Reactive Synthesis of Graphical User Interface Glue Code*

Rüdiger Ehlers[1] and Keerthi Adabala[2]

[1] Clausthal University of Technology, Clausthal-Zellerfeld, Germany
ruediger.ehlers@tu-clausthal.de
[2] University of Bremen, Bremen, Germany
adabala@uni-bremen.de

Abstract. We present an approach to synthesize *glue code* for graphical user interfaces. Such code starts computation and I/O threads in response to user interface events and changes the state of the interface according to the interaction scheme envisioned by the UI designer.
Our approach integrates several ideas that work best in combination. For instance, by translating all specification parts to universal very-weak (UVW) automata and building a game from them, we obtain a natural order over the positions in the game that enables us to prune the game graph substantially while constructing it. Furthermore, we present an approach to compute *kind* strategies that constrain the environment as little as possible and hence make the UIs as responsive as possible. The use of UVWs gives rise to a simple formalization of this idea.
We apply our approach to a case study with an Android (cell phone) application and show experimentally that previous reactive synthesis tools are unable to synthesize controllers for this application.

1 Introduction

The large number of ways in which a user of a program can interact with its graphical user interface (GUI) makes writing code for such interfaces highly difficult. Events in user interfaces trigger computation or I/O threads that can run concurrently while the user interface needs to remain responsive and to service requests that are unconnected to the computation or I/O taking place in the background. Unsurprisingly, user interface *glue code* that reacts to events, changes the state of the user interface, and triggers computation or I/O is difficult and tedious to write and thus susceptible to bugs.

User interfaces often go through several iterations of user testing before they are finalized. While programs for designing the graphical appearance of user interfaces exist, how the system to be developed behaves in response to events is still normally implemented manually, which leaves the implementation burden for every iteration to the UI designer and the user experience (UX) engineer, which makes experimenting with different interaction styles difficult.

* This work was supported by the German Science Foundation (DFG) under Grant No. 322591867.

Automatically *synthesizing* GUI code from user interface interaction specifications has the potential to solve this problem. It is a special case of *reactive synthesis*, where a system that continuously interacts with its environment is computed that satisfies its specification for all possible input sequences. The specification is typically given in some form of temporal logic, such as *linear temporal logic* (LTL). Despite the provably high computational complexity of this problem for most specification logics, practical reactive synthesis has seen a substantial improvement in the last few years, with tools such as *Strix* [15] winning the *2018 Reactive Synthesis Competition* for LTL specifications.

Unfortunately, classical reactive synthesis frameworks are not well-applicable to synthesize GUI glue code:

1. In synthesis from LTL, the system is assumed to read input bit values and set output bit values in every time step. There is no fixed time step in user interfaces – rather, the controller can respond to *events* by executing a sequence of *actions*. These cannot happen in parallel, and there can also be a *last* event after which the system and the controller stall.
2. Specifications for user interfaces are normally huge, requiring a very high scalability of the synthesis approach, which traditional approaches for synthesis from LTL specifications do not provide.
3. Synthesis approaches that trade the full expressivity of LTL against improved efficiency, such as *Generalized Reactivity(1) Synthesis* [7], cannot deal with specifications parts that describe chains of events, which are common in user interface specifications.
4. The application-specific quality metrics for user interface glue code, such as starting computation as quickly as possible and enabling UI elements such as buttons whenever possible (but only then) cannot be accurately captured by traditional quality metrics in reactive synthesis, such as maximizing pay-offs in games [4].

Nevertheless, GUI glue code is an application that is quite a natural fit for reactive synthesis: there is a clear notion of state in user interfaces, the lower bound on the complexity of all interesting problems for LTL induced by the inclusion of propositional logic does not apply (as events cannot happen in parallel in GUIs), and the rapid prototyping cycles during UI/UX design provide a clear motivation for employing synthesis technology.

In this paper, we present an approach to perform reactive synthesis of user interface glue code. We carefully selected and devised components for a synthesis approach that in combination avoid the four drawbacks of classical synthesis algorithms stated above and hence enable the synthesis of GUI glue code. Our approach builds on the following ideas:

- Rather than assuming an execution in fixed timesteps, we use an interaction semantics in which the GUI controller can react to every input event with arbitrary long but finite sequences of output actions.
- We use an LTL fragment that can easily be translated to *universal very weak* automata as specification language [1].

– We reuse the main ideas of the efficient *Generalized Reactivity(1) Synthesis* approach to solve the *synthesis games* built from environment assumptions and system guarantees in this LTL fragment.
– Contrary to how Generalized Reactivity(1) Synthesis was used with binary decision diagrams (BDD) in earlier works, we perform explicit-state game solving, where only those game positions are explored that form the *anti-chain* of incomparable best-case reactions by the system.
– After solving the game, we constrain the strategy to play action sequences that restrict the environment as little as possible, hence making the UIs as responsive as possible.

These design decisions work together in concert. For instance, the use of universal very-weak automata enables us to define a natural order over the obligations of the system player on the further play of the game. This allows the system player to chose its action sequences in a way that all successor positions are optimal with respect to this order, which minimizes the explicit-state size of the game graph. We show that in this way, the synthesis problem for GUI glue code becomes much more tractable than with earlier synthesis approaches. At the same time, the supported specification class is powerful enough to capture user interface interaction rules.

We present the overall methodology in this paper and highlight the insights that led to the choice of its components. We apply our methodology to a case study for a *cost splitting* application for Google Android cell phones on which we demonstrate the scalability of our synthesis approach.

1.1 Related work

Reactive synthesis is an classical topic in the formal methods literature. While the identification of the problem as doubly exponential-time complete for logics such as linear temporal logic (LTL) [16] implied that the problem is intractable in theory, more recently, optimizations that try to make use of the structure of specifications of interest in practice have emerged. Of particular interest in this context is the *Generalized Reactivity(1) Synthesis* approach [7], which trades the full expressivity of LTL against a reduction of the synthesis complexity. The approach, abbreviated as *GR(1) synthesis*, has been shown to be expressive enough for many applications in robotics [13] and device driver synthesis [17]. Just like it is the case for GR(1) synthesis, the approach we present in this paper lies at a *sweet spot* between expressivity and efficiency for interesting application domains.

We focus on the domain of user interfaces. Experience shows that getting them correct is difficult [14], which suggests that more formal approaches to developing user interface code may be useful. Model-driven engineering of user interfaces helps to mitigate this problem [12] and builds on iterative refinement of the model. With the present work, we aim for a less disruptive approach to change the way GUI glue code is engineered. We only require the UI/UX designers to specify the behavior in a form of temporal logic, without the need to model

any other representation of the interaction or perform any form of refinement, hence allowing for very quick revision cycles of the interaction scheme.

While classical reactive synthesis uses environment and system propositions to model the interaction of a system with its environment, GUIs communicate with their environments via *events*, of which only one can happen at a time. Synthesis of event-based systems has been considered previously [8], but incorporating the idea that the system can react with a sequence of actions to input events, as it is the case in GUIs and exploited in our synthesis algorithm to simplify the synthesis problem, appears to be novel.

When synthesizing an implementation, there is normally an infinite number of candidate implementations to choose from. In such a case, implementations that do not prevent the environment from fulfilling the assumptions made about it in the specification are commonly preferred [8, 6]. Furthermore, if a specification comes with a *cost metric* for performing actions, we may be interested in cost-optimal controllers (see, e.g., [4]). In contrast, the *kindness* definition for GUI glue code given in Section 6 is a domain-specific optimization objective that builds on the representation of the specification using very-weak automata.

2 Running Example: `SplitExpenses`

We consider a *cost splitting* application for Android cell phones. With the application, an (informal) team can split its running costs. Team members paying for expenses can add their expenses to a list. The application then gives an overview of who owes how much money to make the split fair. The application uses a server to synchronize the data, and it supports being a member of multiple teams. The application has several views of which one is visible at every point in time:

- Account/team selection
- New team/account
- Overview of who owes how much money
- List of expenses
- Administrator's panel for removing unjustified expense items
- Adding an expense
- Recording reimbursements within the group
- Expenses not yet confirmed by the current user
- List of actions not yet executed due to a missing internet connection

A new group is started by an administrator, who forwards team login codes to other team members. If a user does not have an internet connection, changes performed by the user are not dropped, but rather executed when the connection is restored. So offline use is possible with delayed synchronization. The application has three threads for creating a new team, receiving team data from the server, and submitting changes to the server.

The overall specification for the GUI glue code consists of 118 individual properties (expressed in an LTL fragment that permits an efficient translation to universal very-weak automata [1]). There are, in addition, 8 properties that express assumptions about the environment, such as that all threads eventually terminate and that disabled buttons cannot be clicked by the user.

3 Preliminaries

Basics: Given a finite set X, the set of finite words over X is denoted as X^*, while the set of infinite words is denoted as X^ω.

Linear temporal logic (LTL): Let AP be a finite set of *atomic propositions*. An LTL formula describes a specification over infinite *traces* with trace elements in 2^{AP}. Syntactically, LTL formulas are built using the following grammar:

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi' \mid \psi \wedge \psi' \mid \mathsf{X}\psi \mid \mathsf{G}\psi \mid \mathsf{F}\psi \mid \psi\,\mathcal{U}\,\psi' \mid \psi\,\mathcal{R}\,\psi' \mid \psi\,\mathcal{W}\,\psi'$$

An LTL formula holds at a position in a trace or not. By default, the first element of the trace is looked at. A formal semantics of LTL can be found in [3].

Automata over infinite words: Let Σ be a finite alphabet. An ω-automaton $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ over Σ is a tuple consisting of a set of states Q, the set of initial states Q_0, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, and the set of *final states* F. An infinite word $w = w_0 w_1 \ldots \in \Sigma^\omega$ induces an infinite run $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ in $\mathcal{A}$ if we have $\pi_0 \in Q_0$ and for every $i \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$. Finite runs are defined similarly, where we require that they cannot be further extended. Automata reject or accept words w depending on which runs w induces. The set of words accepted by an automaton is called its *language*. A *universal co-Büchi automaton* accepts all words for which no infinite run exists that visits states in F infinitely often, i.e., for which there are infinitely many $i \in \mathbb{N}$ with $\pi_i \in F$. We also call F the set of *rejecting states* in this paper. We will only be concerned with a subclass of these automata here, which are called *universal very-weak* (or *one-weak*) automata. These are universal co-Büchi automata for which every loop is a self-loop. This requirement can be formalized by stating that there should exist a *leveling* function $l : Q \to \mathbb{N}$ such that for every $(q, x, q') \in \delta$, we have $l(q') \geq l(q)$. For many (but not all) LTL formulas, there exists a universal very-weak (UVW) automaton whose language is the set of traces that satisfies the LTL formula. Pointers to the literature discussing this topic are given in [1]. We say that a UVW state q is left along all runs for the last character of a word $w_0 \ldots w_n$ if $(q, w_n, q) \notin \delta$ or there is no run that is in state q after the prefix word $w_0 \ldots w_{n-1}$.

4 An Execution Semantics for GUI Glue Code

Traditional temporal logics either use a *real-time semantics* in which the passing of time can be reasoned about (such as in *Metric Time Logic* [2]), or assume a discrete-time semantics with regular steps in the execution of a system (such as LTL). For synthesizing GUI glue code, neither of these choices is satisfactory. User interfaces should normally be *patient* and give the user time to react, so that the concrete timing of the interaction with a user should not matter. Given that even for simple real-time temporal logics such as *Metric Interval Time Logic*, the reactive synthesis problem is undecidable [9], real-time logics appear

to be an unsuitable choice. Linear temporal logic (LTL) synthesis on the other hand has been conceptualized for settings with evenly distributed time steps, for which the abstraction that a system runs infinitely long is reasonable, and for which in every step of the system's execution, all input and output propositions of a system have values. This is also not the case for GUIs, as all events such as clicking a button and starting a computation thread are ordered, so that no two events can happen at the same time. The assumption that the system always runs infinitely long is also unreasonable: GUI code always eventually returns control to the operating system (OS) and then only *wakes up* if and when an external event happens. If a user is finished with a user interface, it may never wake up again and hence good GUI glue code must perform actions such as saving settings to flash memory *eagerly* to avoid not being able to satisfy its specification. Finally, the strict temporal alternation between input and output that is common in current reactive synthesis approaches is also not suitable, as a GUI controller can perform arbitrary many actions before giving control back to the operating system, which is a precondition for the next user interaction event (such as a button press) to occur. None of these differences to previous synthesis approaches prevent conflicts with the use of LTL as specification logic if we define a suitable execution semantics for GUI glue code, however, as we show in this section.

We formalize the interaction between the GUI code and its environment by defining a set of *environment events* $\Sigma^{\mathcal{I}}$ that model events not under the control of the GUI glue code to be synthesized and *controller actions* $\Sigma^{\mathcal{O}}$ that the said glue code can trigger. We assume that the controller can only react to events from $\Sigma^{\mathcal{I}}$ with arbitrarily long sequences of actions. Hence, a controller to be synthesized has the form $f : (\Sigma^{\mathcal{I}})^* \to (\Sigma^{\mathcal{O}})^*$, where providing the *history* of past input events to f yields the reaction to the *last* such event. To simplify specifying the desired properties of GUI glue code, we define a designated initialization event $\mathit{init} \in \Sigma^{\mathcal{I}}$ that the controller can always assume to get first when the application starts. Likewise, we define a designated "done" action $\mathit{done} \in \Sigma^{\mathcal{O}}$ that signals that a controller yields control back to the operating system and that always has to be exactly the last action in every sequence returned by f. Thus, the value of $f(w)$ for some $w \in (\Sigma^{\mathcal{I}})^*$ needs to be defined for words w that start with *init*, and $f(w)$ ends with *done* in this case.

Let $w^I = w_0^I w_1^I \ldots$ be a sequence of input events. We say that w^I induces a system trace $w = w_0^I f(w_0^I) w_1^I f(w_0^I w_1^I) w_2^I f(w_0^I w_1^I w_2^I) \ldots$. We define that w satisfies some LTL specification ψ if the word w' that results from translating each letter x in w to the letter $\{x\} \in 2^{\Sigma^{\mathcal{I}} \cup \Sigma^{\mathcal{O}}}$ satisfies ψ. If w^I is finite, the word w eventually ends, and we append an infinite number of repetitions of $\emptyset$ (the empty set) to ψ before interpreting the LTL formula.

Example 1. To motivate these definitions, let us look at a fragment of the expense split application specification. Specifications to be fulfilled by GUI glue code often contain assumptions made about the environment and the guarantees that the controller must fulfill. Let us consider that the guarantees contain the following specification part:

$\psi_g = \mathsf{G}(\mathit{newTeamButton.click} \rightarrow (\neg \mathit{done}\, \mathcal{U}\, \mathit{regTeam.start})$

$\wedge\,(\neg \mathit{regTeam.terminate}\,\mathcal{U}\,(\mathit{regTeam.terminate} \wedge (\neg \mathit{done}\,\mathcal{U}\,\mathit{updateTeamList}))))$

This guarantee states that whenever the GUI button for starting a new team is clicked, then a thread for registering a new team is started before the controller hands back control to the operating system. Registering a new team requires communication with a server, and a GUI should not be blocked until an answer from the server is received. This implies the need to offload the communication task to a separate thread. Furthermore, the specification states that when the thread eventually terminates afterwards, some instantaneous action *updateTeamList* is to be executed before control is given back to the operating system. Both this action and starting the thread lead to user-written back-end code being executed. The *updateTeamList* action is fast enough so that executing it in the context of the GUI does not lead to it blocking, and GUI frameworks such as the one used for Android applications even require such updates to be performed from the GUI (main) thread.

On its own, the specification is unrealizable, because the *regTeam* thread may never terminate. This can be fixed by adding the assumption to the specification that whenever the thread for registering a new team is started, it eventually terminates:

$$\psi_a = \mathsf{G}(\mathit{threadA.start} \rightarrow \mathsf{F}\mathit{threadA.terminates})$$

The specification used for synthesizing the glue code is then $\psi_a \rightarrow \psi_g$.

Figure 1 shows two trace parts of traces satisfying the specification. In the left example, the system correctly starts the team registration thread when the corresponding button is clicked. Eventually, the thread terminates (with no other GUI event happening in this example), and the controller chooses action *updateTeamList* as response, as specified.

The right trace shows an example for a finite trace. After starting the thread and giving back control to the OS by choosing action *done*, the GUI glue code never gets control back. This means that the trace is filled with $\emptyset$ to interpret the specification $\psi_a \rightarrow \psi_g$. Since ψ_a is violated on this trace, this means that the trace also fulfills the specification $\psi_a \rightarrow \psi_g$. Note that this is important as otherwise no controller would exist for this specification as the termination of a manually written thread is outside of the control of the synthesized GUI controller.

5 GR(1) Games for Event-based Specifications

Given an environment event set $\Sigma^{\mathcal{I}}$, a controller action set $\Sigma^{\mathcal{O}}$, an environment assumption formula ψ^A in LTL, and a system guarantee LTL formula ψ^G, the GUI controller synthesis problem is to check if there exists a controller function $f : (\Sigma^{\mathcal{I}})^* \rightarrow (\Sigma^{\mathcal{O}})^*$ such that all traces induced by the controller (using the semantics from the previous section) satisfy the specification $\psi^A \rightarrow \psi^G$.

Controller synthesis is commonly conceptually reduced to solving a game between an environment player and a system player. The environment player

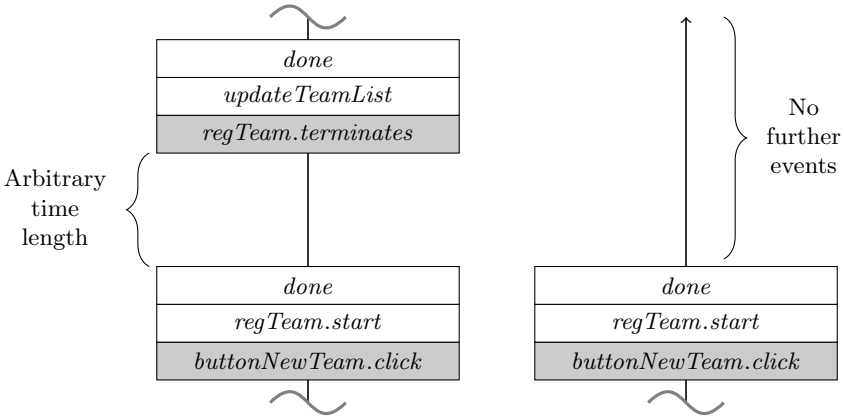

Fig. 1. Two trace parts of a controller satisfying the specification from Example 1. Time progresses from the bottom to the top.

has the role of choosing the uncontrollable input, and the system player wins the game if and only if it can always react in a way such that the *play* satisfies the specification for which the game was built. A play is built from following the edges of the game according to the events and actions that the two players choose. A play is in some position at every point in time, and the positions encode the *obligations* of the system player for the remainder of the play. Readers who want to learn more about game-based synthesis are referred to [5] and for conciseness, we assume familiarity with its basic ideas in the following.

5.1 Game definitions

For the approach in this paper, we build games from assumptions and guarantees that are both representable as universal very-weak automata (UVWs). We identified a fragment of LTL that can be efficiently translated into this form in earlier work [1], so that we can assume that UVWs for the assumptions and guarantees in a specification are given. The following game construction captures the main ideas of Generalized Reactivity(1) synthesis [7], but is adapted to the case of having UVW specifications:

Definition 1. *Let $\mathcal{A}^A = (Q^A, \Sigma, Q_0^A, \delta^A, F^A)$ be a UVW representing ψ^A and $\mathcal{A}^G = (Q^G, \Sigma, Q_0^G, \delta^G, F^G)$ be a UVW representing ψ^G. We define the* synthesis game *induced by $\mathcal{A}^A$ and $\mathcal{A}^G$ as a tuple $\mathcal{G} = (V, \Sigma^I, \Sigma^O, E, v_0)$ with the finite set of positions $V = 2^{Q^A} \times 2^{Q^G}$, the input and output players' action sets Σ^I and Σ^O, the initial position $v_0 = (Q_0^A, Q_0^G) \in V$, and the set of edges $E \subseteq Q \times \Sigma^I \times \Sigma^O \times 2^{F^A} \times 2^{F^G} \times Q'$, which consists of all elements $((v^A, v^G), x^I, x_0^O \dots x_n^O, L^A, L^G, (v'^A, v'^G))$ for which we have:*

$x_n^O = done$

$$
\begin{aligned}
\wedge\ & \exists Q'_0 \dots Q'_{n+2} \subseteq Q^A. \\
& \quad Q'_0 = v^A, Q'_{n+2} = v'^A, Q'_1 = \{q' \in Q^A \mid \exists q \in Q'_0, (q, x^I, q') \in \delta^A\}, \\
& \quad \forall i \in \{0, \dots, n\}.\ Q'_{i+2} = \{q' \in Q^A \mid \exists q \in Q'_{i+1}.(q, x^O_i, q') \in \delta^A\}, \\
& \quad L^A = \{q \in F^A \mid \exists i \in \{0, \dots, n+2\}.q \notin Q'_i \vee (q, x^I, q) \notin \delta^A\} \\
& \qquad \cup \{q \in F^A \mid \exists i \in \{0, \dots, n\}.(q, x^O_i, q) \notin \delta^A\} \\
\wedge\ & \exists Q'_0 \dots Q'_{n+2} \subseteq Q^G. \\
& \quad Q'_0 = v^G, Q'_{n+2} = v'^G, Q'_1 = \{q' \in Q^G \mid \exists q \in Q'_0, (q, x^I, q') \in \delta^G\}, \\
& \quad \forall i \in \{0, \dots, n\}.\ Q'_{i+2} = \{q' \in Q^G \mid \exists q \in Q'_{i+1}.(q, x^O_i, q') \in \delta^G\} \\
& \quad L^G = \{q \in F^G \mid \exists i \in \{0, \dots, n+2\}.q \notin Q'_i \vee (q, x^I, q) \notin \delta^G\} \\
& \qquad \cup \{q \in F^G \mid \exists i \in \{0, \dots, n\}.(q, x^O_i, q) \notin \delta^G\}
\end{aligned}
$$

The edge set definition requires some explanation. An edge $((v^A, v^G), x^I, x^O_0 \dots x^O_n, L^A, L^G, (v'^A, v'^G))$ denotes that if from position (v^A, v^G), the environment player chooses input action x^I and the system player chooses action sequence $x^O_0 \dots x^O_n$, then the next position in a play is (v'^A, v'^G). The additional components L^A and L^G denote the rejecting UVW automaton states that are *left* along such an edge.

The definition of a synthesis game is carefully crafted for the case that the specification from which the game is built comes in the form of assumption and guarantee UVWs, and hence deviates from the definitions found in other works. We chose to keep the strict alternation between the environment and system players' choices, which required that the system player chooses sequences of actions from Σ^O rather than single actions. Keeping the strict alternation greatly simplifies the presentation of the optimizations for game solving defined later in this paper. For the same reason, we also encoded the usual winning condition in GR(1) synthesis into the edges rather than separately (which is explained below). Finally, we do not have explicitly *owned* vertices for the two players. This is only a minor difference in case of strict alternation between the players and ensures that the controller definition from the previous section fits exactly the strategy definition for games given below.

More formally, we say that a sequence $\pi = \pi_0 \pi_1 \dots \in V^\omega$ is a *play* in the game if there exist corresponding decision sequences $\rho^{env} = \rho^{env}_0 \rho^{env}_1 \dots \in (\Sigma^I)^\omega$ and $\rho^{sys} = \rho^{sys}_0 \rho^{sys}_1 \dots \in ((\Sigma^O)^*)^\omega$ for the two players such that $\pi_0 = v_0$ and for every $i \in \mathbb{N}$, there is some suitable edge $(\pi_i, \rho^{env}_i, \rho^{sys}_i, L^A_i, L^G_i, \pi_{i+1})$ in the edge set of the game. We say that the play is *winning* if either:

- there exists some state $q \in F^A$ that appears in only finitely many sets L^A_i (for $i \in \mathbb{N}$), or
- for all states $q \in F^G$, there exist infinitely many sets L^G_i (for $i \in \mathbb{N}$) with $q \in L^G_i$.

Note that the edges are defined in a way such that the positions (v^A, v^G) track in which states runs of the automata $\mathcal{A}^A$ and $\mathcal{A}^G$ can be after reading a prefix

of the *interleaved* decision sequence $\tilde{\rho} = \rho_0^{env}\rho_0^{sys}\rho_1^{env}\rho_1^{sys}\ldots$. Furthermore, the sets L_i^A and L_i^G keep track of which rejecting states *every* run needs to leave (if it is in that state) when reading a part of $\tilde{\rho}$. If and only if some rejecting state occurs only finitely often in $\{L_i^A\}_{i\in\mathbb{N}}$, this means that some run of $\mathcal{A}^A$ gets stuck in a rejecting state when reading $\tilde{\rho}$. Likewise, if and only if some rejecting state occurs only finitely often in $\{L_i^G\}_{i\in\mathbb{N}}$, this means that some run of $\mathcal{A}^G$ gets stuck in a rejecting state when reading $\tilde{\rho}$. Hence, the winning condition of the game implements the requirement that if the interleaved decision sequence satisfies ψ^A, then it also needs to satisfy ψ^G for the system player to win the game. This observation enables us to frame the problem of finding a controller function f for a user interface as defined in the previous section as the problem of finding a *winning strategy* f for the system player in $\mathcal{G}$.

If there is a winning strategy in such a game, then there is also a finite-state one. This follows from the fact that the winning condition type is the same as in GR(1) game structures [7], so that the result that a strategy that is *positional per goal* suffices for GR(1) games carries over to our game definition. In our case, this means that the next choice of the strategy only depends on (1) the current position in the game, (2) the last input chosen by the environment player, and (3) the current *goal* of the system, which in our case is the state in F^G that is to be left next. By letting the controller cycle though all such goals, concatenating the sub-strategies for each goal leads to a correct finite-state controller implementation.

How to solve such games is described in [7]. We use a variation of the algorithm presented in [10] as it permits the definition of goal transitions rather than goal states. This is important to be able to define leaving a rejecting state of the guarantee automaton as a goal.

5.2 Pruning the Game

The game specified in Def. 1 has an infinite number of edges since the system player can choose sequences of actions rather than individual actions. Not all such decision sequences make sense, however. We want to prune the game before actually *solving* it, i.e., determining whether the system player has a winning strategy and computing such a strategy.

We prune the game based on the following observation:

Lemma 1. *Let f be a winning strategy for some game $\mathcal{G}$ built according to Def. 1 and let $\rho_0^{env}\ldots\rho_{m-1}^{env}$ be some finite sequence of actions chosen by the environment.*

(1) By Def. 1, there is exactly one edge $((v^A, v^G), x^I, x_0^O\ldots x_n^O, L^A, L^G, (v'^A, v'^G))$ with $x^I = \rho_{m-1}^{env}$ and $x_n^O = f(\rho_0^{env}\ldots\rho_{m-1}^{env})$ that can be taken at the nth step of the play.

(2) If there is another edge $((v^A, v^G), x^I, \hat{x}_0^O\ldots\hat{x}_m^O, \hat{L}^A, \hat{L}^G, (\hat{v}'^A, \hat{v}'^G))$ such that $\hat{L}^A \subseteq L^A$, $\hat{L}^G \supseteq L^G$, $\hat{v}'^A \supseteq v'^A$, and $\hat{v}'^G \subseteq v'^G$, then the strategy that results from modifying f to return $\hat{x}_0^O\ldots\hat{x}_m^O$ instead of $x_0^O\ldots x_m^O$ for $\rho_0^{env}\ldots\rho_{m-1}^{env}$ is still winning.

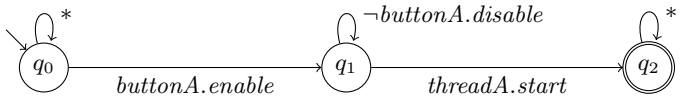

Fig. 2. Example guarantee UVW part for the example from Section 5.2.

Proof. The first claim follows directly from the definition of the game. The second claim follows from the fact that whether a strategy is winning depends on the labels of the positions visited along a play - if a state set v^G is replaced by a subset of v^G, then a suffix strategy is still winning as the states along the v^G components of the positions along of the play will then also be subsets, and hence the system player is less restricted in the possible moves (without being unable to satisfy the winning condition). The same holds for v^A, but in the reverse direction, as a superset of states in a v^A component means that the environment is more restricted without violating its assumptions. Similar arguments can be made about the L^A and L^G components. □

The incorporation of the L^A and L^G components into the argument ensures that the claim also holds if infinitely many strategy choices are replaced. This enables us to prune the game during its construction: we only need to enumerate edges that are not *dominated* by other edges. We say that an edge is dominated by another edge if the criterion from the preceding lemma can be applied to make a winning strategy never take that edge. Since there are only finitely many different sets v^A, v^G, L^A, and L^G, this makes the game graph finite. This idea can be made mathematically precise by defining a partial order over the positions and to then only explore the *anti-chain* of best-case states (for the system). Anti-chains over game positions have also been used in earlier works on reactive synthesis from full linear temporal logic [11].

We observed that anti-chains based game pruning greatly reduces the sizes of the games built from GUI specifications. As a trivial example, consider the case that ψ^G has a conjunct that requires the system to not start a certain thread while a certain button is enabled. Figure 2 shows a part of the corresponding UVW. If there is no reason to enable a button (meaning that enabling it does not change any other state in F^G or F^A to be left or a state in v^A or v^G to be reached) from some position (v^A, v^G), then the pruned game graph will simply not contain an edge from (v^A, v^G) along which the button is enabled, as this would cause state q_1 to become part of the v^G component, which is avoidable and hence is a dominated move.

5.3 Short decision sequences

Specifications for graphical user interfaces reason about events triggered by the user and the change of the interface's state. There are many combinations of actions that do not usually make sense, such as the controller disabling and enabling a button in the same step. When building a game according to Def. 1

and the anti-chains based pruning approach described above, we typically do not want the game edges to include such sequences.

We can avoid them by performing a *breadth-first search* of the feasible controller response sequences while enumerating all edges from a position in a game. If there are multiple ways to get to the same combination of successor position and *leaving state label* (L^A, L^G), the resulting edge will have one of the possible shortest such sequences.

For instance, this approach prevents the action sequence *buttonA.disable*; *buttonA.enable* from being a edge label from a position $(v^A, \{q0, q1\})$ in a game built using the UVW part from Fig. 2 as the only guarantee automaton. Since the successor guarantee automaton state set (for some arbitrary input event) is the same set for both the sequences *buttonA.disable*; *buttonA.enable* and *buttonA.enable* with the same set of rejecting states left along the game edge, only the shorter sequence has a chance to be found in the resulting game (assuming that none of the actions affected the assumption automaton).

6 Kind strategies

Good GUI controllers are *responsive* by enabling GUI elements whenever they can be enabled. While the UI designer could be required to specify exactly in which situations a GUI element should be enabled, this defeats the purpose of synthesis to enable designers to quickly prototype systems. It thus makes sense to integrate this requirement into the synthesis process itself.

The anti-chains based pruning approach from the previous section makes unresponsive controllers particularly likely to be found. For instance, if there is a button that can be disabled or enabled, and there is an assumption that states that the button cannot be clicked if it is not enabled, then there is an incentive for the system to disable the button, as in this case, the resulting successor position in the game is labelled by the additional assumption automaton state that checks this condition. This happens even if the button does not need to be disabled.

We solve this problem by defining a notion of a *kind* strategy that makes use of the fact that assumptions are encoded in UVW form and hence the situation-dependent obligations of the environment can be compared. A kind strategy make *kind* choices, which we define as follows:

Definition 2. *Let (v^A, v^G) be a position in a game $\mathcal{G}$, $x^I \in \Sigma^{\mathcal{I}}$ be an environment action, and $x^{O,1}$ and $x^{O,2}$ be two output sequences from $(\Sigma^{\mathcal{O}})^*$. We say that $x^{O,1}$ is* kinder *than $x^{O,2}$ if for the corresponding successor positions $(v^{A,1}, v^{G,1})$ and $(v^{A,2}, v^{G,2})$ reached by an edge for $(x^I, x^{O,1})$ and $(x^I, x^{O,2})$, respectively, we have that $v^{A,1} \subseteq v^{A,2}$.*

Note that the kindest strategy choices are typically *not* part of a game pruned according to the definition from the previous section, as successor states with smaller assumption automaton sets v^A are filtered out by the algorithm. Yet, the definition captures that the controller should perform actions that restrict

the environment as little as possible, as witnessed by the assumption automaton being in fewer states.

To marry these two concepts, we developed an iterative approach that combines anti-chain based game pruning with the search for kind moves in a synthesized strategy. Whenever a winning strategy is found, we iterate over the reachable positions in the game and for every input event replace the edges for the input event by edges that are strictly kinder than the one selected in the previous strategy (or by multiple ones if different ones are selected for different next goals of the system). Among the strictly kinder ones, we still perform pruning as described in Section 5.2 to keep the game small. Whenever the game becomes losing for the system after this change, our algorithm undoes the change. Otherwise, the changed game is kept. When the process completes for all positions reachable by the last strategy computed, this last strategy is the final kindest strategy found.

Note that this process can increase the number of positions in the game. It also requires many game solving iterations. We picked it because of its simplicity – if there is a strategy that is as kind as possible in every step and still satisfies the specification, the approach finds such a strategy. If kindness in every step conflicts with the system making progress towards leaving rejecting states, the system player does not win any more when trying to remove the last unkind edge useful for satisfying the specification. In this case, the edge removal is undone. The approach naturally avoids the problem that optimal strategies could become infinite-state, as it is the case when integrating quantitative objectives such as mean-payoff into a game in which the players have obligations to fulfil infinitely often [4, p. 150].

7 Case Study

We modelled the scenario from Section 2 using specification parts in the fragment of LTL that is easy to translate to UVWs [1]. All in all, our specification consists of 8 assumption LTL formulas and 118 guarantee LTL formulas.

When writing the specification, we made active use of the fact that we assume that the computed implementations use the shortest decision sequences as defined in Section 5.3 and that the strategy is *kind* as defined in Section 6. For instance, the guarantee parts

$$\mathsf{G}(\mathit{MenuItemOfflineActions} \rightarrow \neg \mathit{done}\, \mathcal{U}\, \mathit{PanelExpensesToApprove.hide})$$
$$\mathsf{G}(\mathit{MenuItemOfflineActions} \rightarrow \neg \mathit{done}\, \mathcal{U}\, \mathit{PanelOfflineActions.show})$$

enforce that when the offline actions menu item is selected from the main menu of the application, the view (panel) for the expenses to approve are hidden, and the offline actions view should be shown. There is no specification part that prevents the controller from hiding the latter view again immediately afterwards. But it is also not necessary, as that would lead to unnecessarily long decision sequences, and hence cannot be part of the strategy. Similarly, the specification does not

have guarantees that require the system to enable buttons in certain situations - whenever there is an assumption that a disabled button cannot be clicked, a kind strategy enables it whenever possible.

To validate our approach, we implemented the ideas presented in this paper in an explicit-state game solving tool written in `C++` and developed a prototype tool that analyzes a GUI layout of an Android application to enumerate the possible events, to build a game from the specification together with the event list, and to translate a computed strategy to synthesized GUI glue code in the programming language Java that can be compiled into the Android application. Both tools can be found at `https://github.com/tuc-es/guisynth` together with the expense splitting application.

To test the effectiveness of the approach presented in this paper, we also implemented translator scripts that encode the synthesis problem into generalized reactivity(1) specifications and into the standard LTL format used for the SyntComp competition. In both cases, we binary-encoded the events and actions to reduce the number of atomic propositions. The resulting specifications are quite complex as for comparability, the execution semantics from Section 4 also needed to be encoded. For the translation to GR(1), doing so would lead to specifications outside of the supported fragment; to mitigate this issue, we restricted all controller decision sequences to be of length at most 16, which is the longest sequences that we found our game solver to produce for the case study.

We use the resulting specifications for the GR(1) and full LTL synthesis tools `Slugs` and [10] `Strix` [15], where the latter won the full LTL synthesis competition SyntComp in 2018. For fairness, we must note that neither `Strix` nor `Slugs` are especially designed for the shape of GUI specifications, while our approach was especially crafted for the execution semantics described in Section 4.

Table 1 shows computation times and strategy sizes for kind and unkind strategies for several versions of our specification which represent its evolution during the writing process of the case study. Computation times for `Slugs` and `Strix` on the translated specifications are given as well, where it needs to be noted that these tools do not compute kind strategies.

It can be observed that the computed strategies are quite small. Since in our strategy definition, the system can output multiple actions at the same step, we also give the sizes of *flattened* finite-state machines from the strategies in which this is not the case. It can be observed that such a representation increases the number of states substantially.

8 Conclusion

In this paper, we presented a framework for the synthesis of graphical user interface glue code. Synthesizing such code allows rapid iteration cycles, and we did a first step towards even more scalable synthesis algorithms that are useful for establishing GUI glue code synthesis in industry. Our solution was carefully

Specification		Translation to UVWs		Game solving (not kind)			Game solving (kind)			Slugs	Strix
Rev.	# Properties	Time	# states	Time	Size strat.	Size st. flat	Time	Size strat.	Size st. flat	Time	Time
1	8	0.04	10	0.00	2	16	0.00	2	16	0.16	3.30
2	21	0.43	20	0.00	4	88	0.00	6	108	675.52	8.64
3	57	2.05	32	0.25	4	246	2.28	14	565	t/o	t/o
4	105	5.77	41	18.44	6	537	45.45	15	1044	t/o	t/o
5	124	10.41	44	182.87	6	630	389.14	15	1245	t/o	t/o
6	115	7.21	44	102.84	6	618	215.32	15	1215	t/o	t/o
7	135	12.0	47	589.04	6	711	1020.28	15	1413	t/o	t/o
8	134	10.76	46	186.28	6	675	283.00	15	1413	t/o	t/o
9	131	12.37	45	106.03	6	660	198.62	15	1392	t/o	t/o
10	127	11.35	45	60.39	6	660	267.90	48	3676	t/o	t/o
11	127	10.6	47	60.31	8	876	268.68	64	5129	t/o	t/o
12	126	13.49	50	483.67	48	3756	916.39	138	10074	t/o	t/o

Table 1. Results for the expense splitting example application. Time-outs (after two hours) are listed as "t/o", all other times are given in seconds (on an Intel i5 Processor with 1,6 GHz clock rate and 6 GB RAM available).

designed to make use of the particular properties of the application domain – thanks to the special execution semantics of GUI glue code, the system can perform multiple actions in a row, which keeps the game graph small when building the game only with best-case responses by the system. This enables explicit-state game solving and helps with the definition of *kind* strategies that represent responsive GUI controllers.

This work is only the first step, though. Our case study shows that the approach is already applicable, but will reach its scalability limit for more complex specifications. There are many opportunities for improvement, though. We did not perform incremental building of the game yet, as it was not necessary - the solving time is dominated by the time needed to find the edges in the game graph. Enumerating the anti-chain of best case responses requires to consider many action sequences by the system player – we believe that there is potential to optimize the search for best-case responses substantially.

In our framework, we separated control and data considerations completely to obtain a decidable synthesis problem. We also assumed that there can be at most one copy of a type of a thread at the same time. The reason is that employing finite-state games does not enable encoding the number of threads of a type running. Extending synthesis by this capability can easily lead to undecidability, and in many applications, it is easy to write threads to collect work and perform it sequentially, which removes the need to have more than one thread of a type running at the same time.

Performing more case studies to collect specifications to drive the scalability of synthesis from UI specifications forward is subject of future work. Furthermore, UI and user experience (UX) designers may benefit from more application-oriented specification languages for UI code rather than LTL. We will explore how such specification languages can look like while still permitting an efficient translation to UVWs.

References

1. Adabala, K., Ehlers, R.: A fragment of linear temporal logic for universal very weak automata. In: ATVA. LNCS, vol. 11138, pp. 335–351. Springer (2018)
2. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: REX. LNCS, vol. 600, pp. 74–106. Springer (1991)
3. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
4. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 140–156. Springer (2009)
5. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: et al., E.M.C. (ed.) Handbook of Model Checking, pp. 921–962. Springer (2018)
6. Bloem, R., Ehlers, R., Könighofer, R.: Cooperative reactive synthesis. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA. LNCS, vol. 9364, pp. 394–410. Springer (2015)
7. 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)
8. D'Ippolito, N., Braberman, V.A., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. ACM Trans. Softw. Eng. Methodol. **22**(1), 9:1–9:36 (2013)
9. Doyen, L., Geeraerts, G., Raskin, J., Reichert, J.: Realizability of real-time logics. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS. LNCS, vol. 5813, pp. 133–148. Springer (2009)
10. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV. LNCS, vol. 9780, pp. 333–339. Springer (2016)
11. Filiot, E., Jin, N., Raskin, J.: Exploiting structure in LTL synthesis. STTT **15**(5-6), 541–561 (2013)
12. Hussmann, H., Meixner, G., Zuehlke, D. (eds.): Model-Driven Development of Advanced User Interfaces, Studies in Computational Intelligence, vol. 340. Springer (2011)
13. Kress-Gazit, H., Lahijanian, M., Raman, V.: Synthesis for robots: Guarantees and feedback for robot behavior. Annual Review of Control, Robotics, and Autonomous Systems **1**, 211–236 (2018)
14. Masci, P., Zhang, Y., Jones, P.L., Curzon, P., Thimbleby, H.W.: Formal verification of medical device user interfaces using PVS. In: Gnesi, S., Rensink, A. (eds.) FASE. LNCS, vol. 8411, pp. 200–214. Springer (2014)
15. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV. LNCS, vol. 10981, pp. 578–586. Springer (2018)
16. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D. (eds.) ICALP. LNCS, vol. 372, pp. 652–671. Springer (1989)
17. Ryzhyk, L., Walker, A., Keys, J., Legg, A., Raghunath, A., Stumm, M., Vij, M.: User-guided device driver synthesis. In: OSDI. pp. 661–676 (2014)