Adapting Behaviors via Reactive Synthesis

Gal Amram[1], Suguman Bansal[2], Dror Fried[3], Lucas M. Tabajara[4], Moshe Y. Vardi[4], and Gera Weiss[5]

[1] Tel-Aviv University, Israel galam1483@gmail.com
[2] University of Pennsylvania, USA suguman@seas.upenn.edu
[3] The Open University of Israel, Israel dfried@openu.ac.il
[4] Rice University, USA {lucasmt,vardi}@rice.edu
[5] Ben-Gurion University of the Negev, Israel geraw@bgu.ac.il

Abstract. In the *Adapter Design Pattern*, a programmer implements a *Target* interface by constructing an *Adapter* that accesses an existing *Adaptee* code. In this work, we present a reactive synthesis interpretation to the adapter design pattern, wherein an algorithm takes an *Adaptee* and a *Target* transducers, and the aim is to synthesize an *Adapter* transducer that, when composed with the *Adaptee*, generates a behavior that is equivalent to the behavior of the *Target*. One use of such an algorithm is to synthesize controllers that achieve similar goals on different hardware platforms. While this problem can be solved with existing synthesis algorithms, current state-of-the-art tools fail to scale. To cope with the computational complexity of the problem, we introduce a special form of specification format, called *Separated GR(k)*, which can be solved with a scalable synthesis algorithm but still allows for a large set of realistic specifications. We solve the realizability and the synthesis problems for Separated GR(k), and show how to exploit the separated nature of our specification to construct better algorithms, in terms of time complexity, than known algorithms for GR(k) synthesis. We then describe a tool, called SGR(k), that we have implemented based on the above approach and show, by experimental evaluation, how our tool outperforms current state-of-the-art tools on various benchmarks and test-cases.

1 Introduction

Inspired by the well known adapter design pattern [18], we study the use of reactive synthesis for generating adapters that translate inputs meant for a target transducer to inputs of an adaptee transducer. Consider, as one motivating example, the practice of adding code to an operating system that mitigates the risk posed by newly discovered hardware vulnerabilities like Spectre and Meltdown [23, 26]. While the discovery of such vulnerabilities puts constraints on how the hardware can be used, the patch of the operating system (called adapter in this paper) takes upon itself to take care of running all applications without change [25]. It does so by allowing applications of the existing interface, while adapting their operation in way that ensures that the system is not exposed to the new threat.

Formally, we propose the following synthesis problem: given two finite-state transducers called *Target* and *Adaptee*, synthesize a finite-state transducer called *Adapter* such that

$$Adaptee \circ Adapter \approx Target.$$

The symbol $\circ$ stands for standard transducer composition and the symbol $\approx$ stands for an equivalence relation, a generalization of sequential equality, which we explain below. In words, we want an *Adapter* that stands between an *Adaptee* and its inputs and guarantees, such that the composition *Adaptee* $\circ$ *Adapter* is equivalent to *Target*. In the vulnerability patching example, *Adaptee* is a model of the constrained hardware and *Target* is a model of the hardware as used before the discovery of the vulnerability, without the new constraints. The *Adapter* that we generate models the patch that mediates between the vulnerable hardware and applications that are not aware of the vulnerability.

In our setting, an input to the synthesis algorithm is the equivalence relation along with the specification of the adaptee and of the target. While the problem of synthesizing an adapter such that *Adaptee* $\circ$ *Adapter* is sequentially equal to *Target* may be useful in some cases [32], we study here a more general problem. This is called for by applications such as the vulnerability covering patches described above. Specifically, we allow our users to specify an equivalence relation between *Adaptee* $\circ$ *Adapter* and *Target* that is not necessarily sequential equality. In this paper, we propose to use ω-regular properties [20] for specifying this equivalence relation, as follows. We assume, without loss of generality, that the outputs of both the *Target* and the *Adaptee* are assignments to disjoint sets of atomic propositions. We then consider sequences of pairs of such assignments that correspond to zipped runs of *Adaptee* $\circ$ *Adapter* and of *Target* over the same input. Having this set of sequences in mind, the user specifies a set of temporal properties using an ω-regular formalism such as LTL or Büchi automata. The transducer *Adaptee* $\circ$ *Adapter* is considered equivalent to *Target* if all the properties that the user specified are satisfied for each sequence in the set [19]. Note that the equivalence relation can be very different than sequential equality, it can, for example, say that *Adaptee* $\circ$ *Adapter* must be, in a way, a "mirror image" of *Target*, as demonstrated by the cleaning robots example in Section 4.1, where *Target* is a robot that cleans some rooms and *Adaptee* $\circ$ *Adapter* is a robot that clean all the rooms that *Target* did not clean.

The solution that we propose in this paper consists of two phases: we first transform the transducers to transition systems and arrive at a game structure that is more amenable for game-based techniques. Then we make use of the specific form of the resulting game and some simplifying assumptions about the form of the equivalence properties to solve the game efficiently. The game structures that we analyze consist of pairs of transition systems called *Input* and *Output*, accompanied by a set of ω-regular properties that specify equivalence relation between the two, as described above. The game that we solve is, then, to find a controller that reads the assignments to the variables of the *Input* and produces a valid sequence of assignments to the variables of the *Output* such that all the properties are satisfied. The translation of the transducers to this

game structure is rather direct, as elaborated in Section 4. The *Input* transition system is generated from the *Target* transducer and the *Output* transition system is generated from the *Adaptee* transducer. This is because we want the *Adapter*, which we generate from the controller as described below, to consider the behavior of the *Target* and to translate it to a command that generates an equivalent behaviour of *Adaptee*. Once we find a controller that solves the game, we can transform it to an *Adapter* as we detail in Section 4.

The synthesis problem that we defined so far is as hard computationally as general LTL synthesis and is thus double exponential in the worst case [37]. To cope with this difficulty, we propose to use a well known fragment of LTL called GR(k). GR(k) generalizes the GR(1) subset of LTL [9], a practical fragment of LTL for which a feasible reactive synthesis algorithm exists (see, e.g., [8, 28, 33]). Furthermore, GR(k) formulas are known to be highly expressive, as they can encode most commonly appearing LTL industrial patterns [15, 29, 30] and DBA properties (see related works for details). In addition to using GR(k), since the *Input* and *Output* in our model are separated transition systems, with separated sets of atomic propositions, we focus on properties that separate input and output variables. That is, our specification has the form $\bigwedge_{i=1}^{k}(\phi_i \rightarrow \psi_i)$, where the ϕ_i and ψ_i are conjunctions of LTL GF (Globally in the Future) formulas over *Input* variables only and *Output* variables only respectively. We call this model *Separated GR(k)*. We show through several case-studies that this fragment of LTL suffices to specify a range of useful equivalence relations.

We study the problems of realizability and synthesis on Separated GR(k) game. For that, we first consider a sub-problem of solving a *weak Büchi* game. Then we identify and make use of a property of separated games that we call *delay property*: the system can delay its response to the environment indefinitely as long as it remains in the same connected component of the game graph. This allows us to decide the realizability of Separated GR(k) in $O(|\varphi| + N)$ symbolic operations, and to synthesize a controller for a realizable specification in $O(|\varphi|N)$ symbolic operations, where φ is the Separated GR(k) specification, and N is the size of the state-space. Thus, Separated GR(k) games are easier to solve that solving GR(k) games which require $O(N^{k+1}k!)$ operations [35]. This demonstrates the efficiency of our framework, since $|\varphi|$ tends to be smaller than N and in most practical cases, $|\varphi| \in O(log(N))$.

The benefits of the complexity-theoretic improvement are reflected in empirical evaluations on our case studies of separated GR(k) formulas. We demonstrate that while separated GR(k) formulas are challenging for state-of-the-art synthesis tools, a symbolic BDD-based implementation of our algorithm solves them scalably and efficiently.

The rest of the paper is organized as follows: Section 2 introduces necessary preliminaries. Separated GR(k) games are introduced and formulated in Section 3. In Section 4 we describe how to use Separated GR(k) games synthesis to generate the adapter transducer, and introduce several use-cases. Next, we turn to solving separated GR(k) games. An overview of our solution approach and a necessary property for correctness of algorithm, called the delay property, is

given in Section 5. A complete symbolic algorithm is presented in Section 6. An empirical evaluation on case-studies is presented in Section 7. Finally, in Section 8 and Section 9 respectively, we give related work and conclude. Detailed proofs appear in the full version of the paper [3].

2 Preliminaries

General Definitions. Given a set of Boolean variables $\mathcal{V}$, a *state over* $\mathcal{V}$ is an assignment s to the variables in $\mathcal{V}$. We describe s as the subset of $\mathcal{V}$ that is assigned True in s. The set of *primed variables of* $\mathcal{V}$ is $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. Then $s' = \{v' \mid v \in s\}$ is the primed state s' over $\mathcal{V}'$. An *assertion over* $\mathcal{V}$ is a Boolean formula over variables $\mathcal{V}$. A state s satisfies an assertion ρ over the same variables, denoted $s \models \rho$, if ρ evaluates to True by assigning *true* to the elements of s. We define the *projection* of a state s on a subset $\mathcal{U} \subseteq \mathcal{V}$ as denoted by $s|_{\mathcal{U}} = s \cap \mathcal{U}$. We extend the notion of projection to a set of states $S \subseteq 2^{\mathcal{V}}$ by defining $S|_{\mathcal{U}} = \{s|_{\mathcal{U}} \mid s \in S\}$.

Our specification is a special form of *Linear Temporal Logic (*LTL*)*. LTL [36] extends propositional logic with infinite-horizon temporal operators. The syntax of an LTL formula over a finite set of Boolean variables $\mathcal{V}$ is defined as follows: $\varphi ::= v \in \mathcal{V} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi \mathsf{U} \varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi$. Here X (Next), U (Until), F (Eventually), G (Always) are temporal operators. The semantics of LTL can be found in [5, Chapter 5].

We model the adapters as transducers. A *transducer* is a deterministic finite-state machine with no accepting states, but with additional output alphabet and an additional function from the set of states to the output alphabet. A formal definition of a transducer is not required for this paper.

The algorithms developed in this paper are symbolic, i.e. manipulate implicit representations of sets of states. To this end, we use *Binary Decision Diagrams (BDDs)* [10] to represent assertions. For a BDD B and sets of variables $\mathcal{V}_1, \cdots \mathcal{V}_n$, we write $\mathsf{B}(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ to denote that B represents an assertion over $\mathcal{V}_1 \cup \cdots \cup \mathcal{V}_n$. For a state s over $\mathcal{V}$, we write $s \models B(\mathcal{V})$ to denote that the assertion that B represents is satisfied by the state s. BDDs support several *symbolic operations*: conjunction ($\vee$), disjunction ($\wedge$), negation ($\neg$), and extraction of variables using the $\exists$ and $\forall$ operators. We measure time complexity of a symbolic algorithm by a worst case #symbolic-operations it performs. A discussion on a rigorous treatment of BDD operations can be found in the paper's full version [3].

Game Structures and Games. We follow the notations of [9]. A game structure $GS = (\mathcal{I}, \mathcal{O}, \theta_{\mathcal{I}}, \theta_{\mathcal{O}}, \rho_{\mathcal{I}}, \rho_{\mathcal{O}})$ defines a turn-based interaction between an *environment* and a *system* players. The input variables $\mathcal{I}$ and output variables $\mathcal{O}$ are two disjoint sets of Boolean variables that are controlled by the environment and system, respectively. The environment's *initial assumption* $\theta_{\mathcal{I}}$ is an assertion over $\mathcal{I}$, and the system's *initial guarantee* $\theta_{\mathcal{O}}$ is an assertion over $\mathcal{I} \cup \mathcal{O}$. The environment's *safety assumption* $\rho_{\mathcal{I}}$ is an assertion over $\mathcal{I} \cup \mathcal{O} \cup \mathcal{I}'$, where the interpretation of $(i_0, o_0, i_1') \models \rho_{\mathcal{I}}$ is that from state (i_0, o_0) the environment

can assign i_1 to the input variables. W.l.o.g, we assume that $\rho_{\mathcal{I}}$ is deadlock free, i.e., for all (i_0, o_0) there exists an i_1 s.t. $(i_0, o_0, i_1') \models \rho_{\mathcal{I}}$. Similarly, the system's *safety guarantee* $\rho_{\mathcal{O}}$ is an assertion over $\mathcal{I} \cup \mathcal{O} \cup \mathcal{I}' \cup \mathcal{O}'$, where the interpretation of $(i_0, o_0, i_1', o_1') \models \rho_{\mathcal{O}}$ is that from state (i_0, o_0) when the environment assigns i_1 to the input variables, the system can assign o_1 to the output variables. Again, w.l.o.g, we assume that $\rho_{\mathcal{O}}$ is deadlock free, i.e., for all (i_0, o_0, i_1') there exists an o_1 s.t. $(i_0, o_0, i_1', o_1') \models \rho_{\mathcal{O}}$.

A play over GS progresses by the players taking turns to assign values to their own variables ad infinitum, where the players must satisfy the initial conditions at the start and the safety conditions thereafter. Formally, a *play* $\pi = s_0, s_1, \ldots$ is an infinite sequence of states over $\mathcal{I} \cup \mathcal{O}$ such that $s_0 \models \theta_{\mathcal{I}} \wedge \theta_{\mathcal{O}}$ and $(s_j, s_{j+1}') \models \rho_{\mathcal{I}} \wedge \rho_{\mathcal{O}}$ for all $j \geq 0$. A *play prefix* is either a play or a finite sequence of states that can be extended to a play. Then a *strategy* is a function $f : (2^{\mathcal{I} \cup \mathcal{O}})^+ \times 2^{\mathcal{I}} \rightarrow 2^{\mathcal{O}}$ such that if $s_0, \ldots, s_m$ is a play prefix, $(s_m, i') \models \rho_{\mathcal{I}}$ and $f(s_0, \ldots, s_m, i) = o$, then $(s_m, i', o') \models \rho_{\mathcal{O}}$. Intuitively, a strategy directs the system on what to assign to the output variables, depending on the history of a play and the most recent assignment by the environment to the input variables. A play prefix is said to be *consistent with a strategy* f if for all states $s_j = (i_j, o_j)$ in that prefix, $f(s_0, \ldots, s_{j-1}, i_j) = o_j$ for all $j \geq 0$. A strategy is memoryless if it only depends on the last state and the most recent assignment to the input variables. Formally, a *memoryless strategy* is a function $f : (2^{\mathcal{I} \cup \mathcal{O}}) \times 2^{\mathcal{I}} \rightarrow 2^{\mathcal{O}}$ such that if $(s_m, i') \models \rho_{\mathcal{I}}$ and $f(s_m, i') = o$, then $(s_m, i', o') \models \rho_{\mathcal{O}}$.

A *game* is a tuple (GS, φ) where GS is a game structure over inputs $\mathcal{I}$ and outputs $\mathcal{O}$ and φ is an LTL formula over $\mathcal{I} \cup \mathcal{O}$ called a *winning condition*. A play π is *winning* for the system if $\pi \models \varphi$. A strategy f *wins from state* s if every play π from s that is consistent with f is winning for the system. A strategy f *wins from* S, where S is an assertion over $\mathcal{I} \cup \mathcal{O}$, if it wins from every state $s \models S$. The *winning region* of the system is the set of states from which it has a winning strategy. A strategy f is *winning* if for every state $i \models \theta_{\mathcal{I}}$ there exists a state $o \in 2^{\mathcal{O}}$ such that $(i, o) \models \theta_{\mathcal{O}}$ and f wins from (i, o). In this paper, we have the following games that are defined over the following winning conditions.

- *Reachability games*: $\mathsf{F}(\varphi)$ where φ is an assertion over $\mathcal{I} \cup \mathcal{O}$.
- *Safety games*: $\mathsf{G}(\varphi)$ where φ is an assertion over $\mathcal{I} \cup \mathcal{O}$.
- *Büchi games*: $\mathsf{GF}(\varphi)$ where φ is an assertion over $\mathcal{I} \cup \mathcal{O}$.
- *GR(k) games*: $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{n_l} \mathsf{GF}(\varphi_{l,i}) \rightarrow \bigwedge_{j=1}^{m_l} \mathsf{GF}(\psi_{l,j}))$ where all $\varphi_{l,i}$ and $\psi_{l,j}$ are assertions over $\mathcal{I} \cup \mathcal{O}$.

Given a game (GS, φ), *realizability* is the problem of deciding whether a winning strategy for the system exists, and *synthesis* is the problem of constructing a winning strategy if one exists. We note that a realizability check can be reduced to the identification of the winning region, W: A winning strategy exists iff for all $i \models \theta_{\mathcal{I}}$ there exists $o \in 2^{\mathcal{O}}$ such that $(i, o) \models \theta_{\mathcal{O}}$ and $(i, o) \in W$. Hence, the synthesis problem can be solved by constructing a strategy that wins from W.

Game Graphs and Weak Büchi Games. The *game graph* for a game structure GS is the directed graph (V, E) with vertices $V = 2^{\mathcal{I} \cup \mathcal{O}}$ and edges

$E = \{(s,t) \mid (s,t') \models \rho_\mathcal{I} \wedge \rho_\mathcal{O}\}$. Intuitively, vertices are states over $\mathcal{I}$ and $\mathcal{O}$, and edges represent valid transitions between states according to the safety conditions. The game graph can be useful for analyzing the structural properties of a game structure via graph-theoretical properties.

A *finite path* in a directed graph (V, E) is a sequence $v_0, \dots, v_n \in V^+$ such that $(v_j, v_{j+1}) \in E$ for all $0 \leq j < n$. An *infinite path* $v_0, v_1, \dots \in V^\omega$ is similarly defined. A vertex u is said to be *reachable* from another vertex v if there is a finite path from v to u. A *strongly connected component* (SCC) of a directed graph (V, E) is a maximal set of vertices within which every vertex is reachable from every other vertex. It is well known that SCCs partition the set of vertices of a directed graph, and that the set of SCCs is partially ordered with respect to reachability. Also note that every infinite path ultimately stays in an SCC.

Let $(GS, \mathsf{GF}\varphi)$ be a game with a Büchi winning condition, and let $S_0 \dots, S_m$ be the set of SCCs that partition the game graph of GS. We say that $(GS, \mathsf{GF}\varphi)$ is a *weak Büchi game* if, given the set $\mathcal{F}$ of states that satisfy the assertion φ, for every SCC S_i, either $S_i \subseteq \mathcal{F}$ or $S_i \cap \mathcal{F} = \emptyset$. Thus, the SCCs of a weak Büchi game are either *accepting components*, meaning all of its states are contained in $\mathcal{F}$, or *non-accepting components*, meaning none of its states is present in $\mathcal{F}$. As a consequence, a play in a weak Büchi game is winning for the system if the play ultimately never exits an accepting component. Similarly, a strategy is winning for the system if it can guarantee that every play will ultimately remain inside an accepting component.

3 Separated GR(k) Games

Our framework relies on the core idea of reducing the problem of adapter generation to synthesizing a *Separated GR(k) game*, which we define in this section. At a high-level, a separated GR(k) differentiates from a regular GR(k) game in a separation between input and output variables in both the game structure and winning condition. We show in later sections that the separation of variables leads to algorithmic benefits to the synthesis problem. Formally we have the following.

Definition 1. *A game structure* $GS = (\mathcal{I}, \mathcal{O}, \theta_\mathcal{I}, \theta_\mathcal{O}, \rho_\mathcal{I}, \rho_\mathcal{O})$ separates variables *over input variables* $\mathcal{I}$ *and output variables* $\mathcal{O}$ *if:*

- *The environment's initial assumption* $\theta_\mathcal{I}$ *is an assertion over* $\mathcal{I}$ *only.*
- *The system's initial guarantees* $\theta_\mathcal{O}$ *is an assertion over* $\mathcal{O}$ *only.*
- *The environment's safety assumption* $\rho_\mathcal{I}$ *is an assertion over* $\mathcal{I} \cup \mathcal{I}'$ *only.*
- *The system's safety guarantee* $\rho_\mathcal{O}$ *is an assertion over* $\mathcal{O} \cup \mathcal{O}'$ *only.*

The interpretation of a game structure which separates variables is that the underlying game graph (V, E) is the product of two distinct directed graphs over disjoint sets of variables: $G_\mathcal{I}$ over the variables $\mathcal{I} \cup \mathcal{I}'$, and $G_\mathcal{O}$ over the variables $\mathcal{O} \cup \mathcal{O}'$. For $\mathcal{J} \in \{\mathcal{I}, \mathcal{O}\}$, the vertices of $G_\mathcal{J}$ correspond to states over $\mathcal{J}$ and there is an edge between states s and t if $(s, t') \models \rho_\mathcal{J}$.

Next, the notion of separation of variables extends to games with GR(k) winning conditions as follows:

Definition 2. *A GR(k) winning condition φ over $\mathcal{I} \cup \mathcal{O}$ separates variables w.r.t. $\mathcal{I}$ and $\mathcal{O}$ if $\varphi = \bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{n_l} \mathsf{GF}(\varphi_{l,i}) \rightarrow \bigwedge_{j=1}^{m_l} \mathsf{GF}(\psi_{l,j}))$ such that each $\varphi_{l,i}$ is an assertion over $\mathcal{I}$ and each $\psi_{l,j}$ is an assertion over $\mathcal{O}$.*

A *Separated GR(k) game* is a GR(k) game (GS, φ) over $\mathcal{I} \cup \mathcal{O}$ in which both GS and φ separate variables w.r.t. $\mathcal{I}$ and $\mathcal{O}$.

A major observation is that in a game played over a separated game structure, the actions of the two players are independent: the environment's actions do no limit the system's actions, and vice versa. In later sections we see how this observation leads to algorithmic improvements in solving separated GR(k) games over a regular GR(k) game. Specifically, in Section 4 we see how to use Separated GR(k) games to generate the adapter transducer. In Sections 5 and 6 we discuss algorithms for realizability and synthesis of Separated GR(k) games.

4 From Transducers to Separated GR(k)

We describe, using an end-to-end-example, how adapter transducer generation can be reduced to synthesis of Separated GR(k) games.

We begin with user-provided *Target* and *Adaptee* transducers. These transducers model the behavior of a system that we want to use (*Adaptee*) and the behavior of a system that we want to emulate (*Target*). For example, the transition systems in Figure 1 formulates the following scenario. (1) *Target* is an hardware interface that we want to support, such that the U (up) and the D (down) commands send the hardware from mode s_0 to modes s_1 and s_2, respectively, from which the S (stay) command keeps the system looping at the chosen mode. (2) *Adaptee* that is a hardware that we can use that also has three modes, but which does not allow the command S after U. Instead, it allows a D command that switches the mode back to s_0.

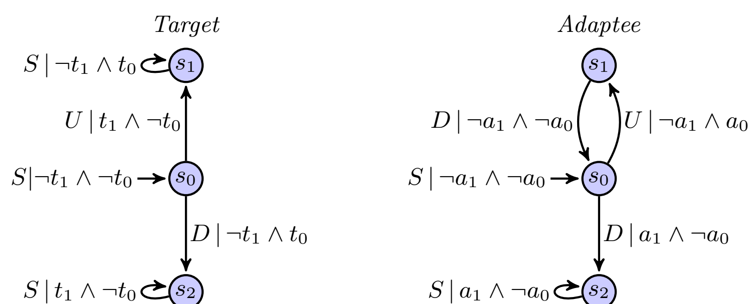


Fig. 1: An example of *Target* and *Adaptee* transducers. In this example, the t_i and a_i variables encode the binary representation of the mode being moved to.

The second step is a formulation of the equivalence relation, where we define the type of emulation that we require. In our example we want to maintain the following property: if *Target* visits a mode s_i infinitely often for a certain input sequence, then so does *Adaptee* ∘ *Adapter*. This can be expressed in LTL as:

$$\bigwedge_{i=0}^{2} \mathsf{GF}(bin_t(s_i)) \rightarrow \mathsf{GF}(bin_a(s_i))$$

where $bin_t(s_i)$ denotes the binary representation of mode s_i using variables t_1, t_0, and similarly for $bin_a(s_i)$ using variables a_1, a_0. Note that in this example we cannot just synthesize an adapter that cycles through all modes in *Adaptee* ∘ *Adapter* infinitely often, since the *Adaptee* transducer does not allow that.

As a third step, to generate a separated GR(k) game, we translate the *Target* and *Adaptee* transducers to *Input* and *Output* transition systems as depicted, for example, in Figure 2. Since *Adaptee* and *Target* are two separate transducers, each with its own structure, it is natural to model these as two separate transition systems on distinct variables. Thus, the transition systems are produced by the well known projection construction that turns an FST into a FSA that accepts the output language of the transducers [32]. Note that in our setting *Target* is translated to *Input* and *Adaptee* is translated to *Output*. This may appear as a role inversion to readers. We propose it because the role of the controller in our setting is to translate the behavior of *Target* to an equivalent behavior of the *Adaptee*.

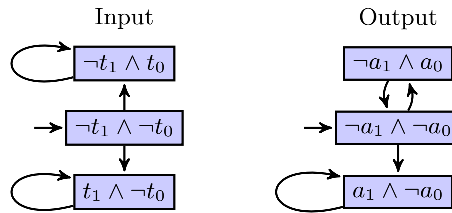


Fig. 2: A direct translation of the *Target* transducer to an *Input* transition system and of the *Adaptee* transducer to an *Output* transition system.

These separate transition systems, together with the specification described above, form a Separated GR(k) that, as a fourth step, we can feed to the Separated GR(k) synthesis algorithm. The output of the algorithm is a transducer called *Controller*, that maps runs of *Input* to runs of *Output*, as shown, in our example, in Figure 3. This, in fact, connects the output of the *Target* to the output of the *Adaptee*.

As a final step, from the controller we can construct the *Adapter* using the formula *Adapter* = *Adaptee*$^{-1}$ ∘ *Controller* ∘ *Target*. This means that *Adapter* contains an internal model of the *Target* and of the *Adaptee*. These internal

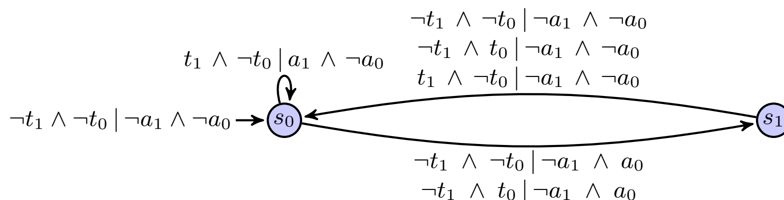


Fig. 3: A controller that reads runs of the *Input* transition system and generates runs of the *Output* transition system such that the specified Separated GR(2) formula is guaranteed to be true.

models are used to translate inputs to expected outputs of the adapter, then feed them to the controller, and then feed the output of the controller to the reverse of *Adaptee* to generate an input to *Adaptee* that emulates the behaviour of *Target*. Note that it is possible to invert transducers symbolically [21].

4.1 Additional Usages of our Technique

We give two more examples to demonstrate uses of Separated GR(k).

Cleaning Robots. This example demonstrates how one can use our technique to fulfill tasks that have not been covered by an execution of an existing transducer. Consider a cleaning robot (the *Target* transducer) that moves along a corridor-shaped house, from room 1 to room n. The robot follows some plan and accordingly cleans some of the rooms. Our goal is to synthesize a controller that activates a second cleaning robot (the *Adaptee* transducer) that follows the first robot and cleans exactly those rooms left uncleaned. Each robot controls a set of variables indicating which room they are in and which rooms they have cleaned, and additionally the original robot controls a variable indicating whether it is done with its cleaning. Our controller is required to fulfill requirements of the form: $\mathsf{GF}(\mathit{done}) \wedge \mathsf{GF}(!\mathit{in}{:}\mathit{clean}_i) \rightarrow \mathsf{GF}(\mathit{out}{:}\mathit{clean}_i)$, $\mathsf{GF}(\mathit{done}) \wedge \mathsf{GF}(\mathit{in}{:}\mathit{clean}_i) \rightarrow \mathsf{GF}(!\mathit{out}{:}\mathit{clean}_i)$.

Railway Signalling. This example demonstrates how one can use our technique to improve the quality of an existing transducer. We consider a junction of n railways, each equipped with a signal that can be turned on (light in green) or off (light in red). Some railways overlap and thus their signals cannot be turned on simultaneously. We consider an overlapping pattern where railways 1-4 overlap, and similarly 3-6, 5-8, and so on.

An existing system (the *Target* transducer) was programmed to be strictly safe in order to avoid accidents, so it never raises two signals simultaneously. We want to improve the system's performance by synthesizing a controller that reads the assignments that the existing transducer produces and accordingly assign values to the signals in such a way as to produce both safe and *maximal*

valuations: the ith signal is turned on if and only if the signal of every rail that overlaps with the ith rail is off. Furthermore, we want to maintain liveness properties of the *Target* system: (1) every signal that is turned on infinitely often by the existing system must be turned on infinitely often by the new system as well, and (2) if a signal is turned on at least once every m steps (where m is a parameter of the specification) by the existing system, then the same holds for the new system.

Note that, in terms of the GR(k) formula, this example is similar to the "hardware" example that we gave; we want to emulate the *Target*'s execution. The crux of the example lies in its *Adaptee*. Here, unlike in the explanatory example, the *Adaptee* is not a given hardware, but rather a virtual component that the user introduced to improve the *Target* performance. In this case the *Adaptee* produces safe and maximal signals.

5 Overview for Solving Separated GR(k) Games

The adapter generation framework described in Section 4 relies on synthesizing a controller from a separated GR(k) game. In this section and the next, we describe how to solve separated GR(k) games. This section gives an overview of the algorithm in Section 5.1 and describes a necessary property, called the delay property, in Section 5.2. The delay property is necessary to prove correctness of our synthesis algorithm. Later, Section 6 gives the complete algorithm and proves its correctness.

5.1 Algorithm Overview and Intuition

Following Section 3, we are given a Separated GR(k) game that consists of a game structure GS and a winning condition in a GR(k) form $\varphi = \bigwedge_{l=1}^{k} \varphi_l$, where $\varphi_l = \bigwedge_{i=1}^{n_l} \mathsf{GF}(a_{l,i}) \rightarrow \bigwedge_{j=1}^{m_l} \mathsf{GF}(g_{l,j})$. Let G be the game graph of GS. Consider an infinite play π in GS. Like every infinite path on a finite graph, π eventually stabilizes in an SCC S. Due to separation of variables, the game graph G can be decomposed into an input graph $G_{\mathcal{I}}$ and an output graph $G_{\mathcal{O}}$. Then the projection of S on the inputs is an SCC $S_{\mathcal{I}}$ in $G_{\mathcal{I}}$, and the projection of S on the outputs is an SCC $S_{\mathcal{O}}$ in $G_{\mathcal{O}}$. The input side of π converges to $S_{\mathcal{I}}$ whereas the output side π converges to $S_{\mathcal{O}}$.

Now, let S be an SCC with projections $S_{\mathcal{I}}$ on $G_{\mathcal{I}}$ and $S_{\mathcal{O}}$ on $G_{\mathcal{O}}$. Then we call S *accepting* if for *every* constraint φ_l, where $l \in \{1, \ldots, k\}$, one of the following holds:

All guarantees hold in S. For every $j \in \{1, \ldots, m_l\}$, there exists $o \in S_{\mathcal{O}}$ such that $o \models g_{l,j}$.

Some assumption cannot hold in S. There exists $j \in \{1, \ldots, n_l\}$ such that for all $i \in S_{\mathcal{I}}$, $i \not\models a_{l,j}$.

Then from the definition of an accepting SCC we have the following: a strategy that makes sure that every play converges to an accepting SCC, in which

all the relevant guarantee states are visited, is a winning strategy for the system in (GS, φ). To synthesize such a strategy, we do the following: (i) synthesize a strategy f_B for which every play converges to an accepting SCC; (ii) synthesize a strategy f_{travel} that travels within every accepting SCC, satisfying as many of the $g_{l,j}$ guarantees as possible. (iii) construct an overall winning strategy f that works as follows: the system plays f_B until reaching an accepting SCC S, then the system switches to f_{travel} to satisfy as many of the $g_{l,j}$ guarantees in S as possible; if the environment moves the play to a non-accepting SCC, the system can start playing f_B again to reach a different accepting SCC.

The strategy f_B can be found by synthesizing the weak Büchi game $(GS, \mathsf{GF}(acc))$, where acc is the assertion that accepts exactly those states that belong to accepting SCCs (note that $(GS, \mathsf{GF}(acc))$ is a well defined weak Büchi game). f_{travel} can be constructed by simply finding a path in $S_{\mathcal{O}}$ that satisfies the maximum number of guarantees.

A complication arises however when switching between f_{travel} and f_B, since it is conceivable that while the system is following f_{travel}, the environment could move to a different SCC that is outside of the winning region of f_B. Thus, it is not clear that we can combine these strategies to make an overall winning strategy for the system. To show that we can indeed combine both strategies, we need the following property that we call the *delay property*: if (i_1, o_1) is a state in the winning region of f_B, and (i_2, o_0) is a state for which there is a path in $G_{\mathcal{I}}$ from i_1 to i_2 and a path in $G_{\mathcal{O}}$ from o_0 to o_1, then (i_2, o_0) is also in the winning region of f_B. We formally state and prove the delay property in Section 5.2. In Section 6 we give details of the construction of f_B, f_{travel} and the use of the delay property to prove correctness of the overall winning strategy f.

5.2 The Delay Property

The delay property essentially says that if an SCC S is contained in the winning region, and the environment moves from S unilaterally to a different SCC S', then S' is also in the winning region of the system. In this section, we prove that the Büchi game $(GS, \mathsf{GF}(acc))$ where $GS = (\mathcal{I}, \mathcal{O}, \theta_{\mathcal{I}}, \theta_{\mathcal{O}}, \rho_{\mathcal{I}}, \rho_{\mathcal{O}})$, as defined in Section 5.1, satisfies the delay property. Throughout this section, we write $G_{\mathcal{I}}$ and $G_{\mathcal{O}}$ to denote the graphs over $2^{\mathcal{I}}$ and $2^{\mathcal{O}}$, respectively, as in Section 5.1. We start with the following lemma that states that the system can still win in spite of a single step delay.

Lemma 1. *Let $i_0, i_1 \in 2^{\mathcal{I}}$ such that $(i_0, i_1') \models \rho_{\mathcal{I}}$, and assume that the system can win from (i_0, o_0). Then the system can also win from (i_1, o_0).*

Proof. Let f be a winning strategy for the system from (i_0, o_0). We construct a winning strategy f_d from (i_1, o_0). Intuitively, f_d acts from state (i_1, o_0) as if it were following f from state (i_0, o_0), with a delay of a single step: the input in the current step is used to choose the output in the next step.

We use f to define f_d inductively over play prefixes of length $m \geq 1$, by setting $f_d((i_1, o_0), \ldots, (i_m, o_{m-1}), i_{m+1}) = f((i_0, o_0), \ldots, (i_{m-1}, o_{m-1}), i_m)$. Note that

f_d is well defined since GS separates variables: from state (i, o), the outputs that can be chosen for the successor state depend only on o, and not on i. Note that by this definition, for every play $(i_1, o_0), (i_2, o_1), \ldots, (i_{m+1}, o_m), \ldots$ consistent with f_d, the play $(i_0, o_0), (i_1, o_1), \ldots, (i_m, o_m), \ldots$ is consistent with f. We remark that we define f_d only for proving the lemma, and it is *not* part of our solution.

Next, we show that f_d is winning from (i_1, o_0). Take a play $(i_1, o_0), (i_2, o_1), \ldots,$ consistent with f_d. By the construction, $(i_0, o_0), (i_1, o_1), \ldots$ is consistent with f. Since this is a play on a weak Büchi game, after some point it must remain in a single SCC S, say from state (i_j, o_j). Since f is a winning strategy, the SCC S must be accepting. Then $o_j, o_{j+1}, \ldots$ is an infinite path in the SCC $S|_\mathcal{O}$, and $i_j, i_{j+1}, \ldots$ is an infinite path in the SCC $S|_\mathcal{I}$. Consequently, $(i_1, o_0), (i_2, o_1), \ldots$ converges to an SCC $\hat{S}$ in which $\hat{S}|_\mathcal{I} = S|_\mathcal{I}$ and $\hat{S}|_\mathcal{O} = S|_\mathcal{O}$. Since the conditions for an SCC D to be accepting depend only on the relation between $D|_\mathcal{I}$ and $D|_\mathcal{O}$, we have that $\hat{S}$ is accepting since S is accepting as well. □

We can now prove the delay property, following by straightforward induction from Lemma 1.

Theorem 1 (Delay Property Theorem). *Let $i_0, \ldots, i_n \in (2^\mathcal{I})^+$ be a path in $G_\mathcal{I}$, and for $m \geq 0$, let $o_{-m}, \ldots, o_0 \in (2^\mathcal{O})^+$ be a path in $G_\mathcal{O}$. Assume that the system can win from (i_0, o_0). Then the system can also win from (i_n, o_{-m}).*

Proof. From (i_n, o_{-m}), the system can simply ignore the inputs and follow the path in $G_\mathcal{O}$ to o_0. Let (i_{n+m}, o_0) be the state at that point in some play. Note that there is a path between i_n and i_{n+m}, and therefore there is a path between i_0 and i_{n+m}. If the system can win from (i_0, o_0) then by using Lemma 1 in the induction steps, the system can win by induction from (i, o_0) for all i such that there is a path in between i_0 and i. Therefore, the system can win from (i_{n+m}, o_0), and by consequence from (i_n, o_{-m}). □

A corollary of Theorem 1 is the following statement about the structure of the winning region of the weak Büchi game $B = (GS, \mathsf{GF}(acc))$ as defined in Section 5.1.

Corollary 1. *The winning region of B is a union of SCCs.*

Proof. Let (i, o) be a state in the winning region of B, let $(\hat{i}, \hat{o})$ be a state in the same SCC S of (i, o), and let $S|_\mathcal{I}$ and $S|_\mathcal{O}$ be the projections of S on $G_\mathcal{I}$ and $G_\mathcal{O}$, respectively. Then there is a path $i_0, \ldots, i_n$ for some $n \geq 0$ in $S|_\mathcal{I}$ such that $i_0 = i$ and $\hat{i} = i_n$. Similarly, there is a path $o_{-m}, \ldots, o_0$ for some $m \geq 0$ in $S|_\mathcal{O}$ such that $\hat{o}_0 = o$ and $\hat{o} = o_{-m}$. Then by the delay property of Theorem 1, the vertex $(\hat{i}, \hat{o}) = (i_n, o_{-m})$ is also in the winning region of B. □

We use Theorem 1 and Corollary 1 in the proof of correctness of the overall winning strategy f, as described in Section 6.2.

6 Algorithms for Solving Separated GR(k) Games

In this section we provide the exact details of our synthesis algorithm for Separated GR(k) games, as described in Section 5.1. Since constructing f_B involves defining and solving a weak Büchi game, we first describe these in Section 6.1. We remark that our weak Büchi game synthesis algorithm works for all weak Büchi games, and not just for the special weak Büchi game defined in Section 5.1. Specifically, it works even when the underlying game structure does not separates variables. Next, in Section 6.2, we complete the algorithm construction and describe the correctness of our overall synthesis algorithm.

6.1 Realizability and Synthesis for Weak Büchi Games

We present a symbolic algorithm to solve synthesis of a weak Büchi game. When represented in explicit state-representation, weak Büchi games are known to be solved in linear-time in the size of the game [12, 27]. In this section, we adapt the algorithm from [12, 27] to symbolic state-space representation. For sake of exposition, we give an overview of the algorithm and then present our symbolic modification.

Overview Given a weak Büchi game, recall that each SCC in its game graph G is either an accepting SCC or a non-accepting SCC. The goal is to find the winning regions in the weak Büchi game. This can be done by backward induction on the topological ordering of the SCCs as follows. Let $(S_0, \dots S_m)$ be a topological sort of the SCCs in G.

Base Case: Consider all *terminal partitions*, say $S_j, \dots, S_m$; that is, every SCC from which no other SCC is reachable. In this case, plays beginning in a terminal SCC will never leave it. Therefore, all states of terminal SCCs that are accepting are in the winning region of the system and all states of terminal SCCs that are non-accepting are not in the winning region of the environment.

Induction Step: Let $\vec{S} = (S_{i+1}, \dots, S_m)$, and suppose that the set $\bigcup \vec{S}$ has been classified into winning regions for the system W^s_{i+1} and the environment W^e_{i+1}, respectively. Let $\vec{S}_{new} = (S_j, S_{j+1}, \dots, S_i)$ be the SCCs from which all edges leaving the SCC lead to an SCC in $\vec{S}$. Further, let A and N be the unions of all accepting SCCs and all non-accepting SCCs in $\vec{S}_{new}$, respectively. Then the basic idea is as follows: The system can win from $s \in N$ if and only if it can force $\mathsf{F}(W^s_{i+1})$ from s. Analogously, the system can win from $s \in A$ if and only if it can force $\mathsf{G}(A \cup W^s_{i+1})$ from s. Hence, by solving these reachability and safety games, we can update W^s_{i+1} and W^e_{i+1} into W^s_j and W^s_j that partition the larger set $\bigcup(S_j, \dots, S_m)$ into winning regions for the system and the environment. The winning strategy can be constructed in a standard way as a side-product of the reachability and safety games in each step, see for example [40, 41].

Symbolic Algorithm for Weak Büchi Games. Given a weak Büchi game $B = ((\mathcal{I}, \mathcal{O}, \theta_\mathcal{I}, \theta_\mathcal{O}, \rho_\mathcal{I}, \rho_\mathcal{O}), \mathsf{GF}(acc))$ with BDDs representing $\theta_\mathcal{I}$, $\theta_\mathcal{O}$, $\rho_\mathcal{I}$, $\rho_\mathcal{O}$

and *acc*, our goal is to compute a BDD for the winning region and to synthesize a memoryless winning strategy for the system. The construction follows a fixed-point computation that adapts the inductive procedure described in the overview: In the basis of the fixed point computation, the winning region is the set of accepting terminal SCCs; in the inductive step, the winning region includes winning states by examining SCCs that are higher in the topological ordering on SCCs. In what follows we describe a sequence of BDDs that we construct towards constructing the overall BDD for the winning region.We use the notation X to denote a set of variables over $\mathcal{I} \cup \mathcal{O}$. For the sake of the current construction, memoryless strategies are given in the form of BDDs over X, X', for further details on the BDDs constructions see the full version for details [3].

BDD constructions. We start by constructing a BDD for a predicate that indicates whether two states in a game structure are present in the same SCC. Let predicate $\mathsf{Reach}(s, t')$ hold if there is a path from state s over $\mathcal{I} \cup \mathcal{O}$ to state t over $\mathcal{I} \cup \mathcal{O}$ in the game structure GS. Similarly, a predicate $\mathsf{Reach}^{-1}(s, t')$ holds if and only if $\mathsf{Reach}(t, s')$ holds. BDDs for Reach and Reach^{-1} can be computed in $O(N)$ symbolic operations using the transition relation of the game structure. Then, a BDD indicating if two states share the same SCC, is constructed in $O(N)$ symbolic operations by $\mathsf{SCC}(X, X') := \mathsf{Reach}(X, X') \wedge \mathsf{Reach}^{-1}(X, X')$.

Next, we construct a BDD for the union of the terminal SCCs, required by the basis of induction for the construction of the winning region. Let predicate $\mathsf{Terminal}(s)$ hold if state s over $\mathcal{I} \cup \mathcal{O}$ is present in a terminal SCC. Then $\mathsf{Terminal}(X) := \forall X' : \mathsf{Reach}(X, X') \rightarrow \mathsf{SCC}(X, X')$. Therefore, given BDDs for Reach and SCC, the construction of $\mathsf{Terminal}$ requires $O(1)$ symbolic operations.

Computing the Winning Region. We now describe the fixed-point computation to construct a BDD for the winning region in a weak Büchi game. Let $\mathsf{Reachability}_{(M,N)}(X)$ denote a BDD generated by solving a reachability game that takes as input a set of source states M and target states N and outputs those states in M from which the system can guarantee to move into N. Similarly, let $\mathsf{Safety}_{(M,N)}(X)$ denote a BDD generated by solving a safety game that takes as input a set of source states M and target states N and outputs those states in M from which the system can guarantee that all plays remain inside the set N. These constructions are standard, details can be found in [20, Chapter 2].

Now, let $\mathsf{Win}(s)$ denote that state s over $\mathcal{I} \cup \mathcal{O}$ is in the winning region. Then, $\mathsf{Win}(X)$ is the fixed point of the BDD $\mathsf{Win_Aux}$ defined below, where the construction essentially follows the high-level algorithm description. The BDD $\mathsf{Acc}(X)$ represents the formula *acc* encoding the set of accepting states. In addition, $\mathsf{DC}^i(X)$ is the union $\bigcup \vec{S}$ of the Downward-Closed set of SCCs, i.e. the SCCs that have already been classified into winning or not-winning, and $\mathsf{DC}^i_{new}(X)$is the union $\bigcup \vec{S}_{new}$ of the SCCs in $\mathsf{DC}^i(X)$ that were not in $\mathsf{DC}^{i-1}(X)$. Finally, $\mathsf{N}^i(X)$ is the subset N of non-accepting states in $\mathsf{DC}^i_{new}(X)$, and $\mathsf{A}^i(X)$ is the subset A of accepting states in $\mathsf{DC}^i_{new}(X)$. We then define $\mathsf{Win_Aux}$ as follows.

Base Case.

1: $\mathsf{Win_Aux}^0(X) := \mathsf{Terminal}(X) \land \mathsf{Acc}(X)$
2: $\mathsf{DC}^0(X) := \mathsf{Terminal}(X)$

Inductive Step.

1: $\mathsf{DC}^{i+1}(X) := \forall X' : \mathsf{Reach}(X, X') \to (\mathsf{SCC}(X, X') \lor \mathsf{DC}^i(X'))$
2: $\mathsf{DC}^{i+1}_{new}(X) := \mathsf{DC}^{i+1}(X) \setminus \mathsf{DC}^i(X)$
3: $\mathsf{N}^{i+1}(X) := \mathsf{DC}^{i+1}_{new}(X) \land \neg\mathsf{Acc}(X)$
4: $\mathsf{A}^{i+1}(X) := \mathsf{DC}^{i+1}_{new}(X) \land \mathsf{Acc}(X)$
5: $\mathsf{Win_Aux}^{i+1}(X) := \mathsf{Win_Aux}^i(X) \lor \mathsf{Reachability}_{(\mathsf{N}^{i+1}(X), \mathsf{Win_Aux}^i(X))}(X)$
$\lor \mathsf{Safety}_{(\mathsf{A}^{i+1}(X), \mathsf{A}^{i+1}(X) \lor \mathsf{Win_Aux}^i(X))}(X)$

To explain the construction of Win, note that a state s in $\mathsf{DC}^{i+1}(X)$ is winning in one of these cases: (i) s is a winning state in $\mathsf{DC}^i(X)$. (ii) s is a non-accepting state in $\mathsf{DC}^{i+1}(X)$ from which the system can force the play into a winning state in $\mathsf{DC}^i(X)$. This set of states can be obtained from $\mathsf{Reachability}_{(\mathsf{N}^{i+1}(X), \mathsf{Win_Aux}^i(X))}(X)$. (iii) s is an accepting state in $\mathsf{DC}^{i+1}(X)$ from which the system can guarantee that every play that leaves the accepting SCC moves into a winning state in $\mathsf{DC}^i(X)$. This set of states can be obtained from $\mathsf{Safety}_{(\mathsf{A}^{i+1}(X), \mathsf{A}^{i+1}(X) \lor \mathsf{Win_Aux}^i(X))}(X)$.

Finally, to check realizability, construct the BDD $\forall \mathcal{I}(\mathsf{InitIn}(\mathcal{I}) \to \exists \mathcal{O}(\mathsf{InitOut}(\mathcal{O}) \land \mathsf{Win}(\mathcal{I} \cup \mathcal{O})))$, where $\mathsf{InitIn}(\mathcal{I})$ and $\mathsf{InitOut}(\mathcal{O})$ are BDDs representing $\theta_\mathcal{I}$ and $\theta_\mathcal{O}$, respectively. This BDD is equal to $true$ iff B is realizable.

The fixed-point computation can be extended in a standard way to also compute a BDD representation $\mathsf{Fb}(X, X')$ of the winning strategy f_B, such that $(s, (i', o')) \models \mathsf{Fb}(X, X')$ iff $f_B(s, i) = o$, as we elaborate in the full version [3]. We then have the following theorem that follows our construction.

Theorem 2. *Realizability and synthesis for weak Büchi games can be done in $O(N)$ symbolic steps.*

Proof Outline. The proposed construction symbolically implements the inductive procedure of the explicit algorithm. Hence, it correctly identifies the system's winning region. It remains to show that the algorithm performs $O(N)$ symbolic operations. First of all, the constructions of SCC and $\mathsf{Terminal}$ take $O(N)$ symbolic operations collectively. It suffices to show that in the i-th induction step, solving the reachability and safety games performs $O(|\mathsf{DC}^{i+1} \setminus \mathsf{DC}^i|)$ operations. This can be proven by a careful analysis of the operations and the sizes of resulting BDDs using standard results on safety and reachability games. □

6.2 Realizability and Synthesis for Separated GR(k) Games

We finally make use of the elements obtained so far towards solving synthesis for Separated GR(k) games. Our construction follows the overview from Section 5.1. To recall, we describe and construct two auxiliary strategies f_B and f_{travel} and

combine them to generate the final strategy f. We use the delay property theorem from Section 5.2 to prove the correctness of our algorithm.

We are given a Separated GR(k) game structure $GS = (\mathcal{I}, \mathcal{O}, \theta_\mathcal{I}, \theta_\mathcal{O}, \rho_\mathcal{I}, \rho_\mathcal{O})$ and a winning condition $\varphi = \bigwedge_{l=1}^{k} \varphi_l$, where $\varphi_l = \bigwedge_{i=1}^{n_l} \mathsf{GF}(a_{l,i}) \rightarrow \bigwedge_{j=1}^{m_l} \mathsf{GF}(g_{l,j}))$. We first represent GS and φ as BDDs by standard means. We then define and construct the following.

Constructing f_B. Auxiliary strategy f_B is the winning strategy of the system player in a weak Büchi game constructed form the separated GR(k) game. To construct a weak Büchi game, we first construct, in $O(|\varphi| + N)$ symbolic operations, a BDD $\mathsf{Acc}(\mathcal{I} \cup \mathcal{O})$ that describes the set of accepting states. The construction is standard. Next, let acc be the assertion represented by Acc (the assertion defined in Section 5.1). Then the weak Büchi game is $B = (GS, \mathsf{GF}(acc))$. Finally, we construct f_B as the winning strategy of B, following Section 6.1.

Constructing f_{travel}. For the construction of f_{travel}, we arbitrarily order all guarantees that appear in our GR(k) formula: $gar_0, \ldots, gar_{m-1}$. For each guarantee gar_j, we construct a reachability strategy $f_{r(j)}$ that, when applied inside an SCC $S_\mathcal{O}$ in the output game graph $G_\mathcal{O}$, moves towards a state that satisfies gar_j without ever leaving $S_\mathcal{O}$. In case no such state exists in $S_\mathcal{O}$, $f_{r(j)}$ returns a distinguished value $\perp$. Note that this strategy can entirely ignore the inputs. We equip f_{travel} with a memory variable mem that stores values from $\{0, \ldots, m-1\}$. Then $f_{travel}(s, i)$ is operated as follows: for $mem, mem+1, \ldots$ we find the first $mem+j \pmod m$ such that the SCC of s includes a gar_j-state, and activate $f_{r(mem+j)}$ to reach such state. If no guarantees can be satisfied in S, we just return an arbitrary output to stay in $S_\mathcal{O}$. The construction of f_{travel} requires $O(|\varphi|N)$ symbolic BDD-operations as we need to construct m reachability strategies (clearly, $m \leq |\varphi|$).

Constructing the overall strategy f. Finally, we interleave the strategies f_B and f_{travel} into a single strategy f as follows: given a state s and an input i, if $s \models \mathsf{Acc}(X)$ (that is, if s is an accepting state), then set $f(s, i) = f_{travel}(s, i)$; otherwise set $f(s, i) = f_B(s, i)$. Whenever f switches from f_B to f_{travel}, the memory variable mem is reset to 0. The next lemma proves that if f_B is winning then so is f.

Lemma 2. *If f_B is a winning strategy for the weak Büchi game $B = (GS, \mathsf{GF}(acc))$, then f is a winning strategy for the Separated GR(k) game (GS, φ).*

Proof. Since f_B is a winning strategy, then for every initial input $i \models \theta_\mathcal{I}$ there is an initial output $o \models \theta_\mathcal{O}$ such that (i, o) is in the winning region of GS. We show that playing f always keeps the play in the winning region of GS, and therefore the play eventually converges to an accepting SCC. Once this happens, following f_{travel} guarantees that φ is satisfied. We know that as long as the play is in the winning region of B, following f_B will keep it inside the winning region.

Therefore, when we switch from f_B to f_{travel} we must be inside the winning region and, by definition of f, in some accepting SCC S. Then f_{travel} makes sure that as long as the environment remains in $S|_{\mathcal{I}}$, the projection of S over the inputs, the system remains in $S|_{\mathcal{O}}$, the projection of S over the output. Thus all in all the play remains in the winning region of S.

Therefore, the only way that the play can leave the winning region is if, when the system is in a state (i_0, o_0) and chooses some output o_{-m} according to f_{travel}, the environment chooses input i_n such that the play leaves S and moves to a state (i_n, o_{-m}) in a different SCC of G. Note, however, that in this case there is a path from i_0 to i_n and a path from o_{-m} to o_0 (since by construction f_{travel} remains in the same SCC in $G_{\mathcal{O}}$). Since (i_0, o_0) is in the winning region, by Theorem 1 we have that (i_n, o_{-m}) is in the winning region as well. □

Final Results. Given Lemma 2, we can obtain our final results on synthesis and realizability of Separated GR(k) games, as follows. Given a Separated GR(k) game (GS, φ), construct acc and solve the weak Büchi game $(GS, \mathsf{GF}(acc))$. Then construct f_B, f_{travel} and f as described above. If realizable, then f_B is a winning strategy and from Lemma 2 we have that f is a winning strategy for (GS, φ). If $(GS, \mathsf{GF}(acc))$ is unrealizable, then the environment can force every play to converge to a non-accepting SCC. Since the GR(k) winning condition cannot be satisfied from a non-accepting SCC, (GS, φ) is also not realizable. Thus we have the following theorem, see [3] for full details.

Theorem 3. *Realizability for separated GR(k) games can be reduced to realizability of weak Büchi games.*

The final result on solving Separated GR(k) games is then as follows, see [3] for full details.

Theorem 4. *Let (GS, φ) be a separated GR(k) game over the input/output variables $\mathcal{I}$ and $\mathcal{O}$, respectively. Then, the realizability and synthesis problems for (GS, φ) are solved in $O(|\varphi| + N)$ and $O(|\varphi| N)$ symbolic operations, respectively, where $N = |2^{\mathcal{I} \cup \mathcal{O}}|$.*

Proof Outline. Realizability and synthesis follow Lemma 2 and Theorem 3. It is left to analyze the number of symbolic operations for constructing f_B and then f. In symbolic operations, constructing acc takes $O(|\varphi| + N)$, and computing the winning region W for $(GS, \mathsf{GF}(acc))$ takes $O(N)$. Checking realizability can be done by checking if for every initial input i there is an initial output o such that $(i, o) \in W$, which takes $O(1)$. The winning strategy f_B can be computed in the process of computing W, taking the same number of operations (see [3] for details). Finally, constructing f_{travel} takes $O((\#gars)N) \leq O(|\varphi| N)$, where $gars$ are all guarantees $\mathsf{GF}(g_{i,\ell})$ that appear in φ. Therefore, constructing f takes $O(|\varphi| N)$ symbolic operations in total. □

Note that this result is an improvement over the complexity of synthesizing GR(k) games in general [35].

7 Implementation and Evaluation

We have implemented our Separated GR(k) framework for realizability and synthesis in a prototype tool SGR(k). The tool implements our symbolic algorithm using the CUDD [39] package for BDD manipulation. Our tool is evaluated on a suite of benchmarks created from the examples described in Section 4.

Benchmark Suite. We have created a suite of parametric benchmarks from the three examples described in Section 4. Our suite consists of 38 realizable specifications. The parametric versions of the examples are described here.

The *multi-mode hardware* example is a generalization of the example presented at the beginning of Section 4. It is parameterized by the number of bits n and has 2^n modes. The *Target* can move from mode 0 to any mode and stay there, while the *Adaptee* can only move from mode 0 to odd-numbered modes, and up and down between modes $2i$ and $2i+1$. The specification consists of $2n$ variables. We generate 10 such benchmarks with $n \in \{1, \dots, 10\}$.

The *cleaning robots* example is parameterized in the number of rooms. For a scenario with n rooms, the specification is written over $4n+1$ variables. We create 10 such benchmarks with $n \in \{1 \dots, 10\}$.

The *railways signalling* example consists of two parameters: a junction of n railways and the frequency parameter m. With parameters n and m, the specification consists of $(2+2\lceil \log m \rceil)n$ variables. We generate 18 benchmarks with $n \in \{2, \dots, 10\}$ and $m \in \{2, 3\}$.

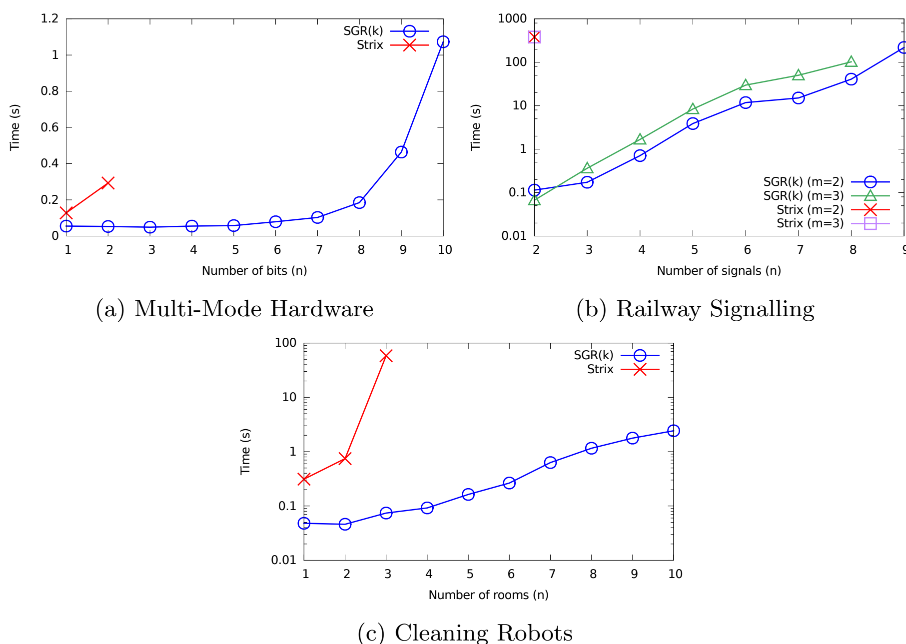
Experimental Setup and Methodology. We evaluate our tool against Strix [1, 31], the current state-of-the-art tool for LTL synthesis and SYNTCOMP 2020 winner of 3 out of 4 tracks [2]. In order to run our benchmarks on Strix, we transform the benchmarks (a game structure and a winning condition) into an LTL formula that characterizes the same winning plays using the strict semantics from [22]. To the best of our knowledge, there is no other synthesis/realizability tool that operates on GR(k) specifications.

We compare the running time for checking realizability. For this, we compare the running time of realizability checks of each benchmark on both tools. Every benchmark is tested 10 times on both tools. We do this to account for the randomness introduced during BDD construction due to the automatic variable ordering by CUDD. For each benchmark we evaluate (a) the number of executions on which the tools terminate and (b) the mean running time over 10 executions.

All experiments were executed on a single node of a high-performance computer cluster consisting of an Intel Xeon processor running at 2.6 GHz with 32GB of memory with a timeout of 10 mins.

Observations and Inferences. Our experiments clearly demonstrate the scalability and efficiency of our tool in solving Separated GR(k) formulas.

Figure 4 plots the mean running time for the three benchmarks. We further report the mean values in Table 1. The table rows refer to the benchmarks we



(a) Multi-Mode Hardware

(b) Railway Signalling

(c) Cleaning Robots

Fig. 4: Mean running time for different classes of benchmarks.

examine, and the columns refer to the value of the parameter n. As an example, for the specification Cleaning(3), SGR(k)'s mean running time is 0.07 sec. (row titled Cleaning(n);SGR(k), column titled 3) and Strix's mean realizability check running time is 58.3 sec. (row titled Cleaning(n);Strix), column titled 4). Cells reading 'TO' indicate experiments reached a timeout.

The results show that our tool solves a significantly larger number of benchmarks than Strix. On the few benchmarks which Strix solves, our tool outperforms it by several orders of magnitude. Although the running time may vary depending on the automatic variable ordering chosen by CUDD, we do not believe it

n		1	2	3	4	5	6	7	8	9	10
MultiMode(n)	SGR(k)	0.06	0.05	0.05	0.06	0.06	0.08	0.1	0.19	0.46	1.07
	Strix	0.13	0.29	TO	TO	TO	TO	TO	TO	TO	TO
Cleaning(n)	SGR(k)	0.05	0.05	0.07	0.09	0.16	0.26	0.63	1.16	1.78	2.43
	Strix	0.31	0.75	58.3	TO	TO	TO	TO	TO	TO	TO
Railways($n, 2$)	SGR(k)	-	0.11	0.17	0.71	3.88	11.8	15.1	40.8	219	TO
	Strix	-	382	TO	TO	TO	TO	TO	TO	TO	TO
Railways($n, 3$)	SGR(k)	-	0.07	0.36	1.67	8.39	29.8	50.3	102	TO	TO
	Strix	-	381	TO	TO	TO	TO	TO	TO	TO	TO

Table 1: Mean realizability check running times (sec.)

would vary enough to significantly change the results. Specifically, we calculated the 99% confidence interval for our results, and validated that for all data points our tool's entire interval lies below the entire interval for Strix.

Only three benchmarks were unsolvable by our tool (in the sense that the majority of the 10 executions timed out). The three benchmarks are the railway signal examples with $(n = 10, m = 2)$, $(n = 9, m = 3)$, and $(n = 10, m = 3)$. These benchmarks consist of a large number of variables (54, 40, and 60, respectively), making them particularly challenging. All executions of the remaining benchmarks were solved in less than 4 mins by our tool.

We also examined the number of solved executions per benchmark. Our tool solved all 10 executions for 35 out of 38 benchmarks. These are the 35 benchmarks that appear as solved in Figure 4. For the railway signalling benchmark with $(n = 10, m = 2)$, our tool solved 2 out of 10 executions. In contrast, Strix was not able to solve even one execution for 31 out of 38 benchmarks. Even increasing the timeout to 8hrs only allowed Strix to solve a single additional benchmark. In total, Strix and our tool verified realizability of 7 benchmarks and 36 out of 38 benchmarks, respectively. In summary, our experiments demonstrate that our tool is able to solve specifications which are challenging for existing tools.

8 Related Work

The Adapter design pattern was introduced in [18], and has been used in many software contexts since. Our interpretation of the pattern is inspired by automata based description of the pattern proposed by Pedrazzini [34]. We reformulated the problem as synthesis of reactive controllers that compose with existing systems to achieve a temporal specification, e.g. [7, 13, 17]. Note that our work differs from such frameworks in its variables separation feature. A work with a concept similar to adapting behaviors is the *Shield synthesis* that studies the problem in which a synthesized controller corrects safety violations of an existing controller [24]. Note that in contrast, our problem is mostly concerned about liveness adaptation.

Reactive synthesis of LTL winning conditions is 2EXPTIME complete in the size of the formula [37], making it difficult to scale for applications. An approach to overcome the computational barrier has been to investigate fragments and variants of LTL with lower complexity for synthesis [4, 14, 16]. One such fragment is GR(k) [9], that offers a balance between efficiency and expressiveness. Specifically, GR(k) games are known to be efficient as they are solved in exponential time in the number of conjunctions k rather than exponential in the state-space [35]. Several studies have also shown that GR(k) specifications are highly expressive. As evidence, all properties expressed by deterministic Büchi automata (DBA) can be expressed in GR(k) [16], where a study of commonly appearing LTL patterns has shown that 52 of 55 patterns are DBA properties [15, 29]. DBA properties have also been identified as common patterns in robotics applications [30].

Finally, Separated GR(k) games exhibit the *delay property*, which intuitively means that the system can win even after delaying its action for a finite amount of time while ignoring the environment before "catching up" with the environment. While this is reminiscent of asynchrony in reactive systems [6, 38], a further exploration of relations between asynchrony and the delay property is required.

9 Conclusion

This paper presents a reactive systems-based model of the adapter design pattern. We model the adapters as transducers and reduce the problem of finding an *Adapter* transducer for a given *Adaptee* and *Target* systems, to the problem of synthesizing strategies for Separated GR(k) games. Through an analysis of theoretical complexity and algorithmic performance, we show that realizability and synthesis of Separated GR(k) games is efficient and scalable. Furthermore, by outperforming Strix, an existing state-of-the-art synthesis tool, we show that algorithms for the Separated GR(k) class of specifications add value to the portfolio of reactive synthesis tools.

The benefits of separation of input and output variables were previously shown in the context of Boolean Functional Synthesis [11]. Through this work, we showed that separation also leads to practically viable solutions in temporal reactive synthesis, specifically when encoding the types of equivalence relations that appear in reactive adaptation (where properties of runs of the first system are compared to properties of runs of the other). Since the systems may be loosely coupled, i.e., they may not run on the same clock, specifications that impose joint temporal constraints on the two systems may not be realizable. Thus, our proposition to use the type of equivalence that separated GR(k) formulas allow, gives users the power needed for comparing the *overall behaviors* of the systems while allowing realizability and efficient synthesis.

The results presented in this paper encourage future studies on the separation of variables in a broader context. For instance, reason about variants of the adapter design pattern that do not separate variables all the way through. That is to say, variants that translate to more general GR(k) specifications in which the separation appears in the input and output systems but not in the specification itself. One could further study the notion of separation of variables in more the general LTL specifications. Another direction is to consider systems that gets two types of input: from the input system (i.e. the *Target*) as well as from an environment. We believe that these future directions would enable the development of tools for synthesis from temporal specifications with a focus on expressing practical applications as well as ensuring scalability and efficiency.

Acknowledgements We thank Supratik Chakraborty and Dana Fisman for useful comments. Work is supported in part by NSF grant 2030859 (CRA's CIFellows Project), NSF grants IIS-1527668, CCF-1704883, IIS-1830549, an award from the Maryland Procurement Office, ISF grant 2714/19, and by the Lynn and William Frankel Center for Computer Science.

Bibliography

[1] Strix Website. https://strix.model.in.tum.de/

[2] The Reactive Synthesis Competition - SYNTCOMP 2020 Results. http://www.syntcomp.org/syntcomp-2020-results/

[3] Amram, G., Bansal, S., Fried, D., Tabajara, L.M., Vardi, M.Y., Weiss, G.: Adapting Behaviors via Reactive Synthesis. CoRR **abs/2105.13837** (2021), http://arxiv.org/abs/2105.13837

[4] Amram, G., Maoz, S., Pistiner, O.: GR(1)*: GR(1) specifications extended with existential guarantees. In: Proc. of FM (2019)

[5] Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)

[6] Bansal, S., Namjoshi, K.S., Sa'ar, Y.: Synthesis of asynchronous reactive programs from temporal specifications. In: Proc. of CAV (2018)

[7] Bansal, S., Namjoshi, K.S., Sa'ar, Y.: Synthesis of coordination programs from linear temporal specifications. Proc. of POPL (2019)

[8] Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In: Proc. of DATE (2007)

[9] Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)

[10] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)

[11] Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: Proc. of FMCAD (2018)

[12] Chatterjee, K.: Linear time algorithm for weak parity games. CoRR **abs/0805.1391** (2008), http://arxiv.org/abs/0805.1391

[13] Ciolek, D., Braberman, V., D'Ippolito, N., Piterman, N., Uchitel, S.: Interaction models and automated control under partial observable environments. IEEE Transactions on Software Engineering **43**(1), 19–33 (2016)

[14] De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proc. of IJCAI (2013)

[15] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. of ICSE (1999)

[16] Ehlers, R.: Generalized rabin(1) synthesis with applications to robust system synthesis. In: Proc. of NASA-FM (2011)

[17] Fried, D., Legay, A., Ouaknine, J., Vardi, M.Y.: Sequential relational decomposition. In: Proc. of LICS (2018)

[18] Gamma, E.: Design patterns: elements of reusable object-oriented software. Pearson Education India (1995)

[19] Grabmayer, C., Endrullis, J., Hendriks, D., Klop, J.W., Moss, L.S.: Automatic sequences and zip-specifications. In: Proc. of LICS (2012)

[20] Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research (2002)

[21] Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. SIGPLAN Not. p. 376–389 (Jun 2017)
[22] Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Proc. of SYNT (2016)
[23] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: Proc. of (S&P'19) (2019)
[24] Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L.R., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. Formal Methods Syst. Des. **51**(2), 332–361 (2017)
[25] Koruyeh, E.M., Khasawneh, K.N., Song, C., Abu-Ghazaleh, N.: Spectre returns! speculation attacks using the return stack buffer. In: Proc. of USENIX (2018)
[26] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Proc. of USENIX (2018)
[27] Löding, C., Thomas, W.: Alternating automata and logics over infinite words. In: Proc. of IFIP. vol. 1872, pp. 521–535 (2000)
[28] Maoz, S., Ringert, J.O.: Synthesizing a lego forklift controller in GR(1): A case study. In: Proc. of SYNT (2015)
[29] Maoz, S., Ringert, J.O.: GR(1) synthesis for LTL specification patterns. In: Proc. of ESEC/FSE (2016)
[30] Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. IEEE Transactions on Software Engineering pp. 1–1 (2019)
[31] Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Proc. of CAV (2018)
[32] Mohri, M.: Finite-state transducers in language and speech processing. Comput. Linguist. **23**(2), 269–311 (Jun 1997)
[33] Ozay, N., Topcu, U., Murray, R.M.: Distributed power allocation for vehicle management systems. In: Proc. of CDC-ECC (2011)
[34] Pedrazzini, S.: The finite state automata's design patterns. In: Automata Implementation. pp. 213–219 (1999)
[35] Piterman, N., Pnueli, A.: Faster solutions of rabin and streett games. In: Proc. of LICS (2006)
[36] Pnueli, A.: The temporal logic of programs. In: Proc. of FOCS (1977)
[37] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. of POPL (1989)
[38] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Proc. of ICALP (1989)
[39] Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. `http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf` (2015)
[40] Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety ltl synthesis. In: Proc. of HVC (2017)
[41] Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic ltlf synthesis. In: Proc. of IJCAI (2017)