Synthesis with Identifiers*

Rüdiger Ehlers[1,2,3], Sanjit A. Seshia[1], and Hadas Kress-Gazit[2]

[1] University of California at Berkeley, Berkeley, CA, United States
[2] Cornell University, Ithaca, NY, United States
[3] University of Kassel, Germany

Abstract. We consider the synthesis of reactive systems from specifications with identifiers. Identifiers are useful to parametrize the input and output of a reactive system, for example, to state which client requests a grant from an arbiter, or the type of object that a robot is expected to fetch.
Traditional reactive synthesis algorithms only handle a constant bounded range of such identifiers. However, in practice, we might not want to restrict the number of clients of an arbiter or the set of object types handled by a robot a priori. We first present a concise automata-based formalism for specifications with identifiers. The synthesis problem for such specifications is undecidable. We therefore give an algorithm that is always sound, and complete for unrealizable safety specifications. Our algorithm is based on computing a pattern-based abstraction of a synthesis game that captures the realizability problem for the specification. The abstraction does not restrict the possible solutions to finite-state ones and captures the obligations for the system in the synthesis game. We present an experimental evaluation based on a prototype implementation that shows the practical applicability of our algorithm.

1 Introduction

Automatically synthesizing reactive systems from their specifications is an ambitious, yet worthwhile challenge. The applicability of synthesis technology ranges from rapid prototyping to specification debugging, which improves system designer productivity and helps to find incorrect assumptions or forgotten requirements at an early stage in a system development process.

Traditionally, the input and output signals of the systems that are computed in reactive synthesis are purely boolean. If we are not interested in synthesizing hardware, but rather software, this view is often not justified. For example, in a mutual exclusion protocol, we might be getting requests for accesses to a shared resource from a group of clients whose size is unknown a-priori. A robot that satisfies some mission specification on the other hand might need to deliver a large variety of objects. In both cases, we are dealing with *identifier values* that form part of the input or output of a reactive system. For the mutual exclusion protocol, the identifiers represent client numbers, whereas for the robot example, they encode the types of the objects that the robot has to deliver.

* This work was partially supported by NSF ExCAPE CCF-1139025/1139138. The first author was also supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no. 259267.

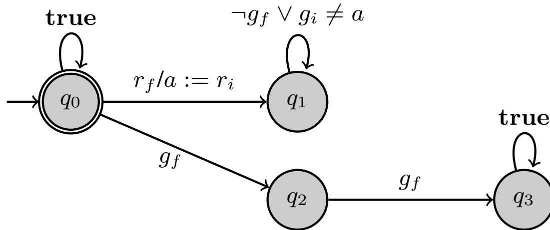


Fig. 1. An example specification (in form of a universal one-weak automaton) for a simple mutual exclusion (mutex) protocol with the input variables r_f (signaling that request is issued) and r_i (representing the identity of the request) and the output variables g_f (signaling that a grant is given) and g_i (for the identity of the grant). The variables r_i and g_i hold values from the domain of identifiers, whereas r_f and g_f are boolean. The automaton models that all requests must eventually be answered by a corresponding grant, and no two grants may be given in successive computation cycles. Accepting states are doubly-circled. As the automaton branches universally, it has to accept along all possible runs for a word to be accepted. Along a transition from q_0 to q_1, we assign a value to variable a that captures that a request with id $a = r_i$ has been issued. In state q_1, a run then waits until the request is answered by a grant, at which point it ends. States q_2 and q_3 ensure that no two grants may be given in successive transitions. All infinite runs for a word must be accepting for the word to be contained in the language of the automaton. Any implementation satisfying this specification is infinite-state, as requests may come in faster than they can be answered. Yet, the specification is realizable as there is no time bound on the answering time.

In this paper, we present an approach to synthesize reactive systems from *specifications with identifier variables*. We present a specification formalism that allows the concise representation of requirements for systems that have identifier input and output variables. By combining universal branching in word automata with identifier variables, we obtain a powerful, yet semantically simple way of describing specifications for such systems. Figure 1 shows an example of such a specification. Automata with universal branching are well-studied in the scope of reactive synthesis as they are a simple, yet expressive, specification model for reactive synthesis algorithms, and at the same time do not blow up under conjunction [10, 12]. This is highly desirable as practical specifications describe sets of properties that a system under design all need to fulfill, which are thus connected by conjunction.

The synthesis algorithm that we propose for such specifications exploits the conciseness of our formalism, as it can handle the identifiers in specifications in a symbolic way. The algorithm is sound, but not complete, as the synthesis problem from specifications with identifiers is undecidable. Additionally, our algorithm is always able to detect unrealizable safety specifications.

The core idea of our algorithmic solution is to build a *pattern-based abstraction* of an infinite realizability checking game in which the winning strategies represent correct implementations. The patterns describe constraints over *run points*, i.e., combinations of states in the universal specification automaton that we can be in (along with the corresponding variable valuations) at the same time. In this manner, we reduce dealing with the infinite *concrete realizability game* to solving a finite *abstract realizability game*. In the abstract game, the system player makes promises about how it can control the evo-

lution of a play in the concrete game. In order to compute the possible transitions in the abstract game, we solve a finite-step subgame between an environment player (that sets the next input to the system) and a system player (that sets the next output of the system) in which the system player tries to prove that it can keep its promises. We apply a solver for quantified boolean formulas (QBF) with free variables (ALLQBF) in order to find a compact representation of all moves of the two players in the abstract realizability game that allow the system player to win the finite-step subgame. Solving the abstract realizability game can then be performed using a classical generalized Büchi game solving algorithm [4]. By starting with a small set of patterns, and gradually letting this set grow whenever the abstract game is lost by the system player, we can balance the precision of the abstract game against the computational burden of building it. Often, a small set of patterns suffices, and we exploit this fact in our construction.

The implementations synthesized with our approach are not necessarily finite-state. For example, for the specification given in Figure 1, any implementation satisfying it needs to be infinite-state, and our synthesis algorithm finds one. This distinguishes our approach from classical reactive synthesis methods and classical abstraction-based solution methods for infinite games [7], which can only find finite-state implementations. An implementation that is the outcome of our synthesis approach uses queues as primary datatype to store information about obligations to be fulfilled. As an example, for the specification in Figure 1, the queue would be used to store all requests not having been served yet. While an infinite-state strategy can surely not be exactly implemented with actual hardware, our synthesized implementations use the available memory in a conservative manner, and are thus implementable in practice for input sequences that do not enforce excessive memory usage.

We start by describing our specification modeling framework in Section 2, followed by a theoretical analysis of the corresponding synthesis problem in Section 3. Then, we show how to synthesize from specifications with identifiers with a sound algorithm in Section 4. This algorithm is also guaranteed to detect unrealizable safety specifications. We give some experimental results on a prototype implementation of our approach in Section 5 and conclude with a summary in Section 6.

1.1 Related Work

The benefit of abstracting from concrete data values is well-known in the scope of verification. Wolper [15] defines the notion of *data-independence*, which intuitively means that the control flow of a program only depends on the equalities of the data items handled by the program. He shows how data-independence eases the verification of a large class of properties. His idea is integral to our synthesis approach as all implementations we compute are data-independent.

Previous work on synthesizing systems with infinite input and output domains only considered specification languages that did not permit connecting the values from the infinite domain over time, but rather only allowed local comparisons in every time step. Cheng and Lee [5] present a synthesis approach for cyber-physical systems in which linear-time temporal logic (LTL) as specification logic is extended to allow comparisons of continuous variables such as sensor values as literals. Tabuada [13] considers similar specifications and describes techniques to synthesize systems that not only meet their

specification, but do so in a way that is robust to pertubations of the input or output values.

The problem of synthesizing systems with infinite input and output domains is related to solving games with an infinite state space and synthesizing arbitrarily scalable systems. Dimitrova and Finkbeiner [6] discuss the solution of infinite incomplete-information games. They present an abstraction-refinement approach that can find finite winning strategies in such games if these exist. Their approach is not suitable for realizable specifications that have no satisfying finite-state implementation.

Walukiewicz considers the problem of solving parity games over a push-down structure [14]. Winning strategies in such games can be infinite-state, just like in our setting. As push-down games extend the expressible specification by non-regular properties rather than allowing an infinitely-sized input/output alphabet, they are not applicable in the settings dealt with in this paper.

Attie and Emerson describe methods to synthesize arbitrarily scalable systems [1]. Starting with a specification, they propose to synthesize a pair of processes that can then be instantiated as often as needed, and the composition of these processes still satisfies the original specification. As the number of allowed instantiations is not bounded, there is no bound of the state space of the product process. Their composed processes can deadlock in some situations, which is undesirable. Jacobs and Bloem [8] consider the same problem for ring architectures of processes. They show that task to be undecidable for specifications in linear-time temporal logic (LTL), but give a sound semi-algorithm. In contrast to our synthesis algorithm, all of these approaches to synthesize arbitrarily scalable systems cannot deal with specifications that always need an infinite number of states in their implementation regardless of the number of process instantiations, and can also not deal with input/output alphabets with an infinite domain.

2 Modeling Parametrized Specifications

Basics: In this work, we consider *reactive systems with data*, for which the data domains for all input and output signals are either boolean or identifiers. Our reactive system thus has an input signal set $\mathcal{I} = \mathcal{I}_B \uplus \mathcal{I}_I$ that consists of boolean input signals $\mathcal{I}_B$ and signals for reading identifiers $\mathcal{I}_I$, and an output signal set $\mathcal{O} = \mathcal{O}_B \uplus \mathcal{O}_I$ that can be decomposed in the same manner. We call $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ the *interface* of a reactive system. We denote by ID the (infinite) set of identifiers; however, we note that, for the scope of this paper, it is not relevant to fix a concrete domain ID, as we consider equality checks as the only operation on them. In our examples, we always use integers for simplicity. The system runs in discrete time steps, called *computation cycles*, for an indefinite number of steps, which we abstract from by considering *infinite runs of the system*. Such a run is formally given as a word $w = w_0 w_1 w_2 \ldots$, where for every $i \in \mathbb{N}$, we have $w_i \in \mathcal{IS} \times \mathcal{OS}$ for the *input assignment set* $\mathcal{IS} = (\mathcal{I}_I \to \mathsf{ID}) \times (\mathcal{I}_B \to \mathbb{B})$ and the *output assignment set* $\mathcal{OS} = (\mathcal{O}_I \to \mathsf{ID}) \times (\mathcal{O}_B \to \mathbb{B})$. For example, the following

word represents a run of a reactive system for the specification in Figure 1:

$$w = \begin{pmatrix} r_f \mapsto \mathbf{false} \\ r_i \mapsto \quad 0 \\ g_f \mapsto \mathbf{false} \\ g_i \mapsto \quad 0 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{true} \\ r_i \mapsto \quad 5 \\ g_f \mapsto \mathbf{true} \\ g_i \mapsto \quad 5 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{true} \\ r_i \mapsto \quad 3 \\ g_f \mapsto \mathbf{false} \\ g_i \mapsto \quad 0 \end{pmatrix} \begin{pmatrix} r_f \mapsto \mathbf{false} \\ r_i \mapsto \quad 0 \\ g_f \mapsto \mathbf{true} \\ g_i \mapsto \quad 3 \end{pmatrix} \ldots \quad (1)$$

Formally, we can specify the *behavior* of a reactive system by a function $f : \mathcal{IS}^+ \to \mathcal{OS}$ that maps input histories to an output to produce next. If for a word w, we have that for all $i \in \mathbb{N}$, $f(w_0|_{\mathcal{IS}}\, w_1|_{\mathcal{IS}} \ldots w_i|_{\mathcal{IS}}) = w_i|_{\mathcal{OS}}$, then we say that w is a *run* of f.

Specifications: Given some reactive system interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, a specification is a language $L \subseteq (\mathcal{IS} \times \mathcal{OS})^\omega$. Given some reactive system behavior function $f : \mathcal{IS}^+ \to \mathcal{OS}$, we say that f satisfies L if all runs of f are contained in L. The realizability problem for a language L is to check for the existence of such a behavior function f, and the synthesis problem is to obtain a representation of such a function f (if it exists).

Universal semi-one-weak automata for specifications: To represent specifications over words of infinite length (and an infinite number of identifiers to choose from) in a finitely-representable way, we use *universal semi-one-weak automata* with identifier variables. Formally, for some system interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, such an automaton is given as a tuple $\mathcal{A} = (Q, S, \delta, q_{init}, F)$, where Q is a finite set of states, $S : Q \to 2^{\mathsf{Var}}$ is a scoping function that describes which variables are defined in which states for some domain of identifier variables Var, $q_{init} \in Q$ is the initial state such that $S(q_{init}) = \emptyset$, F is a set of accepting states, and δ is a finite set of transitions.

Every transition is of the form (q, C, A, q'), where $q \in Q$ is a source state, q' is the target state, C is a set of conditions, and A is a set of assignments. A condition is either of the form $v_1 \neq v_2$, $v_1 = v_2$, $b = \mathbf{true}$, or $b = \mathbf{false}$ for some $b \in \mathcal{I}_B \uplus \mathcal{O}_B$ and $v_1, v_2 \in (S(q) \uplus \mathcal{I}_I \uplus \mathcal{O}_I)$. An assignment is of the form (v, u), where $v \in \mathsf{Var}$, $v \in S(q') \setminus S(q)$, and $u \in \mathcal{I}_I \uplus \mathcal{O}_I$; intuitively, u is copied into variable v. For a transition to be valid, we require that $S(q') = S(q) \uplus \{v \in \mathsf{Var} \mid \exists t \in \mathcal{I}_I \uplus \mathcal{O}_I : (v, t) \in A\}$, and every variable may only occur in A once. Along a sequence of transitions in the automaton, every variable may only be assigned once, and the aim of introducing a scoping function into the automaton definition is to make explicit which variables are defined in which states.

The automata that we are concerned with in this paper are *semi-one-weak*, i.e., are like *weak automata* for all accepting states, and are *one-weak* for all non-accepting states, which we also call *rejecting states* in the following. Formally, $\mathcal{A}$ is weak if we can partition Q into a finite number of subsets that are partially ordered by some comparator $\leq_Q$ such that every subset contains only accepting states or only non-accepting states, and for every transition $(q, C, A, q') \in \delta$, for K_q being the partition element that q is in, and $K_{q'}$ being the partition element that q' is in, we have $K_q \leq_Q K_{q'}$. For our semi-one-weak automata, we furthermore require that every rejecting state is one-weak, i.e., it is the only element in its partition. Informally, this means that the only looping paths in the automaton that contain a rejecting state consist solely of self-loops in the rejecting state.

Words $w = w_0 w_1 \ldots \in (\mathcal{IS} \times \mathcal{OS})^\omega$ induce runs in the automaton, where every point in the run is a combination of a state of $\mathcal{A}$ that the run is in, and a variable valuation for the variables in the scope of the state. Formally, every *run point* is thus an element of $\Pi = \{(q, f) \in Q \times (\mathsf{Var} \rightharpoonup \mathsf{ID}) \mid \mathrm{domain}(f) = S(q)\}$, and we say that some sequence $\pi = \pi_0 \pi_1 \ldots \pi_n \in \Pi^*$ is a finite run if $\pi_0 = (q_{init}, \emptyset)$ and for all $i \in \{0, \ldots, n-1\}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta_\Pi$, and that some sequence $\pi = \pi_0 \pi_1 \ldots \in \Pi^\omega$ is an infinite run if $\pi_0 = (q_{init}, \emptyset)$ and for all $i \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta_\Pi$. In both cases, the relation $\delta_\Pi \subseteq \Pi \times (\mathcal{IS} \times \mathcal{OS}) \times \Pi$ describes the possible transitions in a run on a semantic level. It is defined to consist of all tuples $((q, f), x, (q', f'))$ such that there exists some automaton transition (q, C, A, q') such that for all $c \in C$, we have $(f, x) \models c$, and $f' = f \cup \{(v \mapsto x(m)) \mid (v, m) \in A\}$. We say that a run of $\mathcal{A}$ is accepting if for some $q \in F$, there are infinitely many indices i such that $\pi_i = (q, f)$ for some variable assignment f or if the run is finite. We say that $\mathcal{A}$ accepts a word w if all runs for w are accepting.

To simplify the presentation, we also represent semi-one-weak automata with IDs graphically as shown in Figure 1. Accepting states (i.e., those in F) are drawn doubly-circled. Transitions are depicted as arrows, labeled by the conditions and actions. For example, the arrow from state q_0 to q_1 in the figure is formalized as $(q_0, \{r_f = \mathbf{true}\}, \{(a, r_i)\}, q_1)$, whereas the self-loop on state q_1 actually represents two transitions, namely $(q_1, \{g_f = \mathbf{false}\}, \emptyset, q_1)$ and $(q_1, \{g_i \neq a\}, \emptyset, q_1)$.

3 An Analysis of the Synthesis Problem

We start our analysis of the synthesis problem for specifications with identifiers on a theoretical level. After some basic definitions, we define synthesis games and establish *determinacy* of the games. Finally, we derive the undecidability of the synthesis problem for specification with identifiers.

3.1 Basic Definitions

Let $w = w_0 w_1 \ldots \in (\mathcal{IS} \times \mathcal{OS})^\omega$ be a word and $\mathcal{A} = (Q, S, \delta, q_{init}, F)$ be an automaton over $\mathcal{IS} \times \mathcal{OS}$. We can arrange all runs π that correspond to w and $\mathcal{A}$ in a *run tree*. Formally, such a run tree is given as a tuple $\langle T, \tau \rangle$ with a prefix-closed set T and a function $\tau : T \to \Pi$ that maps every tree node to a state and a variable valuation at this state. Figure 2 shows a graphical representation for a run tree for the automaton from Figure 1 and the example word in Equation 1. We obtain $\langle T, \tau \rangle$ from $\mathcal{A}$ by letting T be the smallest subset of Π^* that contains $(q_{init}, \emptyset)$ and such that for every $\pi_0 \pi_1 \ldots \pi_n \in T$, we have that $\pi_0 \pi_1 \ldots \pi_n \pi_{n+1} \in T$ for precisely those $\pi_{n+1} \in \Pi$ with $(\pi_n, w_n, \pi_{n+1}) \in \delta_\Pi$. For all $\pi_0 \pi_1 \ldots \pi_n \in T$, we set $\tau(\pi_0 \pi_1 \ldots \pi_n) = \pi_n$.

We say that a run tree is accepting if for every infinite sequence $\pi = \pi_0 \pi_1 \ldots \in \Pi^\omega$, if for all $i \in \mathbb{N}$, we have $\pi_0 \pi_1 \ldots \pi_i \in T$, then there exist infinitely many $i \in \mathbb{N}$ such that for $\tau(\pi_0 \pi_1 \ldots \pi_i) = (q, f)$, we have $q \in F$. By definition, for a word, there exists an accepting run tree if and only if the word is accepted.

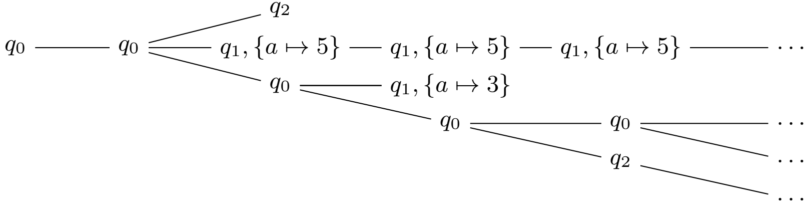


Fig. 2. An example run tree, growing from left to right.

3.2 Synthesis Games

A commonly used formalism to study the synthesis problem are *two-player games*. Formally, a game is defined as a tuple $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v^{init}, \mathcal{F})$. We have two players, called player 0 and player 1. Every player p has a set of vertices V^p, a set of actions Σ^p, and an edge function $E^p : V^p \times \Sigma^p \rightarrow V^{(1-p)}$. Without loss of generality, we assume that the initial position v^{init} is an element of V^0. For the scope of this paper, the winning condition $\mathcal{F}$ is defined as a set of subsets of the edges of player 1, i.e., $\mathcal{F} \subseteq 2^{V^1 \times \Sigma^1}$.

In a synthesis game, we declare one player to be the *system player*, whereas the other player is the *environment player*. The system player tries to *win* the game according to the winning condition $\mathcal{F}$, whereas the environment player tries to prevent this.

During the course of the play, the two players alternate in making their moves. They do so by choosing from their respective sets of actions. Afterwards, the position in the game is updated according to the player's edge function, and the play continues. Since the edge functions are required to be total, the play of the game never ends. During the course of the play, the moves of the two players can be collected into their *decision sequence* $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \rho_2^0 \ldots$, in which for every $i \in \mathbb{N}$ and $p \in \{0, 1\}$, we have $\rho_i^p \in \Sigma^p$. The corresponding *play* of the game represents the sequence of positions visited when the two players choose their actions as described in ρ. Formally, a play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \pi_0^2 \ldots$ corresponding to ρ is defined as $\pi_0^0 = v^{init}$ and for every $i \in \mathbb{N}$ and $p \in \{0, 1\}$, we have $\pi_{i+p}^{1-p} = E_p(\pi_i^p, \rho_i^p)$. We say that a play is winning for player 1 if for all sets of $X \in \mathcal{F}$, player 1 chooses edges in X infinitely often, i.e., there are infinitely many indices $i \in \mathbb{N}$ such that $(\pi_i^1, \rho_i^1) \in X$. Such a winning condition is typically called *transition-based generalized Büchi* for games in which V_0 and V_1 are finite.

Any of the two players in the game can play a *strategy*. Formally, a strategy for player $p \in \{0, 1\}$ is a function $f^p : (\Sigma^{1-p})^* \rightarrow \Sigma^p$. We say that a decision sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \rho_2^0 \ldots$ is in correspondence to some strategy f^p if for all $i \in \mathbb{N}$, we have $\rho_i^p = f^p(\rho_0^{1-p} \rho_1^{1-p} \ldots \rho_{i-1+p}^{1-p})$. If for some strategy f^p, all decision sequences that are in correspondence to f^p induce plays that are winning for player p, then we say that f^p is a winning strategy for player p. We also say that player p *wins the game* whenever it has a winning strategy.

The fact that strategies in games and implementations of systems with identifiers look very similar is no coincidence, as we want to use games to solve synthesis problems – we build games such that the winning strategies for the *system player* in the

games *are* in fact the implementations that we are searching for. Starting from a universal semi-one-weak automaton $\mathcal{A} = (Q, S, \delta, q_{init}, F)$, taking player 1 as the system player, and calling player 0 the *environment player*, we build a game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v^{init}, \mathcal{F})$ such that $\Sigma^0 = \mathcal{IS}$ and $\Sigma^1 = \mathcal{OS}$. We furthermore define:

$$
\begin{aligned}
V^0 &= 2^{\Pi} \\
V^1 &= V_0 \times \mathcal{IS} \\
E^0(v, x) &= (v, x) \text{ for all } v \in V^0, x \in \Sigma^0 \\
E^1((v, x), y) &= \{(q', f') \in \Pi \mid \exists (q, f) \in v, ((q, f), (x, y), (q', f')) \in \delta_{\Pi}\} \\
&\qquad \text{for all } v \in V^0, x \in \Sigma^0, y \in \Sigma^1 \\
v^{init} &= \{(q_0, \emptyset)\} \\
\mathcal{F} &= \bigcup_{(q,f) \in (Q \setminus F) \times (\mathsf{Var} \to \mathsf{ID})} \{\{(X, x, y) \subseteq 2^{\Pi} \times \mathcal{IS} \times \mathcal{OS} \mid \\
&\qquad (q, f) \notin X \vee ((q, f), (x, y), (q, f)) \notin \delta_{\Pi}\}\}
\end{aligned}
$$

In this game, every position in V^0 intuitively describes a set of run points in the run tree, and E^0 and E^1 ensure that whenever the two players construct some prefix decision sequence $\rho = \rho^0_0 \rho^1_0 \rho^0_1 \rho^1_1 \ldots \rho^0_k \rho^1_k$, then for v being the position reached in the game along a play for this sequence, v is precisely the set of run points that are at level k in the run tree for a word starting with $(\rho^0_0, \rho^1_0)(\rho^0_1, \rho^1_1) \ldots (\rho^0_0, \rho^1_0)(\rho^0_k, \rho^1_k)$. Thus, we can intuitively read off the complete run tree for a decision sequence from its induced play. The winning condition $\mathcal{F}$ then characterizes the set of run trees for which along no branch we eventually get stuck in a run point for a rejecting state. This ensures that precisely the decision sequences that have winning plays in the game are accepted by the specification automaton, and thus the game can be called the *synthesis game* for $\mathcal{A}$. Note that we used the semi-one-weakness of our specification automaton and the fact that variable values never change along a run of the automaton in the definition of the winning condition. Without these facts, the winning condition would need to trace the history of a run point in order for the winning plays in the game to represent the traces that satisfy the specification from which we built the game. The winning condition could not be simply concerned with the edges that are taken infinitely often along a play in the game then.

Lemma 1. *Let $\mathcal{A}$ be a specification automaton over some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and $\mathcal{G}$ be a game built from $\mathcal{A}$ and the interface according to the definitions above. If and only if $\mathcal{G}$ is winning for player* 1*, there exists an implementation with interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ all of whose runs are in the language described by $\mathcal{A}$. Furthermore, the winning strategies for player* 1 *in $\mathcal{G}$ are such implementations.*

Determinacy of synthesis games: An important question in game theory is whether a class of games is *determined*, i.e., whether any game in the class admits a winning strategy for one of the players. By the connection between semi-one-weak automata with identifiers and their corresponding games established by Lemma 1, determinacy of all games of the form described above implies that our synthesis problem is actually well-posed: for every specification, there is either an implementation, or we can (theoretically) prove that none exists.

Martin [11] showed that every two-player game for which the winning plays for one of the players form a *Borel set* is determined. This argument is not directly applicable to the type of games built here, as the set of Borel sets is only closed under countable unions/set intersections, but as the identifier domain is infinite, the set of positions in synthesis games can be non-countable. However, note that any identifier value used as input or output of the two players that did not yet occur in the prefix decision sequence in a game always has the same effect on whether a play is going to be winning or not. Thus, we can restrict both players to use fresh identifiers in a certain order (e.g., in increasing order when using integer identifiers) without changing any property of the game, except for the fact that in every position, the two players now only have a finite set of possible moves. This makes the set of positions in the game countable and it can then be shown that the winning plays for any of the players is a Borel set.

Undecidability of synthesis from semi-one-weak automata with identifiers: Despite the simplicity of our specification framework for systems with identifiers, its synthesis problem is unfortunately undecidable. Intuitively, the reason is that we can translate a Turing machine description to a specification that is unrealizable if and only if the Turing machine halts on the empty input tape – the environment in this context provides a sequence of identifiers that serve as addresses on the tape, and the system is required to output the sequence of Turing tape computations along with the Turing machine state. By requiring that an accepting Turing machine state must never be reached, we connect the realizability problem with Turing machine acceptance.

Theorem 1. *Realizability checking for specifications expressed as semi-one-weak universal automata with identifiers is undecidable.*

Proof. See appendix.

4 Synthesis Algorithm

As the realizability problem for specifications represented as semi-one-weak universal automata with identifier variables is undecidable, we can only rely on sound, but incomplete, methods to perform synthesis for such specifications. The main idea pursued in the following is to build a finite game that abstracts from details in the synthesis games defined in Section 3.2. The fact that the only data type we consider in this paper are identifiers comes to our rescue at this point, as the single operation that needs to be supported for them is checking for equality. To characterize a situation in the game, it thus suffices to state the equalities of the variable valuations in different run points by which a game position is labeled. We combine this idea with *sound overapproximation* of game situations to ensure the correctness of the computed implementations.

4.1 Patterns

Let $\mathcal{A}$ be a universal automaton with identifiers, and $\mathcal{G}$ be the game built from $\mathcal{A}$ according to the construction from Sect. 3.2. If for a position $v \in V_0$, changing the initial

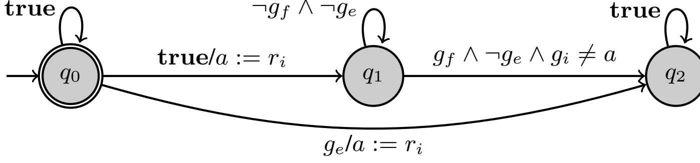


Fig. 3. Example specification for an interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ with $\mathcal{I}_B = \emptyset, \mathcal{I}_I = \{r_i\}, \mathcal{O}_B = \{g_e, g_f\}$, and $\mathcal{O}_I = \{g_i\}$

position to v leads to the game being losing for the system player, then v is called a *bad position*, as once the game reaches v in a play, the system player has no strategy to win. Note that the definition of the games considered here makes sure that if some position v is a bad position, then some other position v' that we can obtain by taking a bijective function $g : \mathsf{ID} \rightarrow \mathsf{ID}$, and replacing every identifier i in v by $g(i)$, is also a bad position, as the concrete values of the identifiers do not matter in our setting, and only their equivalences are of importance. This observation gives rise to the idea of abstracting positions into *pattern*s.

Consider the example specification in Figure 3. In the game that is built according to the construction from Sect. 3.2 from the specification, the position $\{(q_0, \emptyset), (q_1, \{a \mapsto 1\}), (q_1, \{a \mapsto 2\})\}$ is losing for the system player. This can be seen from the fact that from that position, either g_f or g_e have to be set to $\mathbf{true}$ by the system player in order to eventually leave the run points $(q_1, \{a \mapsto 1\})$ and $(q_1, \{a \mapsto 2\})$, with q_1 being rejecting. Since choosing $g_e = \mathbf{true}$ would lead to the transition from q_0 to q_2 being taken, and choosing $g_f = \mathbf{true}$ would lead to taking the transition to q_2 as g_i cannot be 1 and 2 at the same time, we cannot avoid transitioning to q_2, from where we reject a suffix run of the automaton. By the fact that we could replace the concrete identifiers by other values that keep the relationship between the items, and obtain an equally losing position, we call $P = \{(q_0, \emptyset), (q_1, \alpha_1), (q_1, \alpha_2)\}$ a *bad pattern*, as every position that represents an instantiation of this pattern (by substituting the variables α_1 and α_2 by concrete, distinct identifiers values) is losing for the system player.

Note that bad patterns describe sufficient conditions for losing a game. If P is a bad pattern in a game, then the pattern tells us that any position for which we find an instantiation of the bad pattern in its run point set is losing. This way, for example, also the position $\{(q_0, \emptyset), (q_1, a \mapsto 12), (q_1, a \mapsto 42), (q_1, a \mapsto 123)\}$ is losing, as a bad pattern matches a subset of its run points. This stems from the fact that positions are characterized by the run points for runs in a universal automaton, and the more combinations we have, the more properties does the suffix decision sequence have to fulfill in order for the overall decision sequence to be accepted by the automaton.

4.2 Abstract Games

To solve the synthesis problem for a universal semi-one-weak specification automaton $\mathcal{A} = (Q, S, \delta, q_{init}, F)$ with identifiers, we take the *concrete synthesis game* $\mathcal{G}$ built from the specification according to Sect. 3.2, and build an *abstract game* $\mathcal{G}_A$ from $\mathcal{G}$ that is finite, and thus can be solved by practical game solving algorithms. Every position in the abstract game is labeled by a set of *forbidden patterns*. The abstract position

then represents all concrete game positions for which we cannot instantiate any forbidden pattern in the run points by which the concrete game position is labeled. In every abstract position, we require the system player to have suitable next moves for *every* corresponding concrete position. Thus, if the system player can win the game, we know that the specification is realizable.

Patterns can be arbitrarily large, as they can have an arbitrary number of elements. To obtain a finite number of game positions with this idea, we only take patterns from a finite *base set of patterns* $\mathcal{M}$ into consideration. We can, for example, define $\mathcal{M}$ to be the set of all patterns with $\leq b$ elements for some $b \in \mathbb{N}$. With a restricted base set of patterns, our game is only approximate. To achieve the soundness of a synthesis approach based on this idea, we have to ensure that the approximation does not restrict the environment player in any way, and can only put the system player in a disadvantage, as we will do below.

Having only a finite number of positions however does not automatically make the game finite. In fact, the action sets in $\mathcal{G}$ are also infinite. As a remedy, in our abstract game, the two players make *abstract decisions* for their identifier and boolean signals. We use an abstraction that is both simple and powerful: the environment player chooses a subset of the specification automaton transitions as its move, while the system player declares the next forbidden patterns and the states for which it wants to make *progress*.

The idea here is to let the two players announce the *effect* of their choice of moves in the synthesis game rather than giving concrete identifier values, i.e., which automaton transitions are enabled by the move of the environment, and what the successor pattern set is. This idea reduces the two player's decisions to a *finite* domain.

Recall that for a transition $(q, C, A, q') \in \delta$ to fire, *all* constraints in C have to be fulfilled. We call a transition *semi-enabled* (by the environment player) for some choice of boolean and identifier input signal valuations if all constraints over the input signals are satisfied. For an environment player's move to be legal from a position v in the abstract game $\mathcal{G}_A$, there has to exist some position in $\mathcal{G}$ that satisfies all of the constraints of the patterns by which v is marked and some identifier input signal valuation such that the transitions chosen by the player are semi-enabled by the input signals.

After the environment player has made its move, it is the system player's task to (1) choose a set of successor patterns and (2) declare for which rejecting states it wants to perform progress on leaving them. We say that a state is left at a point in the run of the automaton if either the run is not in that state at the point considered, or the state's self-loop is not taken. Consider for example the excerpt from the synthesis game depicted in Figure 4 that we built from the specification in Figure 1. The system player, who owns the left-most position in the figure, can enforce to leave run point $(q_1, \{a \mapsto 3\})$ by choosing $g_f \wedge g_i = 3$ as the next move, and it can enforce to leave run point $(q_1, \{a \mapsto 5\})$ by choosing $g_f \wedge g_i = 5$. Thus, it can declare to be able to make progress on leaving (any run point for) state q_1 and to transition to a position in which the patterns $\{(q_1, \{a \mapsto \alpha_1\}), (q_1, \{a \mapsto \alpha_2\})\}$ and $\{(q_3, \emptyset)\}$ cannot be instantiated.

The system player has to play conservatively, i.e., choose its move while taking into account *every* concrete position that satisfies the constraints imposed by the patterns in v and any of the environment player's concrete input signal values for which the environment player's chosen transition set is valid. In all of these possible cases, there has

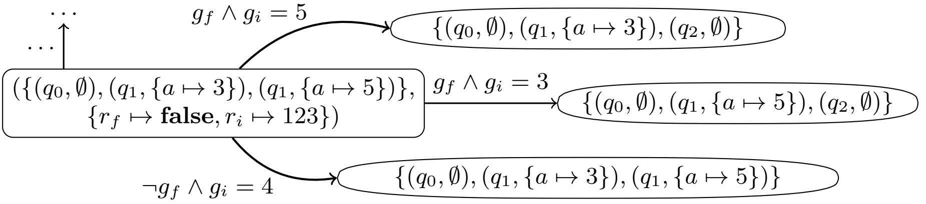


Fig. 4. An excerpt from a concrete synthesis game. Positions of player 0 are drawn as ellipses, while the position of player 1 is denoted as a rectangle

to be some concrete move of the system player that ensures that the resulting successor position in the concrete game satisfies the patterns declared by the player, and at the same time, progress can be performed by leaving any run point for the states declared.

Let us now formalize $\mathcal{G}_A$ using these ideas. The specification automaton $\mathcal{A}$ is given for some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and we have a finite set of base patterns $\mathcal{M}$. A pattern is a set of elements of Θ, where $\Theta = Q \times (\mathsf{Var} \rightharpoonup \{\alpha_i\}_{i \in \mathbb{N}})$ is the set of *pattern atoms*. Without loss of generality, we assume that for every pattern atom (q, f), the domain of f is $S(q)$, and for all $P \in \mathcal{M}$, the set $\{i \in \mathbb{N} \mid \exists (q,f) \in P, e \in S(q) : f(e) = \alpha_i\}$ is of the form $\{0, 1, \ldots, j\}$ for some $j \in \mathbb{N}$. Formally, we define $\mathcal{G}_A = (V_A^0, V_A^1, \Sigma_A^0, \Sigma_A^1, E_A^0, E_A^1, v^{init}, \mathcal{F})$ with the following properties (using the function Post as a placeholder to be explained below):

$$
\begin{aligned}
V_A^0 &= 2^{\mathcal{M}} \cup \{\bot, \top\} \\
V_A^1 &= 2^{\mathcal{M}} \times \Sigma_A^0 \\
\Sigma_A^0 &= 2^{\delta} \\
\Sigma_A^1 &= 2^{\mathcal{M}} \times 2^{Q \setminus F} \\
E_A^0(v, X) &= (v, X) \text{ for all } v \in 2^{\mathcal{M}}, X \in \Sigma_A^0 \\
E_A^1((v, X), (Y_P, Y_D)) &= \mathsf{Post}(v, X, Y_P, Y_D) \text{ for all } (v, X) \in V_A^1, (Y_P, Y_D) \in \Sigma_A^1 \\
\mathcal{F} &= \{V_A^1 \times 2^{\mathcal{M}} \times H_q \mid q \in (Q \setminus F), \\
& \qquad H_q = \{Q' \subseteq Q \setminus F \mid q \in Q'\}\} \\
v^{init} &= \mathcal{M} \setminus \{\emptyset, \{(q_0)\}\}
\end{aligned}
$$

The special positions $\top$ and $\bot$ are declared to be winning/losing for the system player, respectively, so that no successors positions of them need to be defined. All the work in updating the position in the game is deferred to the function Post. Evaluating this function is done in multiple steps. The first step is to check if the environment/input player (player 0) chose a valid move, i.e., if there exists a concrete position that is described by v for which the input player can semi-enable X. Otherwise, the move makes no sense, and we transition to position $\top$, which is a sink (i.e., has no outgoing transitions) and represents that player 0 has made a faulty move.

Then, the *Post* operator checks the system player's move. As the system player declares which patterns should not be instantiable in the next concrete position and along which run points it can promise progress, we move to position $\bot$ whenever the system

player is promising too much. In particular, the system player should be able to keep the promise for all position/concrete input combinations for which the environment player's move is valid. In all of these cases, there has to exist some concrete output that leads to a position that does not allow to instantiate any of the promised patterns. At the same time, the system player should be able to make progress with respect to any of the promised run points without violating one of these promised patterns. More formally, the Post function is defined as follows:

- We have $\mathsf{Post}(v, X, Y_P, Y_D) = \top$ **if there does not** exist some $P \subseteq \Pi$ and $x \in \mathcal{I}_I \times \mathcal{I}_B$ such that:
  - no pattern of v can be instantiated in P, **and**
  - X is the set of transitions that are semi-enabled by P and x.
- We define $\mathsf{Post}(v, X, Y_P, Y_D) = \bot$ if for every $P \subseteq \Pi$ and $x \in \mathcal{IS}$ such that
  - no pattern of v can be instantiated in P, **and**
  - X is the set of transitions that are semi-enabled by P and x,

  we do not have that for every run point $\pi_D = (q, f) \in P$ with $q \in Y_D$, there exists some $y \in \mathcal{OS}$ such that:
  - no pattern in Y_P can be instantiated in $\{\pi' \in \Pi \mid \exists \pi \in P : (\pi, (x, y), \pi') \in \delta_\Pi\}$, and
  - we have that $\{\pi' \in \Pi \mid (\pi_D, (x, y), \pi') \in \delta_\Pi\}$ is empty.
- We have $\mathsf{Post}(v, X, Y_P, Y_D) = Y_P$ in all other cases.

Theorem 2. *Let $\mathcal{A}$ be a specification for some interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$, and $\mathcal{G}_A$ be the abstract game with initial state v^{init} built from $\mathcal{A}$. If $\mathcal{G}_A$ is winning for player 1 from v^{init}, then there exists an implementation $f : \mathcal{IS}^* \to \mathcal{OS}$ such that all words w that are runs of f are accepted by $\mathcal{A}$.*

Proof. To prove the claim, we show how f can be implemented from a strategy in $\mathcal{G}_A$ that is winning for player 1. We describe f as a program that maintains two data structures: (1) the `set` of run points of $\mathcal{A}$ for the prefix of the decision sequence w observed so far, and (2) a `queue` of run points over non-accepting states in which run points for non-accepting states are queued. The implementation always keeps the set up-to-date and uses the queue for scheduling which run points of non-accepting states are to be left next. By cycling through all of them in the queue, it is ensured that we never get stuck in one of these run points along w, so that w is accepted by $\mathcal{A}$. Additionally, f traces the current position p in $\mathcal{G}_A$.

Let f' be a strategy for player 1 to win $\mathcal{G}_A$ from v^{init}. Our implementation f works as follows: Whenever the implementation obtains a new next input $x \in \mathcal{I}_B \times \mathcal{I}_I$, it computes the set of transitions X that the input x semi-activates from the current position p. Let (Y_P, Y_D) be the move that f' performs for X from p. The implementation then computes a concrete output that leads to leaving the first run point in the queue for a state in Y_D. By the definition of Post, it is made sure that we can always find a concrete output such that additionally, the successor run point set is allowed by Y_P. Note that this computation can be performed in finite time, as we are only concerned with finite sets.

As f' is winning, this means that for all non-accepting states q, we infinitely often have $q \in Y_D$ along the play. Thus, every run point for a non-accepting state is eventually left, and for a semi-one-weak automaton $\mathcal{A}$, this means that w is accepted by $\mathcal{A}$. $\square$

4.3 Computing the Transitions in the Abstract Games

While the abstract games described above only have a finite number of positions and thus can be analyzed by standard algorithms for generalized Büchi game solving [4], we only shifted the problem of dealing with an infinite number of positions in the realizability game to dealing with sets $P \subseteq \Pi$ of run points with an unbounded size in the definition of Post. To effectively compute Post in a practical realizability game building algorithm, we have to reduce reasoning about these sets to efficiently decidable problems. Note that if we manage to only reason about sets P of a bounded size, then this already suffices – as only equality and inequality of identifier values matter, we can then simply enumerate all possible equivalence relations between the identifier values.

So it remains to reduce reasoning about such sets P of unbounded size to reasoning about a bounded number of identifier values. First of all, consider checking if $\mathsf{Post}(v, X, Y_P, Y_D) = \top$ holds. We can restrict our search for P to sets of cardinality $|X|$, as we only need at most one run point per transition in order to semi-enable it, and having more run points only makes it harder to ensure that no pattern in v can be instantiated in P. In fact, we can even restrict our search to having precisely one run point (q, f) for every transition in X such that the transition starts from q.

Testing if $\mathsf{Post}(v, X, Y_P, Y_D) = \bot$ holds while only quantifying over finite sets is a bit more difficult. We apply the following idea in order to avoid having to reason over very large sets P in order not to sacrifice soundness. We again consider sets P of cardinality $|X|$ as above, and require that for every concrete input $x \in \mathcal{IS}$ such that X is the set of transitions activated from P and for every run point (q, f) in P with $q \in Y_D$, there exists a concrete output $y \in \mathcal{Y}$ such that the run point (q, f) is left under (x, y). Additionally, we quantify over all run point sets P' of size b_{max}, where b_{max} is the largest size of a pattern in $\mathcal{M}$, and require that from the concrete position P', we do not reach a position through (x, y) in which some pattern in Y_P can be instantiated if $P \cup P'$ does not violate a pattern in v and if not more than $|X|$ transitions are semi-enabled from $P \cup P'$ for X. The idea here is that the system player has to come up with a move that allows leaving any possible run points for the non-accepting states declared in Y_D and that is robust with respect to adding more run points. In a sense, we hide certain run points from the system player, but the system player knows already the patterns that cannot be instantiated in the current concrete position. This allows us to quantify only over sets of size b_{max} in P'. If the system player can choose y such that adding more "surprise" run points does not let it exceed Y_P after the next transition, then the system player has shown that it can make a robust next choice to hold the progress promise Y_D and the successor position promise Y_P after the current round. We only need to consider at most b_{max} predecessor run points for this check as for no pattern, we need more than b_{max} run points before a transition in order to violate it after a transition. By letting player 1 fix its choices before the "surprise" run points are chosen, we only have to quantify over elements in P' once for any possible pattern in Y_P that can potentially be instantiable in the concrete game position after the transition.

As a summary, we use the following *finite-step* game for testing if $\mathsf{Post}'(v, X, Y_P, Y_D) = \bot$ holds, where Post' denotes the approximate version of Post implementing the ideas from above:

1. First, the environment player chooses some run points P (one for each element in X) and concrete input x such that x semi-enables the transitions in X.
2. Then, the environment player chooses some run point $(q, f) \in P$ with $q \in Y_D$ along which the system has to make progress (only if $Y_D \neq \emptyset$).
3. It is then the system player's turn to choose some concrete output y that leads to leaving the run point chosen by the other player (if any). The environment player wins if this is not possible.
4. Finally, the environment player picks b_{max} additional run points P'. If any pattern of Y_P can be instantiated in $\{\pi' \in \Pi \mid \exists \pi \in P \cup P' : (\pi, (x, y), \pi') \in \delta_\Pi\}$ while no pattern in v is instantiable in $P \cup P'$ and X semi-enables the transitions in X for x from $P \cup P'$, the environment player wins. Otherwise the system player wins.

4.4 Applying an (ALL)QBF Solver for Efficient Reasoning in Practice

After we have reduced computing the Post' function (i.e., our approximate version of Post) to a problem over finitely many elements in the previous subsection, it makes sense to discuss how to compute Post' in practice. Observe that testing iff $\mathsf{Post}'(v, X, Y_P, Y_D) = \bot$ holds for some values of v, X, Y_P, and Y_D is the most difficult step and can be formulated as the finite-step game given above. This fact suggests that using a solver for quantified boolean formulas (QBF) is reasonable. We can encode the boolean input and output variables in x and y as simple boolean values. For the identifiers involved in the finite-step game, let $\mathcal{C} = \{c_0, c_1, \ldots, c_n\}$ be the set of identifier variables in the run points and input and output signals involved, and $c_0, c_1, \ldots, c_n$ be the order in which they are introduced. We reserve a family of boolean variables $\{e_{ij}\}_{c_i, c_j \in \mathcal{C}}$ as an *equality matrix* between them that represents which identifier variables point to the same identifier. As equality is an equivalence relation, the matrix $\{e_{ij}\}_{c_i, c_j \in \mathcal{C}}$ must represent such a relation. We assign the task to keep the matrix representing an equivalence relation to the two players in the finite-step game; whenever a player introduces a new variable c_k for $0 \leq k \leq n$ in the game, the player must assign values to $\{e_{ik}, e_{ki} \mid 0 \leq i \leq k\}$ such that $\{e_{ij}\}_{i,j \in \{0,\ldots,k\}}$ is still an equivalence relation.

We encode the QBF instance from the point of view of the system player that asks if for some given v and X, there exists some choice for Y_P and Y_D such that the system player wins the finite-step game explained above. This has the advantage that we can model Y_P and Y_D using *free* variables and apply an ALLQBF [2] solver to compute a boolean formula g that represents all valuations of the free variables that make the quantified boolean formula satisfied. From g, we can then easily enumerate all *Pareto-optimal* moves using a satisfiability (SAT) solver. We call a valuation of Y_P and Y_D Pareto-optimal if no element can be added to Y_D and no element can be added to Y_P such that the resulting valuation of the free variables in the QBF instance is still a model of it. Note that for building the abstract realizability game $\mathcal{G}_A$, we only have to consider the Pareto-optimal choices of the system player as playing non-optimal moves does not help the system player in any way.

For computing the possible values for X that do not let the environment player lose the finite-step game from some position v (i.e., computing whether $\mathsf{Post}'(v, X, Y_P, Y_D) = \top$ holds for arbitrary Y_P and Y_D), we can apply the same

equality matrix encoding. However, this time, we only need a SAT solver as there is no quantifier alternation. Again, we only enumerate the Pareto-optimal choices, i.e., the largest elements X that avoid having $\mathsf{Post}'(v, X, Y_P, Y_D) = \top$.

4.5 Completeness for Unrealizable Safety Specifications

Assume that the specification we are concerned with is of safety type, i.e., all rejecting states in the specification automaton have an unconstrained self-loop. As the (concrete) synthesis game is determined, this means that for every unrealizable such specification, there is some number k such that the environment has already won after k steps, and there is only a finite set of positions that might be visited before that. If we add enough patterns to distinguish all of these positions from all respective other positions, then after analyzing the abstract game, we can see that all positions visited before losing the game (i.e., before entering some rejecting state) only characterize one concrete position each. From this fact we can infer that the specification under concern is in fact unrealizable. Thus, by adding a post-solution abstract game analysis step to check if all abstract positions only represent one concrete position each, the algorithm can always detect unrealizable safety specifications.

5 Experimental Results

We implemented our synthesis approach, without the extensions of Sect. 4.5, in a prototype implementation written in Python, using the SAT solver PicoSAT v.957 [3] and the ALLQBF solver GhostQ 0.85 [9] as solving engines. As there is no other synthesis tool for infinite-state systems to compare against, in our evaluation, we focus on showing the applicability of our techniques on an example of practical relevance.

Case study: We synthesize a controller to let a robot automatically deliver menu items in a restaurant to guests who ordered them. We partition the floor of the restaurant into a set of regions Z and define the neighborhood relation of regions in the restaurant by an adjacency relation $R \subseteq Z \times Z$. The robot can move between adjacent regions in every computation cycle, and pick up or deliver a food item. Food is always picked up from the same region z_{pickup} (i.e., the kitchen), where we assume a tray with prepared food that is continuously replenished to be located. The first customer orders specific food items, while the other one just requests *any* food item to be delivered. Figure 5 depicts the setting.

For our reactive system, we have as interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ with $\mathcal{I}_B = \{r_{order1}, r_{order2}\}$, $\mathcal{I}_I = \{f\}$, $\mathcal{O}_B = \{m_z \mid z \in Z\} \cup \{deliver\}$, and $\mathcal{O}_I = \{pickup\}$. The specification has the following constraints:

- At every point in time, the robot is only in one region m_z, and if the region changes from one cycle to the next one, the predecessor and successor regions are connected by R.
- Whenever food item i is ordered by customer 1 (i.e., we have $f = i$ and $r_{order1} = \mathbf{true}$), then eventually, the robot picks up food item i from z_{pickup} and does not *deliver* it until it is in region 4 (i.e., $m_4 = \mathbf{true}$) at which point it should *deliver* it.

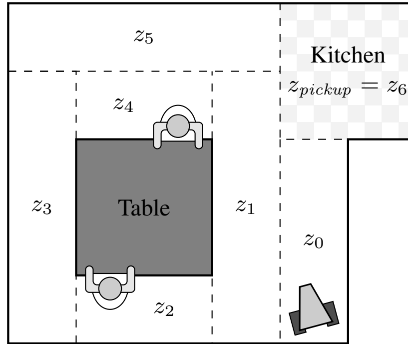


Fig. 5. A restaurant scenario with seven zones and two clients. The robot starts in zone z_0.

– The robot always picks up a food item of kind *pickup* when entering the kitchen.
– The robot must deliver a food item before entering the kitchen again. Deliveries may only take place in regions with customers.
– Whenever customer 2 orders food, then a food item is eventually brought to region 2.
– The robot does not deliver a food item to customer 1 that has not been ordered.
– New orders by a customer are ignored if there are orders by the same customer that have not yet been fulfilled.

We consider one variant of the scenario with the second customer being present, and one variant without that customer. In both cases, around 70 transitions are needed to model the respective scenario.

In addition to the robot scenario, we also considered the mutex protocol from Figure 1 and the example specification from Figure 3.

Results: Table 1 shows the experimental results for building the abstract games. The computation times were obtained on an Intel i5-3230M 2.60GHz computer running an x64-version of Linux. The actual game solving process of the abstract games always took less than 0.1 seconds.

As pattern sets, we always start with all patterns of size at most 1. For the robot waiter scenario, we find the resulting abstract games to be losing for the system player. An analysis of the scenario reveals that the reason is that we have a state q_c^1 that disallows the robot to deliver a menu item to customer 1 that is different to the one previously stored. This state has the task to check that only ordered menu items are delivered. It is entered whenever a menu item is ordered while no request is yet unfulfilled. When entering the state, the menu item requested is stored into the (single) variable in its scope. If we are in this state with two different run points, then there is no menu item that the robot can deliver. However, this is a situation that cannot occur during a play in the concrete synthesis game. By adding the pattern $\{(q_c^1, \alpha_0), (q_c^1, \alpha_1)\}$, this is taken into account in the abstract game and the setting becomes realizable. This modified pattern set is denoted as "1^+" in Table 1.

For the mutex protocol, taking all patterns of size at most 1 suffices. On the other hand, the example specification from Figure 3 is not found to be realizable with the

Table 1. Result table for the prototype implementation of our synthesis approach.

Benchmark:	1-client robot waiter		2-client robot waiter		Mutex	Example from Figure 3		
# States:	17		19		4	3		
# Transitions:	68		72		7	6		
Max. pattern size considered:	1	1^+	1	1^+	1	1	2	3
Time to build abstract game:	19m 10.2s	25m 35.5s	19m 11.5s	28m 50.0s	1.08s	0.6s	1.3s	4.5s
Number of positions in abstract game:	174	216	209	255	10	6	4	4
Number of edges in abstract game:	658	792	966	1152	14	7	4	4
Realizable:	✗	✓	✗	✓	✓	✗	✓	✓

patterns of size at most one. Thus, we also considered all patterns of size up to 2 (and additionally 3), where we removed patterns that are equivalent to other patterns in the set (such as, e.g., $\{(q_1, \alpha_1), (q_2, \alpha_2)\}$ when $\{(q_1, \alpha_2), (q_2, \alpha_1)\}$ is also present). Starting with a maximum pattern size of 2, the specification is found to be realizable.

It can be seen that the robot waiter scenario can be tackled by our approach, despite the large number of transitions in its specification automaton. While at first, this may seem surprising (after all, the number of different moves for the environment player in the abstract game is exponential in the number of automaton transitions), this success can be attributed to the idea to only enumerate the Pareto-optimal moves and use SAT and ALLQBF solvers as efficient reasoning engines, which reduces the size of the abstract game and the computation times.

6 Conclusion & Outlook

In this paper, we presented the first synthesis approach for specifications with identifier variables that is capable of deriving infinite-state implementations for cases in which these are actually needed. For showing the practical feasibility of our approach, we applied it to a robot waiter scenario. Our work can be seen as one of the first steps towards solving the problem of *reactive synthesis with data constraints*. We focused on identifiers as data type here, as these are relatively simple to handle, and thus suitable for one of the first examinations of the reactive synthesis problem with data. We conjecture that our modeling framework, i.e., universal semi-one-weak automata, remains useful when extending the data domain, as the model is both simple and powerful.

Our prototype implementation uses off-the-shelf SAT and (ALL)QBF solvers and employs a simple equivalence-matrix-based approach to deal with the identifiers in this context. We conjecture that there is still a lot of room for improvement, e.g., by optimizing the QBF encoding and using a special (ALL)QBF solver that is tuned towards finding only the Pareto-optimal variable valuations. Also, a counter-example guided abstraction refinement approach to pattern selection might be suitable.

This work was driven by investigating the class of specifications that can be supported in a practical synthesis algorithm working over an infinite data domain. Thus, the specification class and the solution algorithm are carefully aligned. It would be interesting to examine how the specification class can be further extended (such as by loosening the semi-one-weakness requirement). Additionally, it would be useful to develop a suitable specification logic from which the universal automata can be efficiently generated.

References

1. Attie, P.C., Emerson, E.A.: Synthesis of concurrent systems with many similar processes. ACM Trans. Program. Lang. Syst. **20**(1) (1998) 51–115
2. Becker, B., Ehlers, R., Lewis, M.D.T., Marin, P.: ALLQBF solving by computational learning. In Chakraborty, S., Mukund, M., eds.: ATVA. LNCS, Springer (2012) 370–384
3. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
4. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In Seidl, H., ed.: FoSSaCS. Volume 4423 of LNCS., Springer (2007) 153–167
5. Cheng, C.H., Lee, E.A.: Numerical LTL synthesis for cyber-physical systems. CoRR abs/1307.3722 (2013)
6. Dimitrova, R., Finkbeiner, B.: Abstraction refinement for games with incomplete information. In: FSTTCS. (2008) 175–186
7. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J., eds.: ICALP. Volume 2719 of LNCS., Springer (2003) 886–902
8. Jacobs, S., Bloem, R.: Parameterized synthesis. In Flanagan, C., König, B., eds.: TACAS. Volume 7214 of LNCS., Springer (2012) 362–376
9. Klieber, W., Janota, M., Marques-Silva, J., Clarke, E.M.: Solving QBF with free variables. In Schulte, C., ed.: CP. Volume 8124 of LNCS., Springer (2013) 415–431
10. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of LNCS., Springer (2006) 31–44
11. Martin, D.A.: A purely inductive proof of Borel determinacy. In: Recursion theory, Symposium on Pure Mathematics. (1982) 303–308
12. Schewe, S., Finkbeiner, B.: Bounded synthesis. In Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y., eds.: ATVA. Volume 4762 of LNCS., Springer (2007) 474–488
13. Tabuada, P.: Verification and Control of Hybrid Systems. Springer (2009)
14. Walukiewicz, I.: Pushdown processes: Games and model-checking. Inf. Comput. **164**(2) (2001) 234–263
15. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: POPL, ACM Press (1986) 184–193

Appendix

Using First-order Logic as Alternative Specification Formalism

We chose to use universal semi-one-weak automata as specification formalism in this paper, as it has the properties needed in order to decide the realizability of a specification in many practically relevant cases, but at the same time is concise enough to allow an engineer to write down specifications for systems in a simple manner.

Readers with a background in logic may however wonder if a restricted version of first-order logic would not be more appropriate. There exist many cases in which the satisfiability of a sentence in a first-order logic fragment is decidable and it may be possible to choose such a case and extend it from satisfiability to synthesis in order to obtain a decidable logic for reactive synthesis that allows to reason over identifiers.

The problem with this approach, which is the main reason why we refrained from pursuing it, is that even for very simple specifications of reactive systems, the first-order formulas for the specifications fall out of the decidable cases for satisfiability, not even mentioning the problem of generalizing the idea to realizability. Intuitively, the reason is that we have "two sources of unbounded values" in our setting, namely *time* and *identifiers*. There is no bound on the number of different identifier values that we can observe along the run of a system, and there is no time bound of the length of the system's run (in fact, we assume that it is infinite). Furthermore, we have to connect identifier values with time instants, which requires us to use *binary predicates* in first-order logic in addition to needing a binary predicate to represent the order of time. All of these facts require us to use a lot of features of first-order logic to specify them such that we easily leave the decidable fragments of first-order logic.

As an example, consider a simplified version of our mutex described in Figure 1, where the requirement that no two grants are given in successive computation cycles is omitted. We need three binary relations to describe it in first-order logic:

- $r_i(x, y)$ describes that a request with id y is given at time x,
- $g_i(x, y)$ describes that a grant with id y is given at time x, and
- $\leq(x, y)$ describes that time step x is earlier (or at the same time) than time step y.

We also have the following unary predicates:

- $r_f(x)$ describes that a request is given at time x, and
- $g_f(x)$ describes that a grant is given at time x.

The specification for our setting can now be given as $\psi = \psi_{\leq} \wedge \psi_g$ with:

$$\psi_{\leq} = \forall x, y : \leq(x, y) \leftrightarrow (\neg \leq(y, x) \vee (x = y))$$
$$\psi_g = \forall x, y \exists z : (r_i(x, y) \wedge r_f(x)) \rightarrow (g_f(z) \wedge \leq(x, z) \wedge g_i(z, y))$$

Note that we used the equality operator here. A requirement that for every time instant there can only be a request (grant) for one id, respectively, can be added as well, but is not needed for the following line of reasoning. By reusing the quantifiers, we can rewrite ψ to a first-order formula of the form $\psi = \forall x, y \exists z : \psi'$, where ψ' has two binary predicates, two unary predicates, and uses equality.

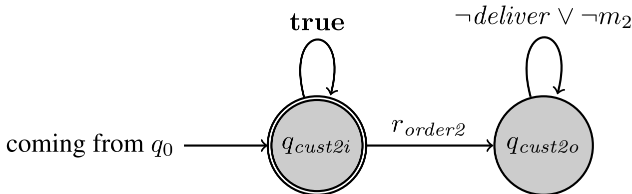
It has been shown by Warren D. Goldfarb[4] that the satisfiability of first-order sentences with this quantifier structure, equality, one binary predicate, and arbitrarily many unary predicates, is undecidable. As the unary predicates refer to boolean input/output in our setting, having many unary predicates is natural. Thus even with such a very simple specification, we naturally miss a decidable sub-class of first-order logic.

The conclusion that can be drawn from this example is that it is hard to avoid undecidability for reactive synthesis with identifiers. However, one should resist the urge to saturate the specification formalism with expressivity as a consequence, as this prevents the application of algorithms for performing synthesis in practically relevant cases anyway. The specification formalism used in the paper is thus chosen carefully to bridge sufficient expressivity for many practical cases and the possibility to apply a synthesis algorithm that works for such cases.

The Complete Specification for the Robot Waiter

Figure 6 shows the automaton that models our specification for the case of having only one customer. The automaton is automatically drawn and the transitions are labeled by the constraints and assignments. If there is no label, this means that we have neither, and thus the transition is taken unconditionally, which we marked in the rest of the paper by adding a $\mathbf{true}$ label.

For the two-customer case, the specification is extended by adding the following two states:



Note that the number of transitions in this figure is **five** as the self-loop on state $cust2o$ represents two transitions. We also remove the transition from q_{check} to q_{fail} that forbids the robot to deliver food to zone 2. Formally, this is the transition $(q_{check}, \{deliver, m_2\}, \emptyset, q_{fail})$.

An Unrealizable Variant of the Robot Waiter Specification

The example specifications considered in the main part of this paper are all realizable, but we also have an unrealizable variant of the robot waiter scenario, where we did not remove the transition mentioned in the previous paragraph. The resulting abstract game is then found to be lost for the system player (computation time for solving the abstract game: $< 0.1s$), as expected, and we obtain the following benchmarking results:

[4] See his 1984 paper "The Unsolvability of the Godel Class with Identity", The Journal of Symbolic Logic, Vol. 49, 1984, for details or "Decidable Fragments of First-Order and Fixed-Point Logic - From prefix vocabulary classes to guarded logics" by Erich Grädel, 2003, for an overview on decidable and undecidable sub-classes of first-order logic.

Benchmark:	2-client robot waiter, unrealizable variant	
# States:	19	
# Transitions:	73	
max. Pattern size considered:	1	1^+
computation time needed:	17m39.6s	26m7.4s
number of positions in abs. game:	177	219
Number of edges in abs. game:	686	820
Realizable:	✗	✗

An Abstract Game Example

For the robot waiter scenarios, the computed abstract games are far too large to include them here. However, for the example specification from Figure 1, this is not the case. Figure 7 shows the (automatically drawn) abstract game.

The figure has positions of player 0 (the environment player), which are marked by boxes, and positions of player 1 (the system player), which are the little circles.

Positions of player 0 are labeled by the set of patterns that cannot be instantiated in the concrete game positions that the abstract game position represents. Edges in the game are given as arrows, and are labeled by the corresponding action. For player 0, the action is the list of enabled transitions, whereas for player 1, we only list the progress information, as the declared pattern set, which is the other element of which an action of player 1 consists, can be seen already from the position moved to. Progress is given for all states in the specification automaton, not just for the non-accepting states.[5] For accepting states, however, we always have progress. The initial position is the one at the top.

All labels are just lists of boolean values. The following table describes how to interpret these lists:

Label type:		**Positions in V_0**	**Edges in E_0**	**Edges in E_1**
Symbol for a true value:		#	1	+
Symbol for a false value:		.	0	-
Meaning of element no.	1	$\{(q_0)\}$	$(q0, \emptyset, \emptyset, q0)$	q_0
	2	$\{(q_2)\}$	$(q0, \{r_f = \mathbf{true}\}, (r_i, a), q_1)\}$	q_1
	3	$\{(q_3)\}$	$(q1, \{g_f = \mathbf{false}\}, \emptyset, q1)$	q_2
	4	$\emptyset$	$(q1, \{g_i \neq\}, \emptyset, q1)$	q_3
	5	$\{(q_1, \{a \mapsto \alpha_0\})\}$	$(q0, \{g_f = \mathbf{true}\}, \emptyset, q2)$	
	6		$(q2, \{g_f = \mathbf{true}\}, \emptyset, q3)$	
	7		$(q3, \emptyset, \emptyset, q3)$	

[5] This is actually just an implementation detail. We found it important, however, that the figures in the appendix are actually computed by our prototype implementation. This is also the reason for the odd order of patterns in the table at the bottom of this page.

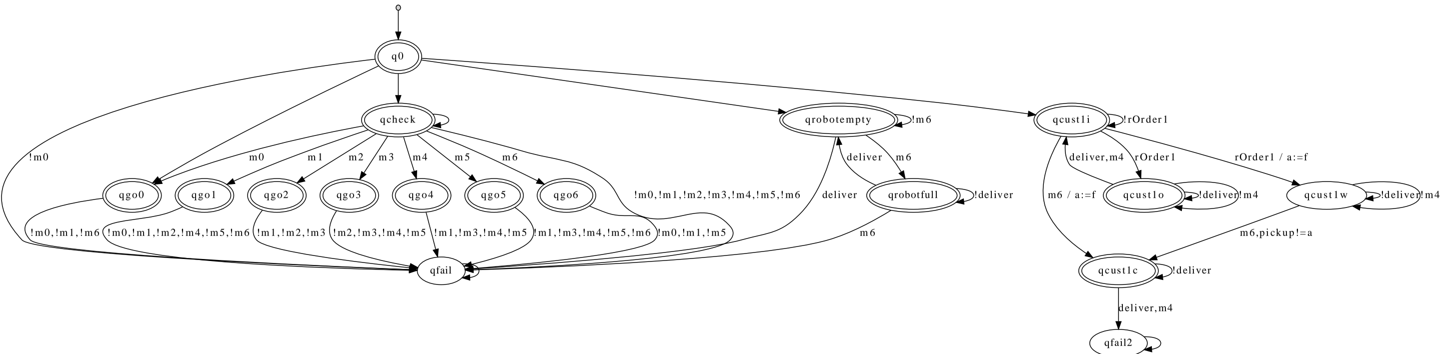


Fig. 6. Specification for the robot waiter (1-customer variant), drawn using `graphviz`. For simplicity, the specification checks the properties only from the second computation cycle onwards. We removed some transitions to keep the figure readable. In addition to the ones in the figure, we have 21 additional transitions of the form $\{(q_{check}, \{m_i, m_j\}, \emptyset, q_{fail}) \mid i, j \in \{0, \ldots, 6\}, i < j\}$, and we have 6 additional transitions of the form $\{(q_{check}, \{deliver, m_i\}, \emptyset, q_{fail}) \mid i \in \{0, 1, 2, 3, 5, 6\}\}$. These test that the robot always declares to be in only one zone at a time and that food is never delivered to a zone in which there is no customer.

The initial state is q_0. From it, there is one transition for checking if the robot starts in zone z_0. For testing if the robot only moves according to the layout of the restaurant and thus cannot skip zones, we have the states q_{go0} to q_{go6}, from which there are transitions to q_{fail} that are taken whenever the successor zone is not in the list of adjacent ones. Entering state q_{fail} makes a run of the automaton non-accepting, as it is not an accepting state and it has an unconditional self-loop. An edge from q_{check} to q_{fail} checks that the robot is always in some zone at every point in time. Together with the edges not shown in the figure, but mentioned above, this ensures that the robot is always in exactly one zone at a time (from the second computation cycle onwards). The states $q_{robotempty}$ and $q_{robotfull}$ monitor that picking up food and delivering food strictly alternates, starting with picking up food.

The five states on the right-hand side of the figure handle the requirement that ordered food, and only ordered food, is to be brought to the customer. The state q_{cust1i} represents idling for customer 1, while q_{cust1o} represents that something has been ordered, but not yet delivered. Being in state q_{cust1c} represents that the robot first has to pick up the right food item before it may deliver to customer 1. When there is no outstanding order, the state is always entered upon pickup to prevent deliveries. We may still be in that state after ordering, which is motivated by the fact that the customer most likely wants freshly made food, which the robot can only pick up *after* ordering. State q_{cust1w} checks that food is eventually delivered to the customer and branches to state q_{cust1c} whenever the robot picks up the wrong food item. Thus, checking the correct delivery of orders is done by the states q_{cust1w} and q_{cust1c} in combination.

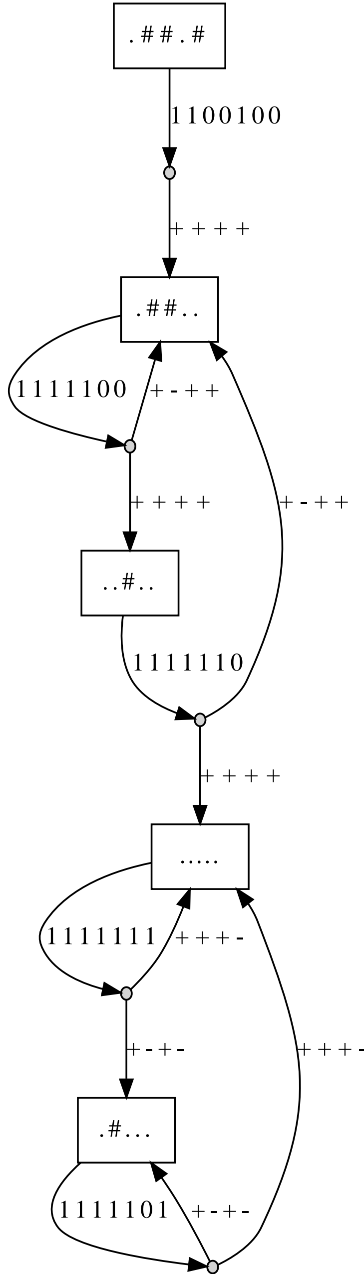


Fig. 7. An example abstract game, built from the specification in Figure 1

Element numbers in the lists always go from left to right. It can be seen from the game that once we have reached one of the bottom-most two positions of player 0, player 1 (the system player) cannot win the generalized Büchi game any more as there is no way to make progress on leaving state q_3 any more. From the position labels, we can see that these are precisely the positions in which the pattern $\{(q_3, \emptyset)\}$ does not hold (i.e., there can exist run points for q_3 in the concrete game positions that the abstract game position represents). The only way to get to one of these positions is however to make progress with respect to leaving state q_1 (i.e., having the system player choose $g_f \neq \mathbf{false}$) while the pattern $\{(q_2, \emptyset)\}$ does not hold. As we are in position q_2 along a branch of the automaton if and only if we have just seen $g_f = \mathbf{true}$, this corresponds to the case that two grants have been given successively, which the specification automaton forbids.

Note that we only have so few positions in our game due to the fact that our implementation leaves out moves by the two players that are strictly worse than other possible moves. For example, the system player can always choose to make less progress. Thus, for example, an edge in the game that is labeled by $+-++$ would have another seven sibling edges with weaker progress guarantees. Furthermore, whenever we have an edge to a position such as ".##..", we would have another similarly labeled edge to the state ".#..." as well, as that pattern set is strictly weaker, and our abstract game formulation allows the system player to declare such a weaker pattern set if it desires.

A Formal Proof of Undecidability for Synthesis from Universal One-weak Automata with Identifiers (Theorem 1)

Theorem 1. *Realizability checking for specifications that are written as semi-one-weak universal automata with identifiers is undecidable.*

Proof. We prove the claim by reducing the problem whether a deterministic Turing machine accepts the empty word to the **unrealizability** of a specification given as a semi-one-weak universal automaton with identifiers.

Assume that we are given a deterministic Turing machine $\mathcal{T} = (Q, \Gamma, \delta, q_0, F)$ with the set of states Q, the tape alphabet Γ, the transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, 0, R\}$, the initial state q_0, and the set of accepting states F. The tape alphabet contains some element $_$, which represents the empty tape cell content.

The system whose realizability we check has the interface $(\mathcal{I}_B, \mathcal{I}_I, \mathcal{O}_B, \mathcal{O}_I)$ with $\mathcal{I}_B = \{\mathit{done_phase1}\}$, $\mathcal{I}_I = \{\mathit{in_id}\}$, $\mathcal{O}_B = \{\mathit{tape_end}\} \cup \{t_0, \ldots, t_z\}$, where z is large enough to allow encoding all elements in $(Q \cup \{\bot\}) \times \Theta$ into valuations of the variables $t_0, \ldots, t_z$, and $\mathcal{O}_I = \{\mathit{out_id}\}$.

The idea of our reduction is the following: we write a specification that allows the environment to provide a sequence of identifier values to the system and once the environment sets $\mathit{done_phase1}$, the system is forced to output the Turing tape configurations of a run of the Turing machine, where the identifiers provided are used as addresses of the Turing tape cells. The tape is thus of bounded length, and the system must never output a configuration with an accepting state. However, the Turing machine is allowed to leave the tape with the tape head, at which point the Turing machine computation basically stops by not changing the tape content anymore.

As it is in the interest of the environment that supplies the input to make the system violate the specification, it is in its interest to supply a set of identifiers that is large enough to never let the tape head get out-of-bounds for the accepting run of the Turing machine (if it exists). If the Turing machine accepts, the environment can then force the system to violate the specification by providing a tape that is long enough. On the other hand, if the Turing machine does not accept a word, then the system can simulate it ad infinitum or until the tape is exceeded, so the length of the tape provided by the environment does not matter.

We take a specification ψ that consists of multiple parts, which for the time being we connect via boolean operators. We first describe the individual parts of the specification and their encoding as universal semi-one-weak automata with identifiers. How to connect the specification parts to one big semi-one-weak automaton is dealt with later.

$$\psi = \psi_{assumptions} \vee (\psi_{init} \wedge \psi_{idsucc} \wedge \psi_{tapeinit} \wedge \psi_{progress} \wedge \psi_{nonterm})$$

Let us first informally define what the parts of the specification are supposed to represent.

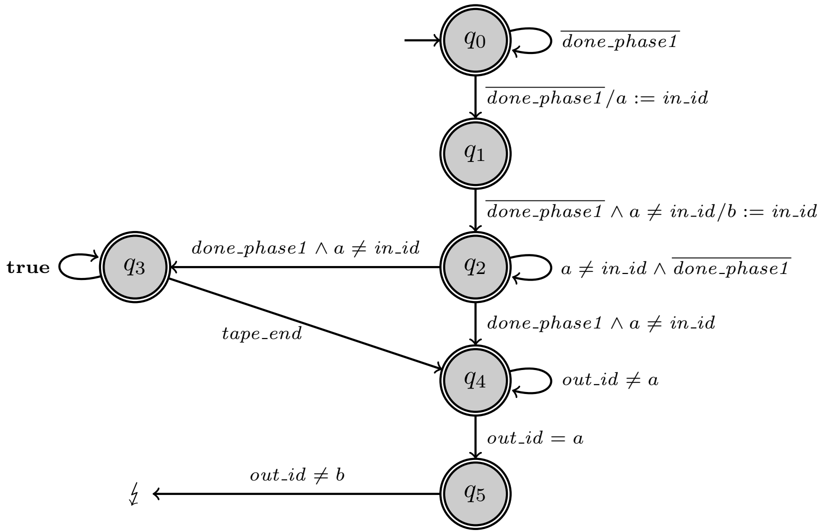
- $\psi_{assumptions}$ describes that the tape provided by the environment is not at least 3 tape cells long. We use this assumption to simplify the description of the other automata.
- ψ_{init} describes that at the start of the execution of the system, the environment can provide a sequence of identifiers, and when $done_phase1$ is then set, the system is forced to output that sequence of identifiers over and over again (separated by occurrences of the $tape_end$ bit being true). If the environment uses the same identifier twice in the sequence, then the system is allowed to cut the sequence shorter.
- ψ_{idsucc} works together with ψ_{init} to ensure that the tape boundaries are handled correctly.

Due to the fact that the system is forced to output some sequence of identifiers all the time, we can use the identifiers as addresses for positions along a Turing tape. So while the system outputs an address along out_id, the bits $\{t_0, \ldots, t_z\}$ represent the Turing tape content and the state at that position. If the Turing machine head is elsewhere, then $t_0, \ldots, t_z$ encode $\perp$ as the state component. The correctness of the Turing tape computation is ensured by the next parts of the specification:

- $\psi_{tapeinit}$ describes that along the output bits that represent the tape content, the system outputs the empty tape (plus q_0 at the tape start) initially.
- $\psi_{progress}$ describes that the tape contents evolve according to the specification of the machine, and that when the head moves out of tape boundaries, the tape head may disappear.
- $\psi_{nonterm}$ describes that the Turing machine never reaches a state in F

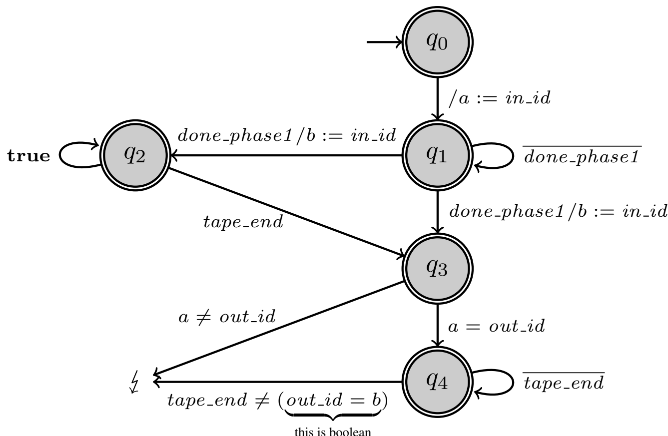
All of these specification parts can be represented as safety automata, i.e., as semi-one-weak universal automata in which the only non-accepting states are ones with self-loops for every element in $\mathcal{IS} \times \mathcal{OS}$.

Let us start by giving ψ_{init}.



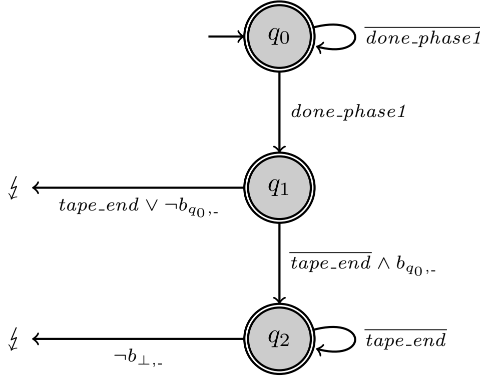
For simplicity, in this automaton and the ones to follow, we abbreviated the error state(s) by $\lightning$. An error state is a rejecting state that has self-loops for all elements in $\mathcal{IS} \times \mathcal{OS}$. The automaton effectively loops in q_0 until the environment declares that the phase in which the identifiers that are to be used as addresses for positions on the Turing tape are declared is over. Along the way, the automaton branches to q_1 and stores two successive identifiers from in_id into the variables a and b. On these runs, we are then in state q_2 that we leave when $done_phase1$ is given. From that point onwards, we move to state q_4 whenever the next tape content is to be given by the system, where we wait until we see identifier a on out_id. Whenever that happens, then b should be given next. State q_3 ensures that this shall happen every time a new tape content is printed out. Note that if the identifier a is repeated before reaching states q_3 and q_4, then a run of the automaton simply ends.

Now ψ_{init} works together with ψ_{idsucc} to ensure that the tape boundaries are dealt with correctly, as described in the following automaton:



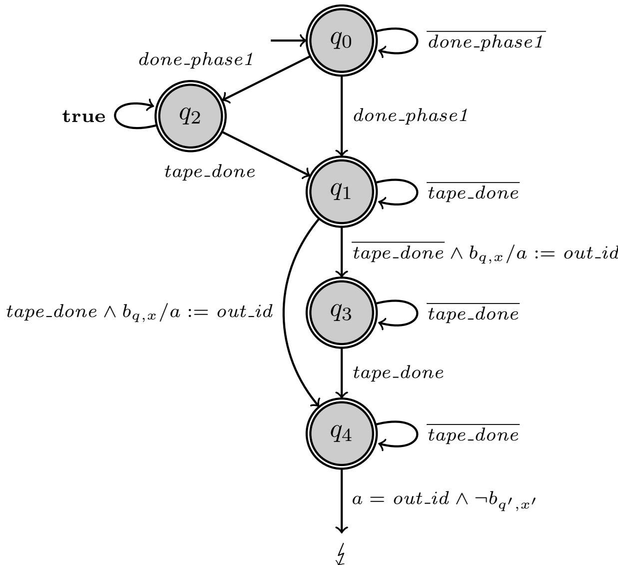
This automaton checks that the first and the last identifiers that are given by the environment during the first phase of the system's execution (i.e., until $\mathit{done_phase1}$ is set) are repeated correctly by the system (all the time).

We continue with the description of $\psi_{tapeinit}$. For some $(q, x) \in (Q \cup \{\perp\}) \times \Gamma$, let $b_{q,x}$ be a boolean formula over the variables $\{t_0, \ldots, t_z\}$ that describes that $t_0, \ldots, t_z$ represents (q, x). The automaton for $\psi_{tapeinit}$ now looks as follows:

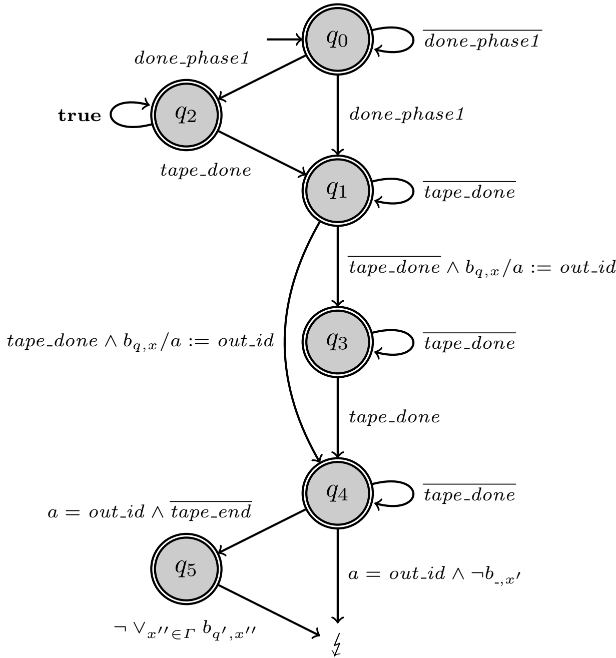


The correct evolution of the tape content is checked by $\psi_{progress}$, which consists of (1) one conjunct for every $(q, x) \in Q \times \Gamma$ that checks the transitions of the Turing machine, (2) one conjunct for every $x \in \Gamma$ that checks that Turing tape cells whose content is x are never altered if the tape head is not at the respective position, and (3) one automaton that tests that the head is always only at at most one position.

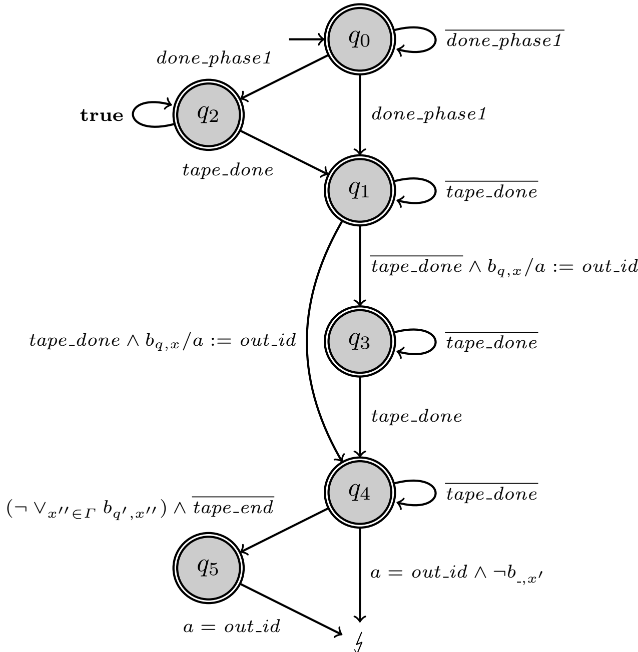
We start with the first group of conjuncts. Let $\delta(q, x) = (q', x', d)$ for some $q' \in Q$, $x \in \Gamma$, and $d \in \{L, 0, R\}$. If $d = 0$, then the automaton for (q, x) looks as follows:



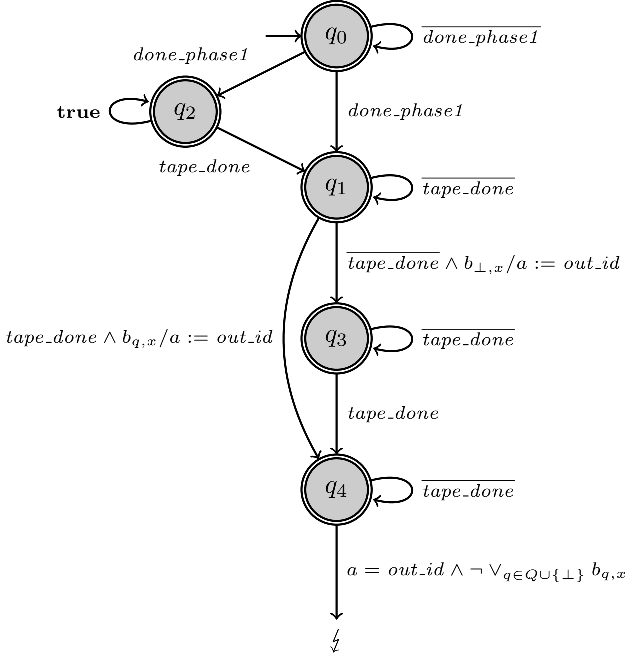
For the case that we have $d = R$, the automaton looks like this:



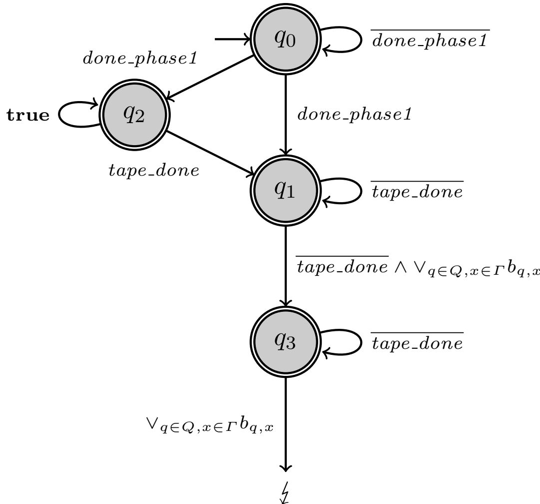
For the case that $d = L$, the automaton looks like this:



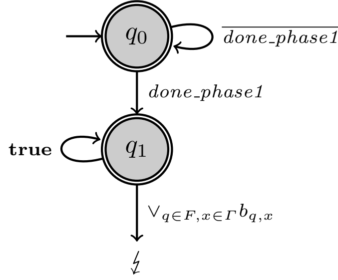
For the second group of conjuncts in $\psi_{progress}$, we instantiate one automaton of the following form for every $x \in \Gamma$:



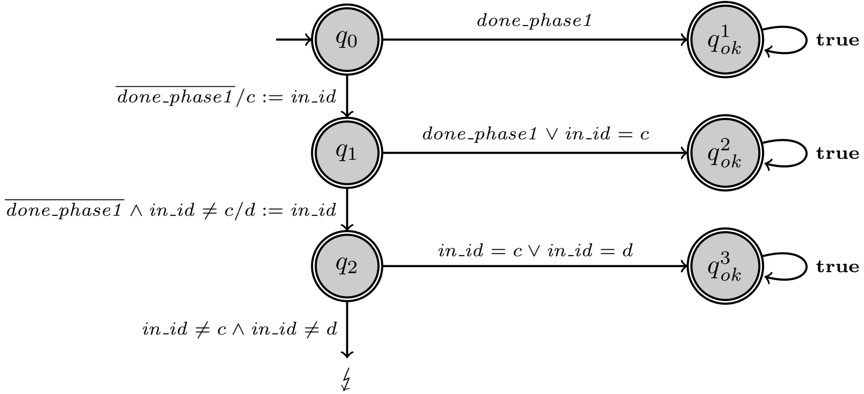
Finally, we have one automaton that checks that the tape head is always at most at one position at a time.



The specification part $\psi_{nonterm}$ now states that the Turing machine shall not eventually transition to an accepting state.



Finally, we apply some assumption over the input. Some of the automata that we have just seen only work correctly if the tape given by the environment has at least three tape cells. We use the following assumption automaton that accepts all inputs that do not satisfy these requirements:



All the specification parts in $\psi_{init} \wedge \psi_{idsucc} \wedge \psi_{tapeinit} \wedge \psi_{progress} \wedge \psi_{nonterm}$ have been given as (sets of) universal semi-one-weak automata. Computing an automaton that represents this conjunction is not difficult: we join the state spaces of the automata, introduce one additional initial state and copy all outgoing transitions from the states that are initial states in the sub-automata to the new initial state. We obtain a semi-one-weak automaton $\mathcal{A}_g$.

Let $\mathcal{A}_a$ be the automaton for $\psi_{assumptions}$. We build one automaton for the overall specification by taking the parallel product of $\mathcal{A}_g$ and $\mathcal{A}_a$. The resulting automaton is one-weak, a safety automaton, and we declare those states to be accepting that are accepting for either $\mathcal{A}_g$ or $\mathcal{A}_a$. Recall that we used ↯ to represent states that are non-accepting and self-looping for all elements in $\mathcal{IS} \times \mathcal{OS}$. Thus, only if we enter ↯ states for both of the factor automata along a run, the run is rejecting. All runs of the product automaton for which this never happens are not rejecting.

As the resulting automaton represents an unrealizable specification if and only if the Turing machine accepts with some bounded tape, and every accepting run of a Turing machine only needs a bounded tape, the claim follows.