“*How Does my Circuit Work?*” Local Explanations for the Behavior of Sequential Circuits

Amirmohammad Nazari, Matin Amini, and Mukund Raghothaman

University of Southern California
{nazaria, matinami, raghotha}@usc.edu

Abstract. There has been a massive amount of work in algorithms to verify and synthesize systems from temporal specifications. In contrast, there has been less work devoted to the problem of helping engineers to understand how and why their systems exhibit certain behaviors. Such understanding is important for them to debug, validate, and modify their implementations in response to changing needs. In this paper, we present one possible formalization of this problem as the task of recovering specifications that locally describe the behavior of individual parts of the circuit, given LTL specifications that globally describe the behavior of the entire circuit. We study the theoretical properties of these *temporal subspecifications*, and show that they are not always expressible in LTL, but can always be described by ω-regular languages. We show that our algorithm can efficiently generate compact subspecifications when applied to benchmarks from the SYNTCOMP 2023 competition. Finally, through a user study, we show that subspecifications improve the accuracy of engineers by a factor of 17 when answering questions about these circuits.

1 Introduction

This paper is about helping engineers to understand and debug sequential circuits. Despite a massive amount of research on verification [6,29,2,11], synthesis [28,17,25] and repair [19,12], there has been comparatively less attention given to the task of aiding engineers in debugging, validating and optimizing their designs. There is admittedly some work on helping engineers automatically derive temporal specifications from their code [23,27], translating these specifications into natural language descriptions [3], and automatically deriving LTL specifications from natural language text [9,10]. However, these are focused more on the tasks of specification engineering, rather than on helping engineers develop and validate beliefs about different parts of their system.

Indeed, as we will see in our user study, participants struggle to explain the operation of even relatively simple sequential circuits. Although it is easy to obtain execution traces of these systems, design, modification and debugging fundamentally involves reasoning about counterfactual (“*what if?*”) behaviors of different parts of the system.

While studying a similar problem in the context of SyGuS program synthesizers, Nazari et al. [26] proposed the concept of *subspecifications*—i.e., automatically derived specifications of individual subexpressions—as a way of locally explaining what different parts of a loop-free program *should do*. Our present paper may be alternatively viewed as asking whether a similar notion of subspecifications can be developed in the context of reactive systems.

In our setting, the subspecification corresponds to the set of valid signals that can be produced by individual latches so that the rest of the system satisfies the desired global specification. This therefore provides a way for engineers to characterize the space of valid behaviors of different components, while abstracting away surrounding parts of the system.

The first question that arises when extending the idea of subspecifications to sequential circuits and temporal specifications involves asking what an appropriate language for expressing these temporal subspecs would even be. As we will see in Section 4, it is easy to design circuits where the subspecs for individual components are inexpressible in LTL, even though the global circuit behavior was specified as an LTL formula. We will then show that these subspecs are always ω-regular and may be conveniently expressed as Buchi automata.

We will report on a user study showing that subspecifications massively help users in a range of debugging and validation tasks (improving their response accuracy by 17×). Finally, we will present an experimental evaluation in which we observe that our algorithm can rapidly derive simple subspecs.

2 Formally Defining Subspecifications

We adapt the following example from the website of the `ltlsynt` tool,[1] distributed as part of the Spot framework [8]. Say an engineer wishes to synthesize a circuit that accepts two Boolean-valued signals i and j as input, and produces an output signal x such that x eventually drops from `true` to `false` iff i and j are both `true` in the initial time step:

$$(i \wedge j) \iff \mathsf{F}(x \wedge \mathsf{X}\,\neg x). \tag{1}$$

In response, `ltlsynt` synthesizes the controller shown in Figure 1, both as a state machine and the corresponding and-inverter graph [1,16].

At this point, say the engineer wishes to understand the purpose of the latch labelled b, with the goal of either optimizing, debugging, or otherwise modifying the circuit. As a first attempt, they might draw a state machine describing the value produced by the latch in response to the history of inputs i and j. See Figure 2a. We observe (unsurprisingly) that this state machine is remarkably similar to the original controller from Figure 1a.

Note however, that although this machine accurately describes the output of latch b, there is a class of questions that it leaves unresolved: For example, if the engineer wishes to optimize or modify this part of the circuit, they would be interested in not just its *current* behavior, but all *possible* behaviors of the latch. They would similarly interested in possible legal alternative behaviors if changing

[1] https://spot.lre.epita.fr/ltlsynt.html

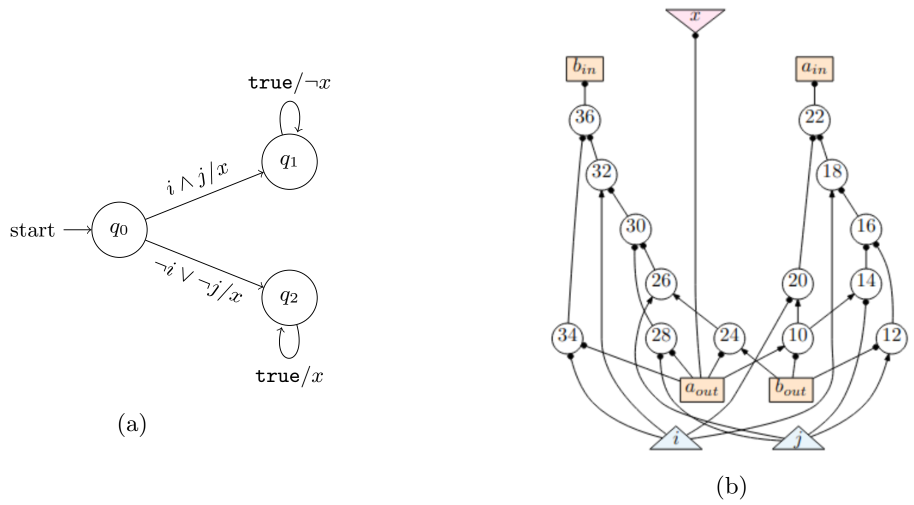


Fig. 1: The controller generated by Spot (`ltlsynt`) for the specification in Equation 1 (1a) and its corresponding and-inverter graph (1b). The numbered circles represent AND gates, and the smaller shaded circles represent inverters. Rectangles represent the input and output sides of latches, and occur in pairs, indicated as X_{in} and X_{out} respectively. Each latch represents a unit time delay and is initialized to `false`. The engineer wishes to know the purpose of the latch b.

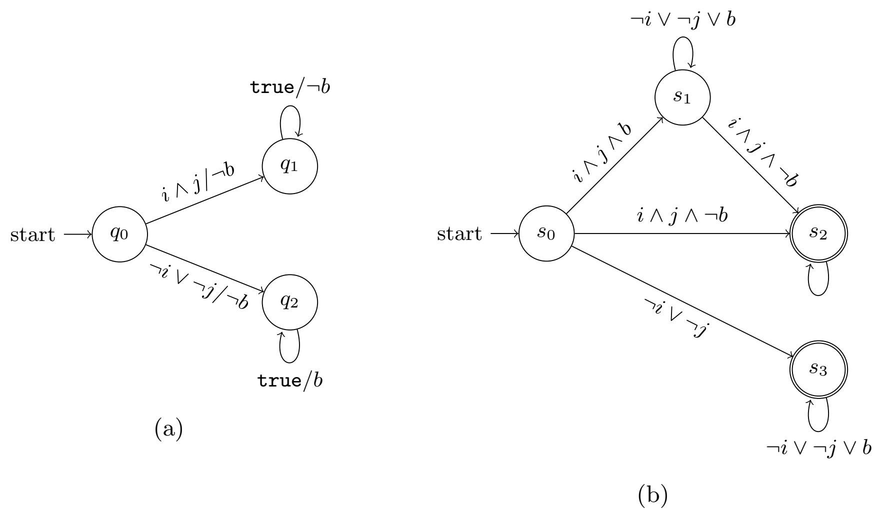


Fig. 2: (2a) Behavior of the latch b from the circuit in Figure 1b. Observe its similarity to the original controller from Figure 1a. (2b) Description of all possible legal behaviors of the latch—i.e., its *subspecification*—so that the entire circuit satisfies Equation 1. Notice that this machine reveals additional possibilities that are not exhibited by the current implementation.

requirements or faults elsewhere in the circuit need to be mitigated by repairs in this part. Therefore, instead of inquiring about the current behavior of latch b, we are interested in the question: What values *should* the latch b produce, so that the rest of the circuit satisfies the specification in Equation 1?

We can show that b can be replaced by any signal that satisfies the property:

$$(i \wedge j) \iff \mathsf{F}(i \wedge j \wedge \neg b). \tag{2}$$

This formula may be equivalently viewed as the Buchi automaton in Figure 2b. Because the future values of i and j are unconstrained, it follows that whenever i and j are `true` in the first time step, b must also produce the initial value `false`. Its output for the rest of time is unconstrained. Alternatively, if either i or j were initially untrue, then it is obligated to obey the constraint $\mathsf{G}(i \wedge j \implies b)$. Our central goal with the idea of subspecifications, that we will now formalize, is to provide a uniform answer to counterfactual questions of this kind.

Background: Sequential circuits, linear temporal logic (LTL), and Buchi automata. Sequential circuits will form our objects of study in this paper. In brief, a sequential circuit $C = (I, O, L, \boldsymbol{f})$ is specified by finite sets of Boolean-valued input and output signals, $I = \{i_1, i_2, \dots, i_m\}$, $O = \{x_1, x_2, \dots, x_n\}$, a finite set of latches, $L = \{a_1, a_2, \dots, a_l\}$, and associated update functions, $f_v : \mathrm{Bool}^{m+l} \to \mathrm{Bool}$, for each $v \in O \cup L$.

The inputs supplied to the circuit may be modeled as an infinite sequence of valuations, $\boldsymbol{\sigma} = \sigma_1, \sigma_2, \dots$, of each of the input signals I. The circuit responds by iteratively computing the values of its latches, $\boldsymbol{\rho}$, and output signals, $\boldsymbol{\tau}$, as follows:

$$\begin{aligned} \rho_0(a) &= \texttt{false}, \\ \rho_{i+1}(a) &= f_a(\sigma_i, \rho_i), \text{ and} \\ \tau_i(x) &= f_x(\sigma_i, \rho_i), \end{aligned}$$

for $a \in L$, $x \in O$, and $i \in \mathbb{N}$. For example, the circuit in Figure 1b may be represented using the set of update functions:

$$\left.\begin{aligned} a_{n+1} &= (\neg i_n \wedge a_n \wedge \neg b_n) \vee (i_n \wedge \neg j_n \wedge a_n \wedge \neg b_n) \vee (i_n \wedge j_n \wedge \neg b_n), \\ b_{n+1} &= (\neg i_n \wedge \neg a_n) \vee (i_n \wedge \neg j_n \wedge \neg a_n) \vee (i_n \wedge j_n \wedge \neg a_n \wedge b_n), \text{ and} \\ x_n &= \neg a_n. \end{aligned}\right\} \tag{3}$$

We specify properties of these sequential circuits using formulas in LTL. Recall that an LTL formula ϕ is a production of the grammar:

$$\phi ::= \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\,\phi \mid \mathsf{F}\,\phi \mid \mathsf{G}\,\phi \mid \phi_1\,\mathsf{U}\,\phi_2,$$

where $v \in I \cup O \cup L$. The interpretation of these formulas over infinite traces is standard. We refer the reader to Clarke et al.'s textbook on model checking [5].

One may alternatively specify properties of infinite signals using Buchi automata. A Buchi automaton is a structure $M = (Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet (in our case, most commonly Bool^{m+n}), $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. The machine in Figure 2b is an example. We say that the machine accepts an ω-string $w = w_1, w_2, \ldots$ if there exists a corresponding run $q_0 \rightarrow^{w_1} q_1 \rightarrow^{w_2} q_2 \rightarrow^{w_3} \cdots$ in which some state $q_f \in F$ occurs infinitely often. We refer to the language that M accepts as $L(M) \subseteq \Sigma^\omega$. Once again, we refer the reader to [5].

Sequential subspecifications. Let $C = (I, O, L, \boldsymbol{f})$ be a sequential circuit, and let ϕ be an LTL specification with free variables from $I \cup O$. Let $b \in L$ be a latch that the engineer wishes to investigate. We can now use C to construct a new circuit:

$$C|_b = (I \cup \{b\}, O, L \setminus \{b\}, \boldsymbol{f} \setminus \{(b, f_b)\}).$$

Informally, this amounts to promoting b, currently represented by a latch, to the status of a new input, while leaving the rest of the circuit unchanged. We say that an LTL formula ψ is a *subspecification* of the latch b with respect to the global specification ϕ if for each sequence of inputs $\tilde{\boldsymbol{\sigma}}$ supplied to $C|_b$, we have:

$$\tilde{\boldsymbol{\sigma}} \models \psi \iff (\boldsymbol{\sigma}, \boldsymbol{\tau}) \models \phi, \tag{4}$$

where $\boldsymbol{\tau}$ is the corresponding sequence of outputs produced by $C|_b$, and $\boldsymbol{\sigma}$ is the projection of $\tilde{\boldsymbol{\sigma}}$ obtained by eliminating the values of b.

Note 1.
1. *Informally*: The subspec ψ specifies all possible alternative values that could have been produced by b, so that the rest of the circuit $C|_b$ still satisfies ϕ.
2. Also note that we do not require the initial circuit C to itself satisfy ϕ. This flexibility is useful when engineers are concerned with problems of debugging and repair, as we will see in the example in Section 3.1.
3. By considering sequential circuits and temporal properties rather than stateless expressions, the definition of subspecs in Equation 4 is a strict generalization of the idea initially developed by [26].

Ideally, one would like to express the specification and subspec in the same language. However, it is easy to construct circuits where the subspec is inexpressible as an LTL formula. We will see an example in Section 4.1. We will therefore primarily be interested in situations where the subspec is represented as a Buchi automaton. We say that a Buchi automaton M with alphabet $\Sigma = \text{Bool}^{|I \cup \{b\}|}$ is the subspecification of b with respect to ϕ if:

$$\tilde{\boldsymbol{\sigma}} \in L(M) \iff (\boldsymbol{\sigma}, \boldsymbol{\tau}) \models \phi,$$

where (as before) $\boldsymbol{\tau}$ is the sequence of outputs produced by $C|_b$, and $\boldsymbol{\sigma}$ is the sequence of valuations obtained from $\tilde{\boldsymbol{\sigma}}$ by projecting out the values of the "*real*" inputs, $i \in I$.

Paper outline. In Section 3, we will present two additional examples illustrating the utility of subspecs for validation and debugging. Then, in Section 4, we will present an algorithm to automatically derive these subspecifications. Finally, in Sections 5 and 6, we will focus on empirically validating their usefulness and our effectiveness in producing compact subspecs.

3 Example Applications

Our examples in this section will be drawn from the Reactive Synthesis Competition, SYNTCOMP 2023 [17]. Like for the example in Section 2, automatically synthesized controllers provide a convenient source of specifications and implementations that are tricky to comprehend.

3.1 Debugging Circuits

We consider the example of `lilydemo12.tlsf` [18]. The original goal was to synthesize a controller that maps a pair of input signals, i, j, to a pair of output signals, x and y, such that:

$$\mathsf{G}\,\neg x \lor \mathsf{G}(i \implies \mathsf{F}\,y) \lor \mathsf{G}(j \implies \mathsf{F}\,x).$$

In response, Spot produces a controller implemented using the following two-latch circuit:

$$\left.\begin{aligned} a_{n+1} &= \neg j_n \land \neg a_n \land \neg b_n, \\ b_{n+1} &= (j_n \land \neg a_n) \lor (\neg a_n \land b_n) \lor (a_n \land \neg b_n), \\ x_n &= (\neg j_n \land a_n \land \neg b_n) \lor (\neg a_n \land b_n), \text{ and} \\ y_n &= (i_n \land \neg a_n) \lor (i_n \land \neg b_n). \end{aligned}\right\} \tag{5}$$

Assume, for the sake of example, that there was a transcription error, and the outputs were incorrectly calculated as follows:

$$x'_n = \underbrace{\neg x_n} \quad \text{and} \quad y'_n = (\underbrace{\neg i_n} \land \neg a_n) \lor (\underbrace{\neg i_n} \land \neg b_n). \tag{6}$$

In other words, two mistakes were made: the output x was incorrectly negated, and the input i was incorrectly negated while being used to compute y. Note that these mistakes correspond to two bit flips in the AIGER-encoded circuit.

At this point, the engineer might wish to explore ways of repairing the system. Among other questions, they might wonder whether its functionality can be restored by changing the values produced by latch b. Although one might draw the state machine corresponding to the current computation of b in a manner similar to what we did in Figure 2a—see Figure 3a—we note that this is useless, because we are uninterested in what b *currently does*, and instead interested in what the latch *should now be doing*.

As part of our user study in Section 5, we asked a group of students to suggest possible ways of repairing the circuit by modifying the behavior of b. Notably,

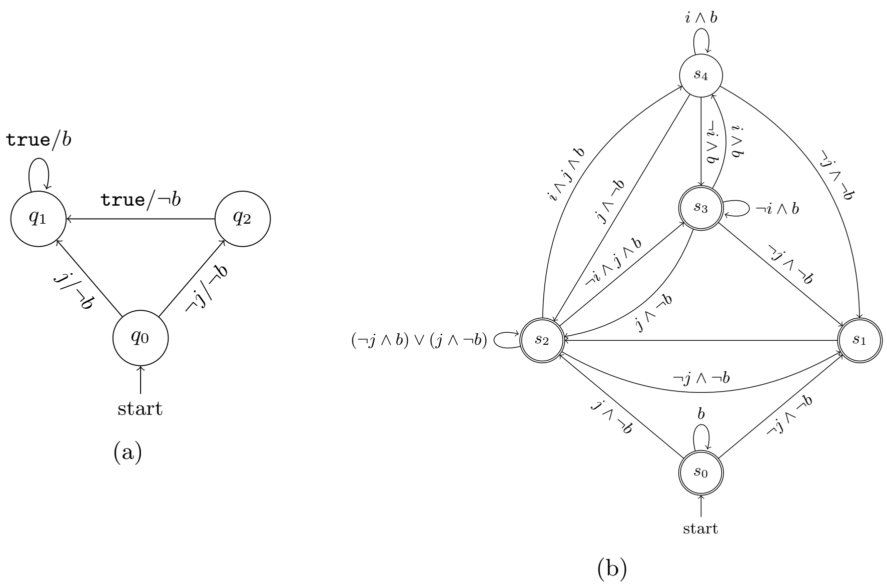


Fig. 3: (3a) The controller for the latch b in the circuit of Equation 5. (3b) A description of all possible behaviors of the latch b so that the rest of the faulty circuit from Equation 6 nevertheless satisfies the specification in Equation 3.1.

without additional assistance, only one participant of nine was able to solve the task, and required approximately 8 minutes to identify a fix.

Alternatively, using our subspecification derivation algorithm from Section 4.2, one discovers that the latch can be replaced with any component all of whose behaviors are accepted by the Buchi automaton shown in Figure 3b. Observe now that the erroneous circuit would satisfy the specification if b were to simply be replaced with a signal that produces the constant value `true`. By replacing value of b in Equation 5 with this new signal, $b'_n = \texttt{true}$, one observes that it results in the new sequence of output values $x''_n = \texttt{false}$ and $y''_n = \neg i_n$. This repaired implementation would therefore satisfy the specification by fulfilling its leftmost term, $\mathsf{G}\,\neg x$.

Eight of the 9 participants in the intervention group in our user study suggested this method of fixing the system. The remaining participant identified the following (only slightly more complicated) fix. They observed that state q_3 was the only non-accepting state in the subspec automaton in Figure 3b and pointed out that all transitions leading to this state would be disabled if $b''_n = \neg i_n$. Plugging in this fix into the faulty update expressions in Equation 6 and simplifying reveals that, in this case, $y''_n = i_n$, so that the repaired implementation works by fulfilling the second term in the specification, $\mathsf{G}(i \implies \mathsf{F}\,y)$.

3.2 Validating Sequences

We now look at `example72.tlsf` from the SYNTCOMP 2023 benchmark suite. Here, we are interested in a two-input (i, j) two-output (x, y) controller such that:

$$\mathsf{G}(\neg x \vee \neg y) \wedge \mathsf{G}(i \implies x \vee \mathsf{X}\, x) \wedge \mathsf{G}(j \implies y \vee \mathsf{X}\, y). \tag{7}$$

The two-latch circuit in question is specified by the following update expressions:

$$\left.\begin{aligned} a_{n+1} &= (i_n \wedge j_n \wedge \neg a_n) \vee (\neg i_n \wedge j_n \wedge \neg a_n \wedge b_n), \\ b_{n+1} &= a_n \wedge \neg b_n \wedge i_n, \\ x_n &= (\neg i_n \wedge j_n \wedge \neg a_n \wedge b_n) \vee (\neg i_n \wedge \neg j_n \wedge \neg a_n) \vee (i_n \wedge \neg a_n), \text{ and} \\ y_n &= (i_n \wedge \neg b_n \wedge a_n) \vee (\neg i_n \wedge j_n \wedge \neg b_n) \vee (\neg i_n \wedge \neg j_n \wedge a_n \wedge \neg b_n). \end{aligned}\right\} \tag{8}$$

As part of their validation process before incorporating this system in their designs, the engineer might simulate the controller under a variety of test inputs. We show an example trace in Figure 4a. In this situation, they observe that both latches a and b uniformly remain at `false`. They ask whether the circuit still works if b is forced to be constantly `true`. Questions like this might conceivably also arise when they are modifying the circuit and running test cases.

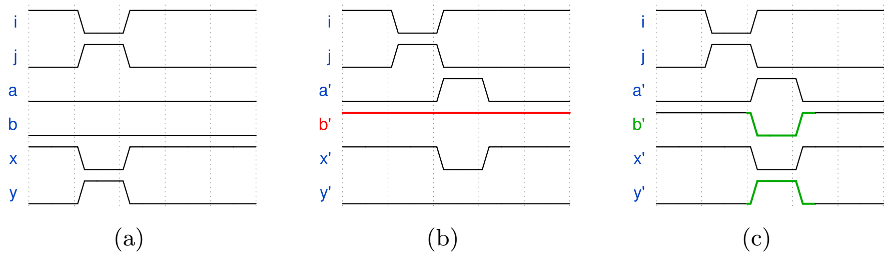


Fig. 4: (4a) Original behavior of the system resulting from signal i turning off and j turning on for one clock cycle in the second time step. The engineer wonders why the circuit would not work if b were to produce the constant value `true`. (4b) The values produced by a', x' and y' in this counterfactual scenario. This execution trace fails the specification because y' never goes high in response to the impulse on j. (4c) Analyzing the subspec for b reveals that pushing it to `false` in the third time step would restore global correctness.

Modifying the behavior of latch b in this manner would affect the computation both of the remaining latch a and of the outputs x, y, resulting in the alternative execution trace shown in Figure 4b. It can be seen that this trace does not satisfy the specification in Equation 7.

We now observe that subspecifications can provide greater insight into the causes for this failure. We show the automatically calculated subspec automaton in Figure 5. It turns out that the new trace for b causes this machine to pass through the sequence of states $q_0 \to q_0 \to q_1$. At this point, the run would terminate because both outgoing transitions from q_1 are disabled. This analysis allows us to localize the fault within this alternative trace to the third time step. Pushing b to `false` for one clock cycle at this point would cause the resulting trace to once again satisfy the global specification. See Figure 4c.

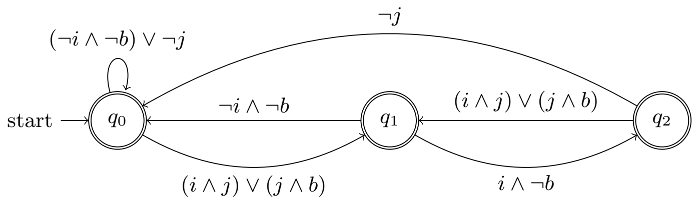

$(\neg i \wedge \neg b) \vee \neg j$
$\neg j$
start
q_0
$\neg i \wedge \neg b$
q_1
$(i \wedge j) \vee (j \wedge b)$
q_2
$(i \wedge j) \vee (j \wedge b)$
$i \wedge \neg b$

Fig. 5: A description of all possible legal behaviors of the latch b so that the circuit from Equation 8 satisfies Equation 7.

4 Expressibility and Automatic Derivation of Subspecs

The principal contribution of this paper over [26] is in extending the idea of subspecifications from the setting of stateless, loop-free expressions to the more general setting of sequential circuits. Naturally, we need to reconsider questions related to expressiveness and develop new algorithms to automatically derive these subspecs (if they exist). Unfortunately, it is easy to show that even if the global specification is provided as an LTL formula, the subspecification of a latch need not itself always be expressible using LTL. See Theorem 1. On the other hand, in Section 4.2 we show that the subspec is always expressible as a Buchi automaton. Our proof of this second result also provides an algorithm to automatically derive these subspecifications.

4.1 Inexpressibility of Subspecifications as LTL Formulas

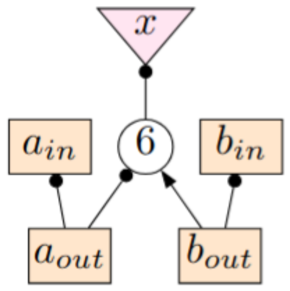
We start with the simple two-latch circuit shown in Figure 6a. Each latch flips its value from the previous time step:

$$a_{n+1} = \neg a_n \quad \text{and} \quad b_{n+1} = \neg b_n. \tag{9}$$

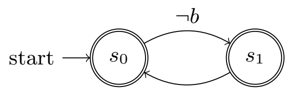
Recall that both latches are initialized as $a_0 = b_0 =$ `false`. The circuit calculates its single output bit as follows:

$$x_n = a_n \vee \neg b_n.$$

Naturally, this circuit always produces the output `true`, thereby satisfying $\mathsf{G}\, x$.



(a)



(b)

Fig. 6: (6a) Representation of the circuit from Equation 9 as an and-inverter graph. (6b) Subspec of the latch b with respect to the global specification, $\mathsf{G}\,x$. Observe that the output value is unconstrained in odd-indexed time steps.

It is easy to see that for this global property, $\mathsf{G}\,x$ to hold, the latch b must produce the value `false` in even time steps, $t = 0, 2, 4, \ldots$. Its value in odd time steps, $t = 1, 3, 5, \ldots$, is unconstrained. This subspecification may be represented using the Buchi automaton in Figure 6b, but is famously inexpressible as an LTL formula [31]. It follows that:

Theorem 1. *There is no LTL formula which describes the subspecification of latch b in the circuit of Figure 6a and with respect to the global specification, $\mathsf{G}\,x$.*

4.2 Automatically Deriving Subspecs as Buchi Automata

We will now describe an algorithm to obtain the subspecification of a latch b when it is requested as a Buchi automaton. Recall that the problem is to replace the latch with a new "*magic*" input signal, and determine all possible sequences of inputs that can be supplied to this new circuit $C|_b$ so that the execution of the rest of the circuit satisfies the given global specification ϕ. For the purpose of illustration, we will continue with the example from Section 2.

As a first step, we write down an LTL formula that describes all possible executions of the circuit $C = (I, O, L, \boldsymbol{f})$:

$$\chi(C) = \bigwedge_{x \in O} \mathsf{G}(x \iff f_x(I, L)) \wedge \bigwedge_{a \in L} (\neg a \wedge \mathsf{G}(\mathsf{X}\, a \iff f_a(I, L)). \qquad (10)$$

This formula, $\chi(C)$, ranges over the free variables $I \cup O \cup L$, and functions in a manner similar to the Tseitin transform [30]:

Lemma 1. *A sequence of valuations $(\boldsymbol{\sigma}, \boldsymbol{\tau}, \boldsymbol{\rho})$ of $I \cup O \cup L$ satisfies $\chi(C)$ iff the circuit C produces the sequence of outputs $\boldsymbol{\tau}$ and latch values $\boldsymbol{\rho}$ when provided with the input sequence $\boldsymbol{\sigma}$.*

Observe that $\chi(C|_b)$ describes all possible executions of the promoted circuit $C|_b$. For example, for the circuit in Equation 3, $\chi(C|_b)$ would be:

$$\mathsf{G}(x \iff \neg a) \land \neg a \land \mathsf{G}(\mathsf{X}\, a \iff (\neg i \land a \land \neg b) \lor (i \land \neg j \land a \land \neg b) \lor (i \land j \land \neg b)).$$

We are interested in executions of $C|_b$ that also satisfy ϕ. Naturally, the formula of interest is $\chi(C|_b) \land \phi$. This formula can be readily transformed into an equivalent Buchi automaton $M_{C,b,\phi}$. Figure 11 in Appendix A shows the resulting construction when applied to our running example. The main outstanding challenge is that $\chi(C|_b) \land \phi$ (and therefore $M_{C,b,\phi}$) ranges over all variables, $I \cup O \cup L$, while the desired subspec in Figure 2b only relates the values of the inputs I and the latch b that is currently being investigated.

Our key insight is that because the circuit is deterministic, the values of the remaining latches, $a \in L \setminus \{b\}$ and outputs $x \in O$ can be uniquely determined from the history of values of $I \cup \{b\}$. Let $M_{C,b,\phi} = (Q, \Sigma, \Delta, q_0, F)$, where $\Sigma = \mathrm{Bool}^{|I \cup O \cup L|}$. We construct a projected-down automaton,

$$M^{\downarrow}_{C,b,\phi} = (Q, \Sigma^{\downarrow}, \Delta^{\downarrow}, q_0, F), \tag{11}$$

by selecting the fields in Σ corresponding to $I \cup \{b\}$, so that $\Sigma^{\downarrow} = \mathrm{Bool}^{|I \cup \{b\}|}$ and

$$\Delta^{\downarrow} = \{(q, a^{\downarrow}, q') \mid (q, a, q') \in \Delta\},$$

and where $a^{\downarrow} \in \mathrm{Bool}^{|I \cup \{b\}|}$ is the symbol obtained by eliminating the unnecessary fields of $a \in \mathrm{Bool}^{|I \cup O \cup L|}$. We establish a correspondence between the executions of $M_{C,b,\phi}$ and the executions of $M^{\downarrow}_{C,b,\phi}$:

Lemma 2. *Whenever the ω-path $\pi = q_0 \to^{a_0} q_1 \to^{a_1} q_2 \to \cdots$ is accepted by $M_{C,b,\phi}$, the corresponding path $\pi^{\downarrow} = q_0 \to^{a_0^{\downarrow}} q_1 \to^{a_1^{\downarrow}} q_2 \to \cdots$ is also accepted by $M^{\downarrow}_{C,b,\phi}$. Conversely, for every path $\pi^{\downarrow}$ accepted by $M^{\downarrow}_{C,b,\phi}$, there exists a path π accepted by $M_{C,b,\phi}$ such that $\pi^{\downarrow}$ is the projection of π.*

Proof. The forward direction is immediate. In the converse direction, recall that every transition $(q, a^{\downarrow}, q') \in \Delta^{\downarrow}$ corresponds to some transition $(q, a, q') \in \Delta$ of $M_{C,b,\phi}$. For each transition $q_i \to^{a_i^{\downarrow}} q_{i+1}$ in $\pi^{\downarrow}$, arbitrarily pick $a_i \in \Sigma$ so that $(q_i, a_i, q_{i+1}) \in \Delta$. It must be the case that $\pi = q_0 \to^{a_0} q_1 \to^{a_2} q_2 \to \cdots$ is a valid path through $M_{C,b,\phi}$. Furthermore, because $M^{\downarrow}_{C,b,\phi}$ preserves the acceptance conditions, it must be the case that π is also accepted by $M_{C,b,\phi}$, thus completing the proof.

Combining Lemmas 1 and 2, we have:

Theorem 2. *For each circuit $C = (I, O, L, \boldsymbol{f})$, specification ϕ, and latch $b \in L$, the Buchi automaton $M^{\downarrow}_{C,b,\phi}$ is a subspecification of b with respect to ϕ.*

Implementation details. (a) We use Owl [21] for the LTL-to-Buchi automaton translation, and the `autfilt` tool in Spot [8] for simplifying the resulting automata. (b) We use Spot to translate transition guards into DNF form. The minterms of this formula can be easily subject to the downward projection operation. (c) Although the definition of subspecs in Section 2 focused on latches, our implementation more generally allows for the computation of subspecs for any component in the AIGER-encoded circuit. Defining subspecs for these components is an easy generalization.

5 Empirically Measuring the Utility of Subspecifications

The first part of our evaluation consisted of a user study to determine whether subspecifications were helpful to engineers. Our goal was to answer the following research questions:

RQ1. Do subspecs help users in distinguishing valid and invalid execution traces?
RQ2. Do subspecs help users in explaining the purpose of individual components?
RQ3. Do subspecs help users in repairing faulty circuits?

5.1 Participants, Tasks, and Study Structure

Participant selection and screening process. The study was conducted after obtaining IRB approval. We recruited 18 graduate students (2 Masters and 16 Ph.D. students) from the Computer Science (CS), Electrical Engineering (EE), and Industrial and Systems Engineering (ISE) departments of two prominent American and Canadian universities. These participants had a range of specializations, including optimization algorithms, human-computer interaction, machine learning and natural language processing, software engineering, MEMS and robotics, computer networking, and theoretical CS.

We started by providing the participants with an introduction to the study, and briefly introducing them to temporal logic and the idea of subspecifications. We then administered a screening quiz with 5 questions to ensure that participants had a baseline level of understanding of these background ideas. All participants received perfect scores in the screening quiz, and were therefore included in the main study.

Tasks and study structure. The study consisted of three tasks. The first task was based on the specification-implementation pair discussed in Section 3.2. It consisted of 4 questions in which participants were asked to predict whether a presented counterfactual trace would cause the rest of the circuit to produce an output trace that satisfied the specification. We also asked participants to justify their responses. The second task built on our introductory example in Section 2, and asked participants to explain, in natural language, the constraints that specific parts of the circuit must satisfy. The last task involved the faulty system discussed in Section 3.1. We pointed participants to different parts of the circuit, and asked them to suggest fixes. We also asked them to justify their responses if implementing a repair was impossible.

The study was formulated as a repeated measures design, i.e., one in which each participant attempted at least one task with access to subspecs and at least one other task without access to subspecs. For each task, participants were randomly assigned to either the intervention or control arms, with exactly 9 participants attempting each task under each condition. The screening quiz, study materials, and (anonymized) participant responses will be included as part of our artifact.

All authors of this paper independently graded participant responses. The pairwise correlation coefficients between our grades were 0.85, 0.88 and 0.90 respectively. We present our average grades (indicating our assessment of their accuracy) in Figure 7.

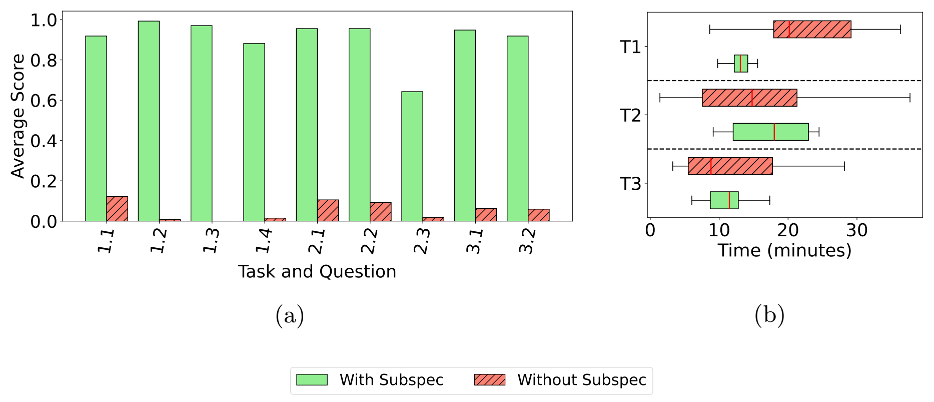


Fig. 7: Accuracy of participant responses and distribution of time needed to complete each of the study tasks. Questions 1.1–1.4 assessed the ability of participants to validate counterfactual traces, Questions 2.1–2.3 asked them to explain what different components should do, and Questions 3.1 and 3.2 assessed their ability to repair faulty circuits.

5.2 RQ1: Distinguishing Valid and Invalid Execution Traces

We draw our conclusions from Questions 1.1–1.4 of the user study. We expected each response to include both a summary Boolean-valued judgment ("*Counterfactual trace leads to valid behavior*" vs. "*Counterfactual trace leads to erroneous behavior*") and a justification.

Two trends are obvious from Figure 7a: Participants without access to subspecs were broadly unsuccessful at the task while participants with access to subspecs were significantly more effective. With and without our intervention, average participant accuracy was 94% and 3% respectively.

In the baseline-without-subspec condition, participants would have had to first mentally simulate the circuit from the provided update equations, and then determine whether the induced response satisfied the provided specification. Anecdotally, given the complexity of the update equations, most participants were unable to even simulate the circuit. In the post-study debrief, several of these participants complained about the complexity of the update equations.

Accordingly, we observed two distinct response patterns from the control group: The first subgroup opted to skip the question after spending a considerable amount of time (Notice the massively larger length of time that these participants spent on Task 1), while the second subgroup provided guesses without sound reasoning or justification. We also noticed that participants gradually became tired, so their response accuracy for Q1.1 was noticeably higher than for the remaining questions.

In contrast, the subspec eliminated the need to mentally simulate the circuit. Participants simply had to trace the behavior of the subspec automaton in response to the inputs and counterfactual latch values. The subspec therefore allowed the participants to visualize and locally reason about the execution trace, without having to engage with the rest of the circuit's components.

5.3 RQ2: Explaining the Purpose of Individual Components

Now, we draw our conclusions from observing participant responses to Questions 2.1–2.3. Specifically, these questions asked participants to describe the required behavior of individual latches so that the rest of the circuit satisfied the specification. As a point of elaboration, we asked participants for the considerations that designers must keep in mind while modifying the implementation.

Notice that, unlike the first task (which admitted a clear solution strategy even without access to subspecs,) this second task was open-ended. Here, most participants in the control group confessed to not even knowing where to start. Participants who had access to subspecs tended to approach the problem by performing a case analysis on the subspec automaton.

The components of interest in Questions 2.1 and 2.2 admitted relatively simple subspecs which only constrained their behavior in the initial time step. Consequently, all participants had a relatively higher accuracy for these two questions. In contrast, Question 2.3 was exactly the setting of latch b that we examined while initially motivating subspecs in Section 2.

In this case, subspecs made some behaviors obvious: In particular, if $i \wedge j$ was `false` in the initial time step, then b was required to satisfy $\mathsf{G}(i \wedge j \implies b)$. Similarly, if $i \wedge j \wedge \neg b$ held in the initial time step, then all requirements were lifted for the rest of time. On the other hand, if $i \wedge j$ was true in the first time step, and b also assumed the value `true`, then the subspec automaton would transition to the state s_1, which did not admit a winning strategy. It was therefore crucial for the latch b to produce the initial value `false` when initially $i \wedge j$. Four of the nine participants who had access to subspecs (and nobody in the control group) were able to completely articulate this requirement.

5.4 RQ3: Repairing Faulty Circuits

Finally, we focus on our observations of participant responses to Questions 3.1 and 3.2. Both these questions involve the specification-implementation pair from Section 3.1.

Once again, participants in the control group complained about having insufficient information to complete the task. We also noticed them becoming tired: after spending considerable effort and still being unsuccessful in Question 3.1, some of them chose to skip Question 3.2.

While designing the user study, we expected this to be the hardest of the three tasks. We were surprised that participants with access to subspecs achieved an average score of 93%, and needed the least amount of time among all three tasks. Another notable observation was that the circuit could not be repaired by modifying the component highlighted in Question 3.2. We show its subspec in Figure 8. For this question, the average score of participants in the intervention group was 92%, indicating that most of them successfully identified and justified the unrepairability of component m.

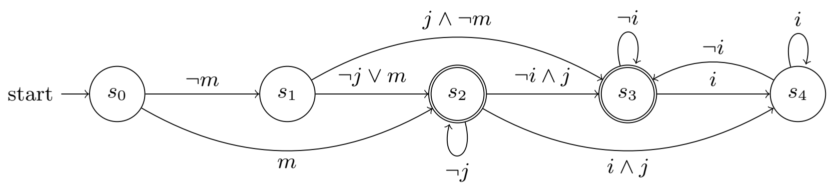


Fig. 8: Subspecification of a component m of the circuit from Equation 6 with respect to the specification in Equation 3.1.

6 Effectiveness of the Subspec Generation Procedure

Next, we measured the effectiveness of our algorithm for deriving simple subspecifications. We were interested in two research questions:

RQ4. Does the algorithm generate "*simple*" subspecs?
RQ5. How long does the procedure take to construct these subspecs?

6.1 Benchmarks and Experimental Setup

Benchmarks. We ran Strix [25], the winner of the SYNTCOMP 2023 Competition on all benchmark specifications used in the competition. We collected the generated controllers and corresponding circuit implementations. We set a 5 minute timeout on the synthesizer, within which the solver was able to successfully synthesize 635 controllers. Recall from the discussion in Section **??** that our implementation is able to calculate subspecifications for not just latches, but more generally, for any component in an AIGER-encoded circuit. We focused on controllers which had less than 100 such components, resulting in 545 specification-implementaiton pairs, and which collectively contained 13,208 components. We ran the subspecification generation tool on each of these components with a timeout of 10 minutes per run. At the end of this data collection process, we had access to subspecs for 11,453 components.

Experimental setup. We ran our experiments on a four-year old workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 21.04. We expect similar results to be obtained on most recent desktop and laptop computers.

6.2 RQ4: Effectiveness in Simplification

We measured the sizes of the generated subspecs. As such, one would expect that the size of these subspecs is dependent on the complexity of the specification or the controller being investigated. Therefore, in Figures 9a and 9b, we present the distributions of subspec size (# of states) when compared to the size of the original specification (# of AST nodes) and the size of the implementation respectively.

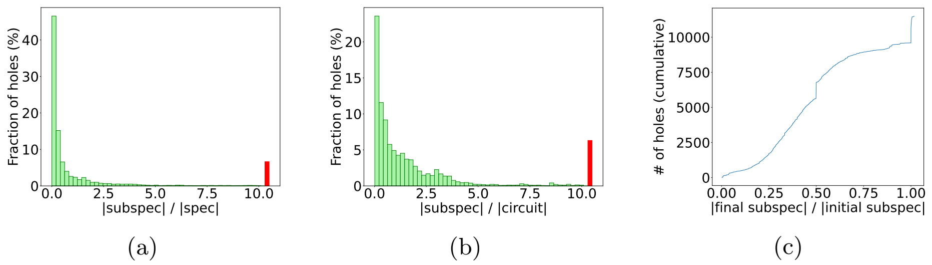


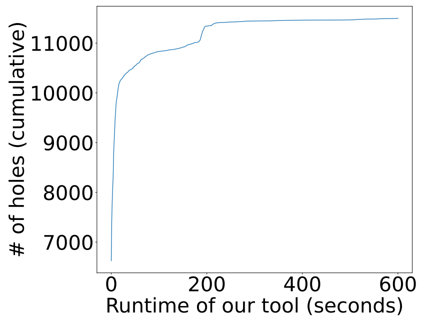
Fig. 9: (9a) Distribution of the size of the generated subspec (measured as the number of states in the subspec automaton), in comparison to the size of the original specification (measured as the number of AST nodes). The red colored bar indicates cases where the subspec size was $> 10\times$ of the specification. (9b) Distribution of subspec size when compared to the size of the controller (measured as the number of components in the AIGER implementation). (9c) Effectiveness of the subspec simplification pass described in Section **??**.

We notice that as many of 46% of the components in question admit subspecs that are less than 20% of the size of the original specification. Furthermore, in 75% of the cases, the subspec is smaller than the original specification. In only 7% of the cases is the subspec $> 10\times$ of the size of the original specification. We make similar observations when comparing the size of the subspec to the size of the circuit that surrounds the component of interest: in this case, the corresponding numbers are 22%, 55%, and 6% respectively. Of course, all these comparisons need to be interpreted with some care, because of the different units of measurement associated with the subspec and the original specification / implementation.

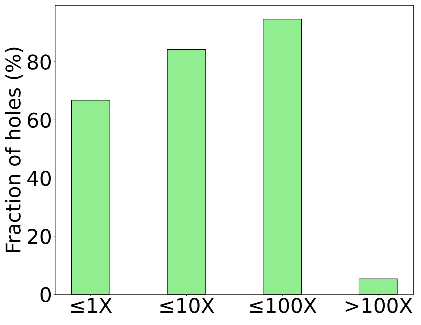
Nevertheless, we may broadly conclude that our algorithm is effective in generating simple subspecifications. We also note that we post-process the subspec initially produced by our procedure using the automata simplification routine implemented in Spot's `autfilt` tool. Figure 9c shows measurements of the effectiveness of this simplification procedure. It achieves a $\geq 50\%$ compression in 50% of all cases. This appears to be because most of the states in the originally constructed automaton, $M_{C,b,\phi}$ reason about other parts of the circuit, and are useless after the downward projection into $M^{\downarrow}_{C,b,\phi}$. We therefore believe that such post-processing passes are important in obtaining simple subspecs.

6.3 RQ5: Time Needed to Derive Subspecifications

We present measurements of the running time of the subspec generation procedure in Figures 10a and 10b: These figures respectively describe the absolute running time and a comparison to the time needed to synthesize the original controller.



(a)



(b)

Fig. 10: Cactus plot of the subspec derivation time (10a) and comparison to the time needed for originally synthesizing the circuit (10b).

Note that 57% of cases require less than a second for subspec generation, and we are faster than the original synthesis run in 66% of cases. Only 5% of the cases require long periods of waiting. Our long-term goal is that engineers consult subspecs in an interactive manner while designing their systems. The current performance of the algorithm seems adequate for this purpose.

7 Related Work

Verification and synthesis of reactive systems. Automatic verification and synthesis are foundational and widely studied problems [4]. Many algorithms and tools have been proposed [25,11], and annual competitions are conducted to identify and promote advances [2,17]. By identifying bugs and issuing certificates of correctness, verification tools greatly help in designing reliable systems. Indeed, one of the important attractions of model checking is its ability to generate counter-example traces when the system fails to satisfy the desired property [7]. However, these counter-example traces describe executions of the entire system, and are not immediately helpful in localizing the fault or in devising repairs. As such, these are not questions about the *current* behavior of the system, but rather, of its *desired* behavior.

Deriving and explaining LTL specifications. There has also been some concern about the inaccessibility of formal specification languages to engineers working in applied domains such as robotics, and who are themselves not necessarily experts in verification technology. To address these concerns, there has been some amount of work in making these formalisms more accessible: for example, by automatically deriving temporal logic specifications from system models [23,27], translating LTL formulas into natural language descriptions [3], and using various kinds of translation technology to convert requirements expressed in natural language into LTL, STL, MTL, and other kinds of temporal logic formulas [9,10]. Of course, these techniques focus more on issues of understanding and obtaining good specifications, rather than on the task of explaining the mechanics of the system under consideration.

Modular verification and local reasoning. The key idea in this paper was to reverse-engineer (temporal) specifications for individual components in a composite system. As such, this task is intimately tied to the problem of modular verification [13]. The promise of modular verification is that proving and composing component-level properties can lead to more scalable verification. In a sense, our hope with subspecifications is the same: that engineers will find properties of specific parts intuitive and easy-to-reason about when isolated from the rest of the system. Of course, one challenge with modular verifiers is inferring properties of individual modules. Similar challenges might also arise when applying subspecs to very large systems.

Program comprehension and repair. Subspecifications are also closely connected to the problem of program repair. Most simply, the behaviors exhibited by the program patch must satisfy the corresponding subspecification. Program repair has been extensively studied, both in the setting of large-scale code [24,22] and in the setting of reactive systems [15]. Another notable body of research focuses on program comprehension: one approach involves sophisticated techniques to visualize program executions [14], while the other—for e.g., the famous Whyline tool [20]—once again involves counterfactual questions.

References

1. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2007)
2. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design. FMCAD (2024)
3. Cherukuri, H., Ferrari, A., Spoletini, P.: Towards explainable formal methods: From ltl to natural language with neural machine translation. In: International Working Conference on Requirements Engineering: Foundation for Software Quality. pp. 79–86. Springer (2022)
4. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Journal of Symbolic Logic **28**(4) (1963)
5. Clarke, E., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. MIT Press, 2nd edn. (2018)
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on logic of programs. pp. 52–71. Springer (1981)
7. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. Communications of the ACM **52**(11), 74–84 (2009)
8. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
9. Fuggitti, F., Chakraborti, T.: NL2LTL: A Python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In: AAAI (2023), system Demonstration.
10. Fuggitti, F., Chakraborti, T.: NL2LTL: A Python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In: ICAPS (2023)
11. Goel, A., Sakallah, K.: Avr: abstractly verifying reachability. In: Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26. pp. 413–422. Springer (2020)
12. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18. pp. 358–371. Springer (2006)
13. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 843–871 (1994)
14. Guo, P.J.: Online python tutor: embeddable web-based program visualization for cs education. In: Proceeding of the 44th ACM technical symposium on Computer science education. pp. 579–584 (2013)
15. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of reactive programs. In: 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems. pp. 3–12. IEEE (2012)

16. Jacobs, S.: Extended AIGER format for synthesis. CoRR **abs/1405.5793** (2014), http://arxiv.org/abs/1405.5793
17. Jacobs, S., Perez, G., Schlehuber-Caissier, P.: Data, scripts, and results from SYNTCOMP 2023 (2023). https://doi.org/10.5281/zenodo.8161423, https://doi.org/10.5281/zenodo.8161423
18. Jobstmann, B., Bloem, R.: Optimizations for ltl synthesis. In: Formal Methods in Computer Aided Design. pp. 117–124. FMCAD (2006). https://doi.org/10.1109/FMCAD.2006.22
19. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17. pp. 226–238. Springer (2005)
20. Ko, A.J., Myers, B.A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 151–158 (2004)
21. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Automated Technology for Verification and Analysis. pp. 543–550. ATVA, Springer (2018)
22. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. Ieee transactions on software engineering **38**(1), 54–72 (2011)
23. Lemieux, C., Park, D., Beschastnikh, I.: General ltl specification mining (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 81–92. IEEE (2015)
24. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering. pp. 691–701 (2016)
25. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: International Conference on Computer Aided Verification. pp. 578–586. Springer (2018)
26. Nazari, A., Huang, Y., Samanta, R., Radhakrishna, A., Raghothaman, M.: Explainable program synthesis by localizing specifications. Proceedings of the ACM on Programming Languages **7**(OOPSLA2) (2023). https://doi.org/10.1145/3622874, https://doi.org/10.1145/3622874
27. Neider, D., Roy, R.: What is formal verification without specifications? a survey on mining ltl specifications. In: Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III, pp. 109–125. Springer (2024)
28. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 179–190 (1989)
29. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: International Symposium on programming. pp. 337–351. Springer (1982)
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus (1983), https://api.semanticscholar.org/CorpusID:123007433
31. Wolper, P.: Temporal logic can be more expressive. Information and Control **56**(1), 72–99 (1983). https://doi.org/https://doi.org/10.1016/S0019-9958(83)80051-5, https://www.sciencedirect.com/science/article/pii/S0019995883800515

A Expressibility and Automatic Derivation of Subspecs

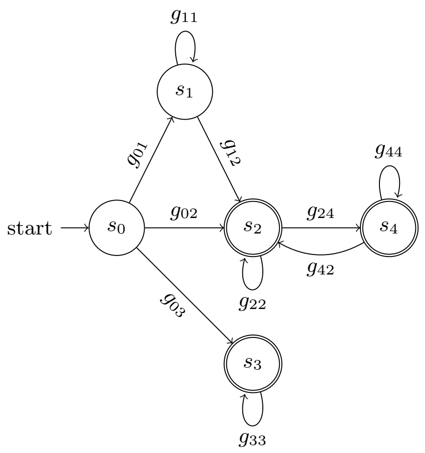


Fig. 11: Buchi automaton $M_{C,b,\phi}$ for the latch b in the circuit described in Equation 8, and with respect to the specification of Equation 7. The transition guards are provided by $g_{01} = i \wedge j \wedge \neg a \wedge b \wedge x$, $g_{02} = i \wedge j \wedge \neg a \wedge \neg b \wedge x$, $g_{03} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x)$, $g_{11} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$, $g_{12} = i \wedge j \wedge \neg a \wedge \neg b \wedge x$, $g_{22} = a \wedge \neg b \wedge \neg x$, $g_{24} = a \wedge b \wedge \neg x$, $g_{33} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$, $g_{42} = \neg a \wedge \neg b \wedge i \wedge j \wedge x$, and $g_{44} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$ respectively.