26] Shahar Maoz and Jan Oliver Ringert. 2021. Spectra: a specification language for reactive systems. *Softw. Syst. Model.* 20, 5 (2021), 1553–1586. https://doi.org/10.1007/s10270-021-00868-z

[27] Shahar Maoz and Rafi Shalom. 2021. Unrealizable Cores for Reactive Systems Specifications. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 25–36. https://doi.org/10.1109/ICSE43902.2021.00016

[28] Shahar Maoz and Ilia Shevrin. 2020. Just-In-Time Reactive Synthesis. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 635–646. https://doi.org/10.1145/3324884.3416557

[29] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. 2019. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering (TSE)* (2019). https://doi.org/10.1109/TSE.2019.2945329

[30] Leandro Nahabedian, Victor Braberman, Nicolas D'Ippolito, Shinichi Honiden, Jeff Kramer, Kenji Tei, and Sebastian Uchitel. 2020. Dynamic update of discrete event controllers. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1220–1240.

[31] Daniel Neider and Ufuk Topcu. 2016. An Automaton Learning Approach to Solving Safety Games over Infinite Graphs. In *TACAS*. 204–221. https://doi.org/10.1007/978-3-662-49674-9_12

[32] Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. 2011. Distributed power allocation for vehicle management systems. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference, CDC-ECC 2011, Orlando, FL, USA, December 12-15, 2011*. IEEE, 4841–4848. https://doi.org/10.1109/CDC.2011.6161470

[33] Necmiye Ozay, Ufuk Topcu, Richard M. Murray, and Tichakorn Wongpiromsarn. 2011. Distributed Synthesis of Control Protocols for Smart Camera Networks. In *2011 IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, Chicago, Illinois, USA, 12-14 April, 2011*. IEEE Computer Society, 45–54. https://doi.org/10.1109/ICCPS.2011.22

[34] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[35] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL*. ACM Press, 179–190.

[36] Leonid Ryzhyk and Adam Walker. 2016. Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. (EPTCS)*, Ruzica Piskac and Rayna Dimitrova (Eds.), Vol. 229. 84–99. http://dx.doi.org/10.4204/EPTCS.229

[37] H. Shi, W. Dong, R. Li, and W. Liu. 2020. Controller Resynthesis for Multirobot System When Changes Happen. *Computer* 53, 12 (2020), 69–79. https://doi.org/10.1109/MC.2020.3017343

[38] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M. Murray. 2011. TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control (HSCC '11)*. ACM, New York, NY, USA, 313–314. https://doi.org/10.1145/1967701.1967747

[39] Ji Zhang and Betty H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 371–380. https://doi.org/10.1145/1134285.1134337

[40] Ji Zhang and Betty H. C. Cheng. 2006. Using temporal logic to specify adaptive program semantics. *J. Syst. Softw.* 79, 10 (2006), 1361–1369. https://doi.org/10.1016/j.jss.2006.02.062