Explainable Reactive Synthesis*

Tom Baumeister[1], Bernd Finkbeiner[2], and Hazem Torfah[3]

[1] Saarland University, Saarland Informatics Campus
s8tobaum@stud.uni-saarland.de
[2] CISPA Helmholtz Center for Information Security
finkbeiner@cispa.saarland
[3] University of California, Berkeley, USA
torfah@berkeley.edu

Abstract. Reactive synthesis transforms a specification of a reactive system, given in a temporal logic, into an implementation. The main advantage of synthesis is that it is automatic. The main disadvantage is that the implementation is usually very difficult to understand. In this paper, we present a new synthesis process that explains the synthesized implementation to the user. The process starts with a simple version of the specification and a corresponding simple implementation. Then, desired properties are added one by one, and the corresponding transformations, repairing the implementation, are explained in terms of counterexample traces. We present SAT-based algorithms for the synthesis of repairs and explanations. The algorithms are evaluated on a range of examples including benchmarks taken from the SYNTCOMP competition.

Keywords: reactive synthesis · temporal logic · SAT-based synthesis

1 Introduction

In reactive synthesis, an implementation of a reactive system is automatically constructed from its formal specification. Synthesis allows developers to define the behavior of a system in terms of a list of its desired high-level properties, delegating the detailed implementation decisions to an automatic procedure. However, the great advantage of synthesis, that it is *automatic*, can also be an obstacle, because it makes it difficult for the user to *understand* why the system reacts in a particular way. This is particularly troublesome in case the user has written an incorrect specification or forgotten to include an important property. The declarative nature of formal specifications gives the synthesis process the liberty to resolve unspecified behavior in an arbitrary way. This may result in implementations that satisfy the specification, yet still behave very differently from the developer's expectations.

* This work was partially supported by the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR: 248, 389792660), the European Research Council (ERC) Grant OSARES (No. 683300), the DARPA Assured Autonomy program, the iCyPhy center, and by Berkeley Deep drive.

In this paper, we propose a new synthesis process that, while still fully automatic, provides the user with an explanation for the decisions made by the synthesis algorithm. The *explainable synthesis* process builds the implementation incrementally, starting with a small subset of the desired properties and then adding more properties, one at a time. In each step, the algorithm presents an implementation that satisfies the currently considered specification and explains the changes that were required from the previous implementation in order to accomodate the additional property. Such an explanation consists of a counterexample trace, which demonstrates that the previous implementation violated the property, and a transformation that minimally repairs the problem.

As an example, consider the specification of a two-client arbiter in linear-time temporal logic (LTL) shown in Figure 1a. The specification consists of three properties: $\varphi_{mutex}, \varphi_{fairness}$ and $\varphi_{non\text{-}spurious}$, requiring *mutual exclusion*, i.e., there is at most one grant at a time, *fairness*, i.e., every request is eventually granted, and *non-spuriousness*, i.e., a grant is only given upon receiving a request[4]. Let us suppose that our synthesis algorithm has already produced the transition system shown in Figure 1b for the partial specification $\varphi_{mutex} \wedge \varphi_{fairness}$. This solution does not satisfy $\varphi_{\text{non-spurious}}$. To repair the transition system, the synthesis algorithm carries out the transformations depicted in Figures 1c to 1g. The transformations include a label change in the initial state and the redirection of five transitions. The last four redirections require the expansion of the transition system to two new states t_2 and t_3. The synthesis algorithm justifies the transformations with counterexamples, depicted in red in Figures 1c to 1f.

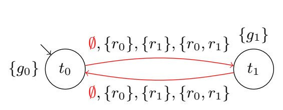
The algorithm justifies the first two transformations, (1) changing the label in the initial state to $\emptyset$ as depicted in Figure 1c and (2) redirecting the transition $(t_0, \emptyset, t_1)$ to $(t_0, \emptyset, t_0)$, as shown in Figure 1d, by a path in the transition system that violates $\varphi_{non\text{-}spurious}$, namely the path that starts with transition $(t_0, \emptyset, t_1)$. Changing the label of the initial state causes, however, a violation of $\varphi_{fairness}$, because no grant is given to client 0. This justifies giving access to a new state t_2, as shown in Figure 1e and redirecting the transition with $\{r_0\}$ from t_0 to t_2. The third transformation, leading to Figure 1f, is justified by the counterexample that, when both clients send a request at the same time, then only client 1 would be given access. Finally, the last two transformations, redirecting $(t_1, \{r_0\}, t_0)$ to $(t_1, \{r_0\}, t_3)$ and $(t_1, \{r_0, r_1\}, t_0)$ to $(t_1, \{r_0, r_1\}, t_3)$, are justified by the counterexample that if both clients alternate between sending a request then client 0 will not get a grant. This final transformation results in the transition system shown in Figure 1g, which satisfies all three properties from Figure 1a.

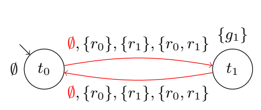
We implement the explainable synthesis approach in the setting of *bounded synthesis* [9,4]. Bounded synthesis finds a solution that is minimal in the number of states; this generalizes here to a solution that is obtained from the previous solution with a minimal number of transformations. Like bounded synthesis, we

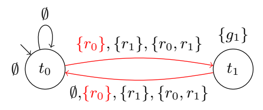
[4] The sub-formula $r_i \, \mathcal{R} \, \neg g_i$ states that initially no grant is given to client i as long as no request is received from this client. After that, the formula $\square(g_i \rightarrow r_i \vee (\bigcirc(r_i \mathcal{R} \neg g_i)))$ ensures that a grant is active only if the current request is still active, otherwise, and from this point on, no grants are given as long as no new request is received.

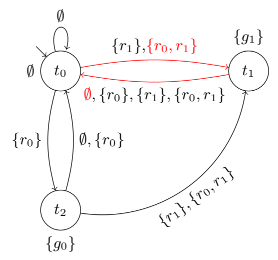
$$\varphi_{mutex} := \square(\neg g_0 \vee \neg g_1)$$
$$\varphi_{fairness} := \bigwedge_{i\in\{0,1\}} \square(r_i \rightarrow \diamondsuit g_i)$$
$$\varphi_{non\text{-}spurious} := \bigwedge_{i\in\{0,1\}} ((r_i \mathcal{R} \neg g_i) \wedge \square(g_i \rightarrow r_i \vee (\bigcirc(r_i \mathcal{R} \neg g_i))))$$

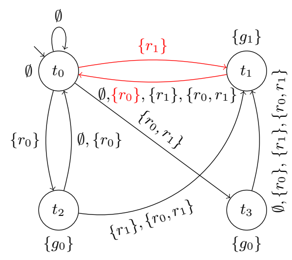
(a) Specification of a two-client arbiter


(b) An implementation for specification $\varphi_{mutex} \wedge \varphi_{fairness}$


(c) Δ_1: Changing the label of the initial state


(d) Δ_2: Redirecting the transition $(t_0, \emptyset, t_1)$ to $(t_0, \emptyset, t_0)$


(e) Δ_3: Redirecting the transition $(t_0, \{r_0\}, t_1)$ to $(t_0, \{r_0\}, t_2)$


(f) Δ_4: Redirecting the transition $(t_0, \{r_0, r_1\}, t_1)$ to $(t_0, \{r_0, r_1\}, t_3)$

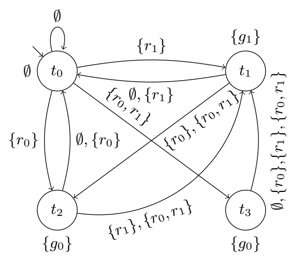

(g) Δ_5: Redirecting the transitions (t_1, v, t_0) to (t_1, v, t_3) for $v \in \{\{r_0, r_1\}, \{r_0\}\}$

Fig. 1: Using explainable reactive synthesis to synthesize an implementation for a two-client arbiter. Clients request access to the shared resource via the signals r_0 and r_1. Requests are granted by the arbiter via the signals g_0 and g_1.

use a SAT encoding to find the transition system, with additional constraints on the type and number of transformations. As explained above, the transformations involve a change of a state label or a redirection of a transition. Within the given budget of states, new states are accessed by redirecting transitions to these states. In the example in Figure 1, a budget of four states is fixed and initially unreachable states, such as t_2 and t_3, are accessed by redirecting transitions to them as done in Figure 1e and Figure 1f. For the construction of explanations, we use bounded model checking [9]. In this way, both the repair and the explanation can be ensured to be minimal. We evaluate our approach on a series of examples, including benchmarks from the SYNTCOMP competition [12].

Related Work

The importance of incremental algorithms for solving the reactive synthesis problem has quickly manifested itself in the research community after the introduction of the problem. By decomposing a synthesis problem into smaller instances and combining the results of these instances to a solution for the full problem, the hope is to provide scalable algorithms for solving the in general difficult problem [6,15,16,17,20,21]. For example, for a set of system specifications, one can construct implementations for the individual specifications and construct an implementation for the full specification by composing the results for the smaller specifications [15]. To check the realizability of a specification, one can check the realizability of gradually larger parts of the specification [21]. Refinement-based synthesis algorithms incrementally construct implementations starting with an abstraction that is gradually refined with respect to a given partial order that guarantees correctness [6,16,17,20]. The key difference between our approach and the incremental approaches mentioned above is the underlying repair process. The advantage of program repair is that it constructs an implementation that is close to the original erroneous implementation. In our approach, this makes it possible to derive explanations that justify the repairs applied to the previous implementation. Other repair problems for temporal logics have previously been considered in [13,3]. In [13], an expression or a left-hand side of an assignment is assumed to be erroneous and replaced by one that satisfies the specification. In [3], the repair removes edges from the transition system. By contrast, our repair algorithm changes labels of states and redirects transitions.

A completely different approach to make synthesized implementation more understandable is by posing conditions on the structure of synthesized implementations [14,8]. Bounded synthesis [9] allows us to bound the size of the constructed implementation. Bounded cycle synthesis [7] additionally bounds the number of cycles in the implementation. Skeleton synthesis [10] establishes the relations between the specification and the synthesized implementation to clarify which parts of the implementation are determined by the specification and which ones where chosen by the synthesis process.

2 Preliminaries

Linear-time Temporal Logic: As specification language, we use Linear-Time Temporal Logic (LTL) [19], with the usual temporal operators Next $\bigcirc$, Until $\mathcal{U}$ and the derived operators Release $\mathcal{R}$, which is the dual operator of $\mathcal{U}$, Eventually $\diamondsuit$ and Globally $\square$. Informally, the Release operator $\varphi_1 \mathcal{R} \varphi_2$ says that φ_2 holds in every step until φ_1 *releases* this condition. LTL formulas defining specifications for reactive systems are defined over a set of atomic propositions $AP = I \cup O$, which is partitioned into a set I of input propositions and a set O of output propositions. We denote the satisfaction of an LTL formula φ by an infinite sequence $\sigma \colon \mathbb{N} \to 2^{AP}$ of valuations of the atomic propositions by $\sigma \vDash \varphi$. For an LTL formula φ we define the language $\mathcal{L}(\varphi)$ by the set $\{\sigma \in (\mathbb{N} \to 2^{AP}) \mid \sigma \vDash \varphi\}$.

Implementations: We represent implementations as *labeled transition systems.* For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-transition system is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a finite set of states T, an initial state $t_0 \in T$, a transition function $\tau\colon T \times \Upsilon \rightarrow T$, and a labeling function $o\colon T \rightarrow \Sigma$. For a set I of input propositions and a set O of output propositions, we represent reactive systems as 2^O-labeled 2^I-transition systems. For reactive systems, a *path* in $\mathcal{T}$ is a sequence $\pi \in \mathbb{N} \rightarrow T \times 2^I$ of states and directions that follows the transition function, i.e., for all $i \in \mathbb{N}$, if $\pi(i) = (t_i, e_i)$ and $\pi(i+1) = (t_{i+1}, e_{i+1})$, then $t_{i+1} = \tau(t_i, e_i)$. We call a path initial if it starts with the initial state: $\pi(0) = (t_0, e)$ for some $e \in 2^I$. We denote the set of initial paths of $\mathcal{T}$ by $\mathit{Paths}(\mathcal{T})$. For a path $\pi \in \mathit{Paths}(\mathcal{T})$, we denote the sequence $\sigma_\pi\colon i \mapsto o(\pi(i))$, where $o(t,e) = (o(t) \cup e)$ by the *trace* of π. We call the set of traces of the paths of a transition system $\mathcal{T}$ the language of $\mathcal{T}$, denoted by $\mathcal{L}(\mathcal{T})$.

For a given finite sequence $v^* \in (2^I)^*$, we denote the transitions sequence where we reach a state t' from state t after applying the transition function τ for every letter in v^* starting in t by $\tau^*(t, v^*) = t'$. The size of a transition system is the size of its set of states, which we denote by $|\mathcal{T}|$.

For a set of atomic propositions $AP = I \cup O$, we say that a 2^O-labeled 2^I-transition system $\mathcal{T}$ satisfies an LTL formula φ, if and only if $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(\varphi)$, i.e., every trace of $\mathcal{T}$ satisfies φ. In this case, we call $\mathcal{T}$ a model of φ.

3 Minimal Repairs and Explanations

In this section, we lay the foundation for explainable reactive synthesis. We formally define the transformations that are performed by our repair algorithm and determine the complexity of finding a minimal repair, i.e., a repair with the fewest number of transformations, with respect to a given transition system and an LTL specification and show how repairs can be explained by counterexamples that justify the repair.

3.1 Generating Minimal Repairs

For a 2^O-labeled 2^I-transition system $\mathcal{T} = (T, t_0, \tau, o)$, an *operation* Δ is either a *change of a state labeling* or a *redirection of a transition* in $\mathcal{T}$. We denote the transition system $\mathcal{T}'$ that results from applying an operation Δ to the transition system $\mathcal{T}$ by $\mathcal{T}' = \mathit{apply}(\mathcal{T}, \Delta)$.

A state labeling change is denoted by a tuple $\Delta_{\text{label}} = (t, v)$, where $t \in T$ and $v \in 2^O$ defines the new output v of state t. The transition system $\mathcal{T}' = \mathit{apply}(\mathcal{T}, \Delta_{\text{label}})$ is defined by $\mathcal{T}' = (T, t_0, \tau, o')$, where $o'(t) = v$ and $o'(t') = o(t')$ for all $t' \in T$ with $t' \neq t$.

A transition redirection is denoted by a tuple $\Delta_{\text{transition}} = (t, t', V)$, where $t, t' \in T$ and $V \subseteq 2^I$. For a transition redirection operation $\Delta_{\text{transition}} = (t, t', V)$, the transition system $\mathcal{T}' = \mathit{apply}(\mathcal{T}, \Delta_{\text{transition}})$ is defined by $\mathcal{T}' = (T, t_0, \tau', o)$, with $\tau'(t, v) = t'$ for $v \in V$ and $\tau'(t, v) = \tau(t, v)$ for $v \notin V$. For $t'' \neq t$ and $v \in 2^I$, $\tau'(t'', v) = \tau(t'', v)$.

A finite set of operations ξ is called a *transformation.* A transformation ξ is *consistent* if there is no transition system $\mathcal{T}$ and $\Delta_1, \Delta_2 \in \xi$ such that $\mathit{apply}(\mathit{apply}(\mathcal{T}, \Delta_1), \Delta_2) \neq \mathit{apply}(\mathit{apply}(\mathcal{T}, \Delta_2), \Delta_1)$, i.e. the resulting transition system does not differ depending on the order in which operations are applied. For a consistent transformation ξ, we denote the 2^O-labeled 2^I-transition system $\mathcal{T}'$ that we reach after applying every operation in ξ starting with a 2^O-labeled 2^I-transition system $\mathcal{T}$ by $\mathcal{T}' = \mathit{apply}^*(\mathcal{T}, \xi)$.

Note that there is no operation which explicitly adds a new state. In the example in Figure 1, we assume a fixed number of available states (some that might be unreachable in the initial transition system). We reach new states by using a transition redirection operation to these states.

Definition 1 (Minimal Repair). *For an* LTL*-formula φ over $AP = I \cup O$ and a 2^O-labeled 2^I-transition system $\mathcal{T}$, a consistent transformation ξ is a* repair *for $\mathcal{T}$ and φ if $\mathit{apply}^*(\mathcal{T}, \xi) \vDash \varphi$. A repair ξ is* minimal *if there is no repair ξ' with $|\xi'| < |\xi|$.*

Example 1. The arbiter Arb_1 in Figure 1c can be obtained from the round-robin arbiter Arb_0, shown in Figure 1b, by applying $\Delta_{\text{label}} = (t_0, \emptyset)$, i.e. $\mathit{Arb}_1 = \mathit{apply}(\mathit{Arb}_0, \Delta_{\text{label}})$. Arbiter Arb_3, depicted in Figure 1e is obtained from Arb_1 with the transformation $\xi_1 = \{\Delta_{\text{transition1}}, \Delta_{\text{transition2}}\}$ where $\Delta_{\text{transition1}} = (t_0, t_0, \{\emptyset\})$ and $\Delta_{\text{transition2}} = (t_0, t_2, \{\{r_0\}\})$ such that $\mathit{apply}^*(\mathit{Arb}_1, \xi_1) = \mathit{Arb}_3$. A minimal repair for Arb_0 and $\varphi_{\mathit{mutex}} \wedge \varphi_{\mathit{fairness}} \wedge \varphi_{\mathit{non\text{-}spurious}}$, defined in Section 1, is $\xi_2 = \{\Delta_{\text{label}}, \Delta_{\text{transition1}}, \Delta_{\text{transition2}}, \Delta_{\text{transition3}}, \Delta_{\text{transition4}}\}$ with $\Delta_{\text{transition3}} = (t_0, t_3, \{\{r_0, r_1\}\})$ and $\Delta_{\text{transition}} = (t_1, t_2, \{\{r_0\}, \{r_0, r_1\}\})$. The resulting full arbiter Arb_5 is depicted in Figure 1g, i.e. $\mathit{apply}^*(\mathit{Arb}_0, \xi_2) = \mathit{Arb}_5$.

We are interested in finding minimal repairs. The *minimal repair synthesis problem* is defined as follows.

Problem 1 (Minimal Repair Synthesis) *Let φ be an* LTL*-formula over a set of atomic propositions $AP = I \cup O$ and $\mathcal{T}$ be a 2^O-labeled 2^I-transition system. Find a minimal repair for φ and $\mathcal{T}$?*

In the next lemma, we show that for a fixed number of operations, the problem of checking if there is a repair is polynomial in the size of the transition system and exponential in the number of operations. For a small number of operations, finding a repair is cheaper than synthesizing a new system, which is 2EXPTIME-complete in the size of the specification [18].

Lemma 1. *For an* LTL*-formula φ, a 2^O-labeled 2^I-transition system $\mathcal{T}$, and a bound k, deciding whether there exists a repair ξ for $\mathcal{T}$ and φ with $|\xi| = k$ can be done in time polynomial in the size of $\mathcal{T}$, exponential in k, and space polynomial in the length of φ.*

Proof. Checking for a transformation ξ if $\mathit{apply}^*(\mathcal{T}, \xi) \vDash \varphi$ is PSPACE-complete [22]. There are $|\mathcal{T}| \cdot 2^{|O|}$ different state labeling operations and $|\mathcal{T}|^2 \cdot 2^{|I|}$ transition redirections. Thus, the number of transformations ξ with $|\xi| = k$ is bounded by

$\mathcal{O}((|\mathcal{T}|^2)^k)$. Hence, deciding the existence of such a repair is polynomial in $|\mathcal{T}|$ and exponential in k. $\square$

The size of a minimal repair is bounded by a polynomial in the size of the transition system under scrutiny. Thus, the minimal repair synthesis problem can be solved in time at most exponential in $|\mathcal{T}|$. In most cases, we are interested in small repairs resulting in complexities that are polynomial in $|\mathcal{T}|$.

Theorem 1. *For an* LTL*-formula φ, a 2^O-labeled 2^I-transition system $\mathcal{T}$, finding a minimal repair for $\mathcal{T}$ and φ can be done in time exponential in the size of $\mathcal{T}$, and space polynomial in the length of φ.*

3.2 Generating Explanations

For an LTL-formula φ over $AP = I \cup O$, a transformation ξ for a 2^O-labeled 2^I-system $\mathcal{T}$ is *justified* by a counterexample σ if $\sigma \nvDash \varphi$, $\sigma \in \mathcal{L}(\mathcal{T})$ and $\sigma \notin \mathcal{L}(apply^*(\mathcal{T}, \xi))$. We call σ a *justification* for ξ. A transformation ξ is called *justifiable* if there exists a justification σ for ξ.

A transformation ξ for $\mathcal{T}$ and φ is *minimally* justified by σ if ξ is justified by σ and there is no $\xi' \subset \xi$ where σ is a justification for ξ'. If a transformation ξ is minimally justified by a counterexample σ, we call σ a *minimal* justification.

Definition 2 (Explanation). *For an* LTL*-formula φ over $AP = I \cup O$, an initial 2^O-labeled 2^I-transition system $\mathcal{T}$, and a minimal repair ξ, an* explanation *ex is defined as a sequence of pairs of transformations and counterexamples. For an explanation $ex = (\xi_1, \sigma_1), \ldots, (\xi_n, \sigma_n)$, it holds that all transformations $\xi_1, \ldots, \xi_n$ are disjoint, $\xi = \bigcup_{1 \le i \le n} \xi_i$, and each transformation ξ_i with $1 \le i \le n$ is minimally justified by σ_i for $apply^*(\mathcal{T}, \bigcup_{1 \le j < i} \xi_j)$ and φ.*

Example 2. Let $\varphi_1 = g \wedge \bigcirc \neg g$ over $I = \{r\}$ and $O = \{g\}$ and consider the 2^O-labeled 2^I-transition system $\mathcal{T}$ with states $\{t_0, t_1\}$, depicted in Figure 2. For $\mathcal{T}$ and φ_1, the transformation ξ with $\xi = \{\Delta_{\text{transition}}\}$ where $\Delta_{\text{transition}} = (t_0, t_1, \{\{g\}, \emptyset\})$, is not justifiable because $\mathcal{L}(\mathcal{T}) = \mathcal{L}(apply^*(\mathcal{T}, \xi))$. For our running example, introduced in Section 1, the transformation $\xi_1 = \{\Delta_{\text{label}}\}$ that is defined in Example 1, is justifiable for the round-robin arbiter Arb_0 and $\varphi_{\text{mutex}} \wedge \varphi_{\text{fairness}} \wedge \varphi_{\text{non-spurious}}$. It is justified by the counterexample $\sigma_1 = (\{g_0\} \cup \emptyset, \{g_1\} \cup \emptyset)^\omega$, indicated by the red arrows in Figure 1b. Further, σ_1 is a minimal justification. The transformation $\xi_2 = \{\Delta_{\text{label}}, \Delta_{\text{transition1}}\}$ for Arb_0 is not minimally justified by σ_1 as σ_1 is a justification for ξ_1 and $\xi_1 \subset \xi_2$. An explanation ex for Arb_0, the LTL-formula $\varphi_{\text{mutex}} \wedge \varphi_{\text{fairness}} \wedge \varphi_{\text{non-spurious}}$ and the minimal repair $\xi_3 = \{\Delta_{\text{label}}, \Delta_{\text{transition1}}, \Delta_{\text{transition2}}, \Delta_{\text{transition3}}, \Delta_{\text{transition4}}\}$ is $ex = (\Delta_{\text{label}}, \sigma_1), (\Delta_{\text{transition1}}, \sigma_2), (\Delta_{\text{transition2}}, \sigma_3), (\Delta_{\text{transition3}}, \sigma_4), (\Delta_{\text{transition4}}, \sigma_5)$

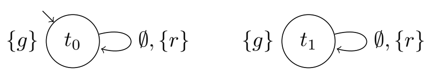


Fig. 2: A transition system over $I = \{r\}$ and $O = \{g\}$ that is not a model of φ_1.

with $\sigma_2 = (\emptyset \cup \emptyset, \{g_1\} \cup \emptyset)^\omega$, $\sigma_3 = (\emptyset \cup \{r_0\}, \{g_1\} \cup \{r_0\})^\omega$, $\sigma_4 = (\emptyset \cup \{r_0, r_1\}, \{g_1\} \cup \emptyset)^\omega$ and $\sigma_5 = (\emptyset \cup \{r_1\}, \{g_1\} \cup \{r_0\})^\omega$. The different justifications are indicated in the subfigures of Figure 1.

In the next theorem, we show that there exists an explanation for every minimal repair.

Theorem 2. *For every minimal repair ξ for an* LTL*-formula φ over $AP = I \cup O$ and a 2^O-labeled 2^I-transition system $\mathcal{T}$, there exists an explanation.*

Proof. Let $\xi = \{\Delta_1, \dots, \Delta_n\}$ be a minimal repair for the LTL-formula φ and the transition system $\mathcal{T}$. An explanation ex can be constructed as follows. Let $\sigma \in \mathcal{L}(\mathcal{T})$ with $\sigma \nvDash \varphi$. Since ξ is a minimal repair, $\sigma \notin \mathcal{L}(apply^*(\mathcal{T}, \xi))$. The smallest subset $\xi' \subseteq \xi$ with $\sigma \notin \mathcal{L}(apply^*(\mathcal{T}, \xi'))$ is minimally justified by σ. Thus (ξ', σ) is the first element of the explanation ex. For the remaining operations in $\xi \backslash \xi'$, we proceed analogously. The counterexample σ is now determined for $apply^*(\mathcal{T}, \xi')$. The construction is finished if either every transformation is minimally justified by a counterexample and there is no operation left or there is no justification for a transformation which clearly contradicts that ξ is a minimal repair. Hence, ex is an explanation for ξ. $\square$

From the last theorem we know that we can find an explanation for every minimal repair. It is however important to notice that it is not necessarily the case that we can find justifications for each singleton transformation in the repair as shown by the following example. Let $\varphi_2 = \neg g \wedge \bigcirc \neg g \wedge ((\square \neg r) \rightarrow \bigcirc \bigcirc g)$ over $I = \{r\}$, $O = \{g\}$ and consider the 2^O-labeled 2^I-transition system $\mathcal{T}$ with the set of states $\{t_0, t_1, t_2\}$, depicted in Figure 3. For φ_2 and $\mathcal{T}$, the transformation ξ with $\xi = \{\Delta_{\text{transition1}}, \Delta_{\text{transition2}}\}$ where $\Delta_{\text{transition1}} = (t_0, t_1, \{\emptyset\})$, and $\Delta_{\text{transition2}} = (t_1, t_2, \{\emptyset\})$, is a minimal repair. The counterexample $\sigma = \emptyset^\omega$ is the only one with $\sigma \in \mathcal{L}(\mathcal{T})$. For an explanation $ex = (\xi_1, \sigma_1), (\xi_2, \sigma_2)$ where ξ_i is a singleton, for all $1 \leq i \leq 2$, either $\xi_1 = \{\Delta_{\text{transition1}}\}$ or $\xi_1 = \{\Delta_{\text{transition2}}\}$. However, in both cases, $\sigma \in \mathcal{L}(apply^*(\mathcal{T}, \xi_1))$. Thus, there are minimal repairs where not every operation can be justified on its own. Furthermore, it should be noted that explanations are not unique as there can exist different justifications for the same transformation, i.e. there can exist multiple different explanations for the same minimal repair. For the round-robin arbiter in Figure 1b and the specification $\varphi_{mutex} \wedge \varphi_{fairness} \wedge \varphi_{non\text{-}spurious}$, the transformation $\xi_1 = \{\Delta_{\text{label}}\}$ can be minimally justified by $(\{g_0\} \cup \emptyset, \{g_1\} \cup \emptyset)^\omega$ and by $(\{g_0\} \cup \{r_1\}, \{g_1\} \cup \emptyset)^\omega$.

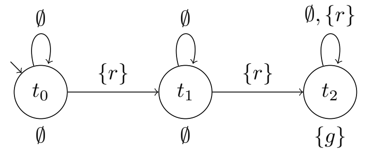


Fig. 3: A transition system over $I = \{r\}$ and $O = \{g\}$ that is not a model of φ_2.

We refer to the problem of finding an explanation for a minimal repair as the the *explanation synthesis problem.*

Algorithm 1 MinimalRepair($\mathcal{T}, \varphi$)

```
1: $left \leftarrow 0$
2: $right \leftarrow |\mathcal{T}| + |\mathcal{T}| \cdot |\mathcal{T}|$
3: $exist \leftarrow false$
4: while $left < right$ do
5:     $k \leftarrow \lfloor \frac{left+right}{2} \rfloor$
6:     $(found, \xi) \leftarrow$ Repair$(\mathcal{T}, \varphi, k)$
7:     if $found$ then
8:         $right \leftarrow k - 1$
9:         $\xi_{min} \leftarrow \xi$
10:        $exists \leftarrow true$
11:    else
12:        $left \leftarrow k + 1$
13: $(found', \xi') \leftarrow$ Repair$(\mathcal{T}, \varphi, left)$
14: if $!exists$ then
15:     return no minimal repair exists
16: if $found'$ then
17:     return $\xi'$
18: else
19:     return $\xi_{min}$
```

Problem 2 (Explanation Synthesis) *Let φ be an LTL-formula over $AP = I \cup O$, $\mathcal{T}$ be a 2^O-labeled 2^I-transition system and ξ be a minimal repair. Find an explanation ex for φ, $\mathcal{T}$ and ξ.*

4 SAT-based Algorithms for Minimal Repair and Explanation Synthesis

In this section, we present SAT-based algorithms to solve the minimal repair synthesis problem and the explanation synthesis problem.

4.1 Generating Minimal Repairs

The procedure MinimalRepair($\mathcal{T}, \varphi$), shown in Algorithm 1, solves the minimal repair synthesis problem. For a given LTL-formula φ over $AP = I \cup O$ and 2^O-labeled 2^I-transition system $\mathcal{T}$, Algorithm 1 constructs a minimal repair ξ if one exists. We use binary search to find the minimal number k of required operations. The possible number of operations can be bounded by $|\mathcal{T}| + |\mathcal{T}| \cdot |\mathcal{T}|$ as there are only $|\mathcal{T}|$ state labelings and $|\mathcal{T}| \cdot |\mathcal{T}|$ transition redirects. Checking whether there is a transformation ξ with $|\xi| \leq k$ such that $apply^*(\mathcal{T}, \xi) \vDash \varphi$ is done by using the procedure Repair($\mathcal{T}, \varphi, k$) which is explained next.

Repair($\mathcal{T}, \varphi, k$) : To check whether there is a repair ξ for a 2^O-labeled 2^I-transition system $\mathcal{T}$ with k operations, we need to ensure that the resulting transition system is a model for φ, i.e. $apply^*(\mathcal{T}, \xi) \vDash \varphi$. To check the existence of a transition system $\mathcal{T}'$ with bound $n = |\mathcal{T}'|$ that implements φ, we use the SAT-based encoding of bounded synthesis [4]. Bounded synthesis is a synthesis procedure for LTL-formulas that produces size-optimal transition systems [9]. For a given LTL formula φ, a universal co-Büchi automaton $\mathcal{A}$ that accepts

$$\begin{aligned}\phi_{cost} = & \bigwedge_{0\leq t,n<|\mathcal{T}|, c\leq k} \mathit{rdTrans}_{t,n,c} \wedge \mathit{notRdTrans}_{t,n,c} \wedge \neg \mathit{cost}_{t,n,k+1} \\ & \bigwedge_{0\leq t<|\mathcal{T}|, c\leq k} \mathit{changeLabel}_{t,c} \wedge \mathit{notChangeLabel}_{t,c} \wedge \neg \mathit{cost}_{t,|\mathcal{T}|,k+1}\end{aligned}$$

$$\mathit{rdTrans}_{t,n,c} = \begin{cases} \mathit{trans}_{0,0} \rightarrow \mathit{cost}_{0,0,1} & \text{if } t = 0 \wedge n = 0 \\ \mathit{cost}_{t-1,|\mathcal{T}|,c} \wedge \mathit{trans}_{t,n} \rightarrow \mathit{cost}_{t,n,c+1} & \text{if } t > 0 \wedge n = 0 \\ \mathit{cost}_{t,n-1,c} \wedge \mathit{trans}_{t,n} \rightarrow \mathit{cost}_{t,n,c+1} & \text{if } n > 0 \end{cases}$$

$$\mathit{notRdTrans}_{t,n,c} = \begin{cases} \neg\mathit{trans}_{0,0} \rightarrow \mathit{cost}_{0,0,0} & \text{if } t = 0 \wedge n = 0 \\ \mathit{cost}_{t-1,|\mathcal{T}|,c} \wedge \neg\mathit{trans}_{t,n} \rightarrow \mathit{cost}_{t,n,c} & \text{if } t > 0 \wedge n = 0 \\ \mathit{cost}_{t,n-1,c} \wedge \neg\mathit{trans}_{t,n} \rightarrow \mathit{cost}_{t,n,c} & \text{if } n > 0 \end{cases}$$

$$\begin{aligned}\mathit{changeLabel}_{t,c} &= \mathit{cost}_{t,|\mathcal{T}|-1,c} \wedge \mathit{label}_t \rightarrow \mathit{cost}_{t,|\mathcal{T}|,c+1} \\ \mathit{notChangeLabel}_{t,c} &= \mathit{cost}_{t,|\mathcal{T}|-1,c} \wedge \neg\mathit{label}_t \rightarrow \mathit{cost}_{t,|\mathcal{T}|,c}\end{aligned}$$

$$\begin{aligned}\mathit{label}_t &= \bigvee_{o\in O} \begin{cases} o'_t & \text{if } o \notin o(t) \\ \neg o'_t & \text{if } o \in o(t) \end{cases} \\ \mathit{trans}_{t,t'} &= \bigvee_{i\in 2^I} \begin{cases} \tau'_{t,i,t'} & \text{if } \tau(t,i) \neq t' \\ \bot & \text{if } \tau(t,i) = t' \end{cases}\end{aligned}$$

Fig. 4: The constraint ϕ_{cost} ensures that at most k operations are applied.

$\mathcal{L}(\varphi)$ is constructed. A transition system $\mathcal{T}$ satisfies φ if every path of the run graph, i.e. the product of $\mathcal{T}$ and $\mathcal{A}$, visits a rejecting state only finitely often. An annotation function λ confirms that this is the case. The bounded synthesis approach constructs a transition system with bound n by solving a constraint system that checks the existence of a transition system $\mathcal{T}$ and a valid annotation function λ. In the bounded synthesis constraint system for the 2^O-labeled 2^I-transition system $\mathcal{T}' = (T, t_0, \tau', o')$, the transition function τ' is represented by a variable $\tau'_{t,i,t'}$ for every $t, t' \in T$ and $i \in 2^I$. The variable $\tau'_{t,i,t'}$ is true if and only if $\tau'(t,i) = t'$. The labeling function o' is represented by a variable o'_t for every $o \in O$ and $t \in T$ and it holds that o'_t is true if and only if $o \in o'(t)$. For simplicity, states are represented by natural numbers.

To ensure that the transition system $\mathcal{T}'$ can be obtained with at most k operations from a given transition system $\mathcal{T} = (T, t_0, \tau, o)$, the bounded synthesis encoding is extended with the additional constraint ϕ_{cost} shown in Figure 4. For states t, t', the constraint $\mathit{trans}_{t,t'}$ holds iff there is a redirected transition from t to t', i.e. there exists an $i \in 2^I$ with $\tau'(t,i) = t'$ and $\tau(t,i) \neq t'$. The constraint label_t holds iff the state labeling of state t is changed, i.e. $o(t) \neq o'(t)$. To count

Algorithm 2 EXPLANATION($\mathcal{T}, \varphi, \xi$)

```
1:  $\mathcal{T}_{old} \leftarrow \mathcal{T}$
2:  $ex \leftarrow ()$
3:  while $\xi \neq \emptyset$ do
4:      $\sigma \leftarrow \mathrm{BMC}(\mathcal{T}_{old}, \varphi)$
5:      $minimal \leftarrow false$
6:      $\xi' \leftarrow \xi$
7:      while $!minimal$ do
8:          $minimal \leftarrow true$
9:          for $\Delta \in \xi'$ do
10:             $\mathcal{T}_{new} \leftarrow apply^*(\mathcal{T}_{old}, \xi' \backslash \{\Delta\})$
11:             if $\sigma \notin \mathcal{L}(\mathcal{T}_{new})$ then
12:                 $minimal \leftarrow false$
13:                 $\xi' \leftarrow \xi' \backslash \{\Delta\}$
14:     $\mathcal{T}_{old} \leftarrow apply^*(\mathcal{T}_{old}, \xi')$
15:     $ex \leftarrow ex.\textsc{Append}(\xi', \sigma)$
16:     $\xi \leftarrow \xi \backslash \xi'$
17: return $ex$
```

the number of applied operations, we use an implicit ordering over all the possible operations: starting from state 0, we first consider all potential transition redirects to states $0, 1, \ldots, |\mathcal{T}| - 1$, then the potential state label change of state 0, then the transition redirects from state 1, and so on. For state t and operation n, where n ranges from 0 to $|\mathcal{T}|$ (where $n < |\mathcal{T}|$ refers to the transition redirect operation to state n and $n = |\mathcal{T}|$ refers to the state label change operation), the variable $cost_{t,n,c}$ is true if the number of applied operations so far is c. This bookkeeping is done by constraints $rdTrans_{t,n,c}$, $notRdTrans_{t,n,c}$, $changeLabel_{t,c}$ and $notChangeLabel_{t,c}$. Constraints $rdTrans$ and $notRdTrans$ account for the presence and absence, respectively, of transition redirects, constraints $changeLabel_{t,c}$ and $notChangeLabel_{t,c}$ for the presence and absence of state label changes. In order to bound the total number of operations by k, ϕ_{cost} requires that $cost_{t,n,k+1}$ is false for all states t and operations n.

In the next theorem, we state the size of the resulting constraint system, based on the size of the bounded synthesis constraint system given in [4].

Theorem 3. *The size of the constraint system is in* $\mathcal{O}(nm^2 \cdot 2^{|I|} \cdot (|\delta_{q,q'}| + n\log(nm)) + kn^2)$ *and the number of variables is in* $\mathcal{O}(n(m\log(nm) + 2^{|I|} \cdot (|O| + n)) + kn^2)$*, where* $n = |\mathcal{T}'|, m = |Q|$ *and* k *the number of allowed operations.*

4.2 Generating Explanations

We describe now how we can solve the explanation synthesis problem for a given LTL-formula φ over $AP = I \cup O$, a 2^O-labeled 2^I-transition system $\mathcal{T}$ and a minimal repair ξ. The minimal repair ξ can be obtained from Algorithm 1. The construction of the explanation follows the idea from the proof of Theorem 2 and is shown in Algorithm 2. An explanation ex is a sequence of transformations ξ_i and counterexamples σ_i such that every transformation ξ_i can be minimally justified by σ_i. A counterexample σ for the current transition system $\mathcal{T}_{old}$ is obtained by Bounded Model Checking (BMC) [2] and is a justification for ξ as ξ is a minimal repair. BMC checks if there is a counterexample of a given bound n that satisfies the negated formula $\neg\varphi$ and is contained in $\mathcal{L}(\mathcal{T})$. The constraint system $\phi_{\mathcal{T}} \wedge \phi_{loop} \wedge [\![\neg\varphi]\!]$ is composed of three components. $\phi_{\mathcal{T}}$ encodes the transition system $\mathcal{T}$, where each state $t \in T$ is represented as a boolean vector. ϕ_{loop} ensures the existence of exactly one loop of the counterexample, and the fixpoint formula $[\![\neg\varphi]\!]$ ensures that the counterexample satisfies the LTL formula. To obtain a minimal justification, we need to ensure that there is no transformation $\xi' \subset \xi$ such that σ justifies ξ'. As long as there is an operation Δ such that $\sigma \notin \mathcal{L}(apply^*(\mathcal{T}_{old}, \xi' \backslash \{\Delta\}))$, σ is not a minimal justification for ξ'. Otherwise σ minimally justifies ξ' and (ξ', σ) can be appended to the explanation. The algorithm terminates and returns an explanation if ξ is empty, i.e. every operation is justified. The presented algorithm solves the BMC-problem at most $|\xi|$-times and the number of checks if a counterexample is contained in the language of a transition system is bounded by $|\xi|^2$. The correctness of Algorithm 2 is shown in Theorem 2.

5 Experimental Results

We compare our explainable synthesis approach with **BoSy** [5], a traditional synthesis tool, on several benchmarks. After describing the different benchmark families and technical details, we explain the observable results.

5.1 Benchmark Families

The benchmarks families for arbiter, AMBA and load balancer specifications are standard specifications of SYNTCOMP [11]. For the scaling benchmarks only a constant number of operations is needed. The remaining benchmarks synthesize support for different layers of the OSI communication network.

- **Arbiter:** An *arbiter* is a control unit that manages a shared resource. Arb_n specifies a system to eventually grant every request for each of the n clients and includes mutual exclusion, i.e. at most one grant is given at any time. $ArbFull_n$ additionally does not allow spurious grants, i.e. there is only given a grant for client i if there is an open request of client i.
- **AMBA:** The *ARM Advanced Microcontroller Bus Architecture* (AMBA) is an arbiter allowing additional features like locking the bus for several steps. The specification $AMBAEnc_n$ is used to synthesize the encode component of the decomposed AMBA arbiter with n clients that need to be controlled. $AMBAArb_n$ specifies the arbiter component of a decomposed AMBA arbiter with an n-ary bus, and $AMBALock_n$ specifies the lock component.
- **Load Balancer:** A *load balancer* distributes a number of jobs to a fixed number of server. $LoadFull_n$ specifies a load balancer with n clients.

Table 1: Benchmarking results of BoSy and our explainable synthesis tool

Initial		Extended		Size	Operations	Number	Time in sec.	Time in sec.
Ben.	Size	Ben.	Size	Aut.	chL/rdT	Just.	BoSy	Explainable
Arb_2	2	Arb_4	4	5	0/3	3	0.348	1.356
Arb_4	4	Arb_5	5	6	0/2	2	2.748	12.208
Arb_4	4	Arb_6	6	7	0/3	3	33.64	139.088
Arb_4	4	Arb_8	-	9	-	-	timeout	timeout
Arb_2	2	$ArbFull_2$	4	6	3/7	10	0.108	0.352
$ArbFull_2$	4	$ArbFull_3$	8	8	1/18	19	26.14	288.168
$AMBAEnc_2$	2	$AMBAEnc_4$	4	6	1/11	12	0.26	7.16
$AMBAEnc_4$	4	$AMBAEnc_6$	6	10	1/21	22	5.76	973.17
$AMBAArb_2$	-	$AMBAArb_4$	-	17	-	-	timeout	timeout
$AMBAArb_4$	-	$AMBAArb_6$	-	23	-	-	timeout	timeout
$AMBALock_2$	-	$AMBALock_4$	-	5	-	-	timeout	timeout
$AMBALock_4$	-	$AMBALock_6$	-	5	-	-	timeout	timeout
$LoadFull_2$	3	$LoadFull_3$	6	21	1/10	10	6.50	49.67
$Loadfull_3$	6	$LoadFull_4$	-	25	-	-	timeout	timeout
$BitStuffer_2$	5	$BitStuffer_3$	7	7	2/7	9	0.08	1.02
$BitStuffer_3$	7	$BitStuffer_4$	9	9	1/6	7	0.21	3.97
$ABPRec_1$	2	$ABPRec_2$	4	9	2/5	7	0.11	1.52
$ABPRec_2$	4	$ABPRec_3$	8	17	4/9	13	2.87	326.98
$ABPTran_1$	2	$ABPTran_2$	4	31	1/5	5	0.76	76.93
$ABPTran_2$	4	$ABPTran_3$	-	91	-	-	timeout	timeout
TCP_1	2	TCP_2	4	6	1/4	5	0.05	0.19
TCP_2	4	TCP_3	8	8	3/14	17	0.58	14.47
$Scaling_4$	4	$Scaling'_4$	4	4	4/0	4	0.02	0.10
$Scaling_5$	5	$Scaling'_5$	5	5	4/0	4	0.03	0.22
$Scaling_6$	6	$Scaling'_6$	6	6	4/0	4	0.04	0.51
$Scaling_8$	8	$Scaling'_8$	8	8	4/0	4	0.12	2.54
$Scaling_{12}$	12	$Scaling'_{12}$	12	12	4/0	4	34.02	167.03

- **Bit Stuffer:** *Bit stuffing* is a part of the physical layer of the OSI communication network which is responsible for the transmission of bits. Bit stuffing inserts non-information bits into a bit data stream whenever a defined pattern is recognized. $BitStuffer_n$ specifies a system to signal every recurrence of a pattern with length n.
- **ABP:** The *alternating bit protocol* (ABP) is a standard protocol of the data link layer of the OSI communication network which transmits packets. Basically, in the ABP, the current bits signals which packet has to be transmitted or received. $ABPRec_n$ specifies the ABP with n bits for the receiver and $ABPTran_n$ for the transmitter.
- **TCP-Handshake:** A *transmission control protcol* (TCP) supports the transport layer of the OSI communication network which is responsible for the end-to-end delivery of messages. A TCP-handshake starts a secure connection between a client and a server. TCP_n implements a TCP-handshake where n clients can continuously connect with the server.
- **Scaling:** The $Scaling_n$ benchmarks specify a system of size n. To satisfy the specification $Scaling'_n$ a constant number of operations is sufficient.

5.2 Technical Details

We instantiate BoSy with an explicit encoding, a linear search strategy, an input-symbolic backend and moore semantics to match our implementation. Both tools only synthesize winning strategies for system players. We use **ltl3ba**[1] as the converter from an LTL-specification to an automaton for both tools. As both constraint systems only contain existential quantifiers, **CryptoMiniSat**[23] is used as the SAT-solver. The solution bound is the minimal bound that is given as input and the initial transition system is synthesized using BoSy, at first. The benchmarks results were obtained on a single quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz) with 32 GB RAM, running a GNU/Linux system. A timeout of 2 hours is used.

5.3 Observations

The benchmark results are shown in Table 1. For each benchmark, the table contains two specifications, an initial and an extended one. For example, the initial specification for the first benchmark Arb_2 specifies a two-client arbiter and the extended one Arb_4 specifies a four-client arbiter. Additionally, the table records the minimal solution bound for each of the specifications. Our explainable synthesis protoype starts by synthesizing a system for the initial specification and then synthesizes a minimal repair and an explanation for the extended one. If the minimal repair has to access additional states, that are initially unreachable, our prototype initializes them with a self loop for all input assignments where no output holds. The traditional synthesis tool BoSy only synthesizes a solution for the extended specification. The size of the universal co-Büchi automaton, representing the extended specification is reported. For the explainable synthesis, the applied operations of the minimal repair and the number of justifications is given. For both tools, the runtime is reported in seconds.

The benchmark results reveal that our explainable synthesis approach is able to solve the same benchmarks like BoSy. In all cases, except two, we are able to synthesize explanations where every operation can be single justified. The applied operations show that there are only a small number of changed state labelings, primarily for reaching additional states. Since only minimal initial systems are synthesized, the outputs in the given structure are already fixed. Redirecting transitions repairs the system more efficient. In general, the evaluation reveals that the runtime for the explainable synthesis process takes a multiple of BoSy with respect to the number of applied operations. Thus, the constraint-based synthesis for minimal repairs is not optimal if a small number of operations is sufficient since the repair synthesis problem is polynomial in the size of the system as proven in Lemma 1. To improve the runtime and to solve more instances, many optimizations, used in existing synthesis tools, can be implemented. These extensions include different encodings such as QBF or DQBF or synthesizing strategies for the environment or a mealy semantics.

6 Conclusion

In this paper, we have developed an explainable synthesis process for reactive systems. For a set of specification, expressed in LTL, the algorithm incrementally builds an implementation by repairing intermediate implementations to satisfy the currently considered specification. In each step, an explanation is derived to justify the taken changes to repair an implementation. We have shown that the decision problem of finding a repair for a fixed number of transformations is polynomial in the size of the system and exponential in the number of operations. By extending the constraint system of bounded synthesis, we can synthesize minimal repairs where the resulting system is size-optimal. We have presented an algorithm that constructs explanations by using Bounded Model Checking to obtain counterexample traces. The evaluation of our prototype showed that explainable synthesis, while more expensive, can still solve the same benchmarks as a standard synthesis tool. In future work, we plan to develop this approach into a comprehensive tool that provides rich visual feedback to the user. Additionally, we plan to investigate further types of explanations, including quantitive and symbolic explanations.

References

1. Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. Ltl to büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
2. Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.
3. Borzoo Bonakdarpour and Bernd Finkbeiner. Program repair for hyperproperties. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 423–441, Cham, 2019. Springer International Publishing.
4. Peter Faymonville, Bernd Finkbeiner, Markus Rabe, and Leander Tentrup. Encodings of bounded synthesis. pages 354–370, 03 2017.
5. Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In *Proceedings of CAV*, volume 10427 of *LNCS*, pages 325–332. Springer, 2017.
6. Bernd Finkbeiner and Swen Jacobs. Lazy synthesis. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.
7. Bernd Finkbeiner and Felix Klein. Bounded cycle synthesis. volume 9779 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2016.
8. Bernd Finkbeiner and Felix Klein. Reactive synthesis: Towards output-sensitive algorithms. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 25–43. IOS Press, 2017.
9. Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.
10. Bernd Finkbeiner and Hazem Torfah. Synthesizing skeletons for reactive systems. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for*

Verification and Analysis - 14th International Symposium, ATVA 2016 Proceedings, Lecture Notes in Computer Science, 2016.

11. Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Perez, Jean-Francois Raskin, Ocan Sankur, and Leander Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants and results. In *SYNT 2017*, volume 260 of *EPTCS*, pages 116–143, 2017.
12. Swen Jacobs, Roderick Bloem, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Philipp J. Meyer, Thibaud Michaud, Mouhammad Sakr, Salomon Sickert, Leander Tentrup, and Adam Walker. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. *CoRR*, abs/1904.07736, 2019.
13. Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV Proceedings*. Springer, 2005.
14. Hadas Kress-Gazit and Hazem Torfah. The challenges in specifying and explaining synthesized implementations of reactive systems. In *Proceedings CREST@ETAPS*, EPTCS, pages 50–64, 2018.
15. Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.
16. P. Nilsson and N. Ozay. Incremental synthesis of switching protocols via abstraction refinement. In *53rd IEEE Conference on Decision and Control*, pages 6246–6253, 2014.
17. Hans-Jörg Peter and Robert Mattmüller. Component-based abstraction refinement for timed controller synthesis. In Theodore Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), December 1 - December 4, 2009, Washington, D.C., USA*, pages 364–374, Los Alamitos, CA, USA, December 2009. IEEE Computer Society.
18. A. Pnueli and Roni Rosner. On the synthesis of a reactive module. *Automata Languages and Programming*, 372:179–190, 01 1989.
19. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS 77, page 4657, USA, 1977. IEEE Computer Society.
20. G. Reissig, A. Weber, and M. Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4):1781–1796, 2017.
21. Leonid Ryzhyk and Adam Walker. Developing a practical reactive synthesis tool: Experience and lessons learned. In Ruzica Piskac and Rayna Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, volume 229 of *EPTCS*, pages 84–99, 2016.
22. A. Sistla and Edmund Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32:733–749, 07 1985.
23. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 244–257. Springer Berlin Heidelberg, 2009.