Triggers for Reactive Synthesis Specifications

Gal Amram
Tel Aviv University
Israel

Dor Ma'ayan
Tel Aviv University
Israel

Shahar Maoz
Tel Aviv University
Israel

Or Pistiner
Tel Aviv University
Israel

Jan Oliver Ringert
Bauhaus University Weimar
Germany

***Abstract*—Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. Two of the main challenges in bringing reactive synthesis to practice are its very high worst-case complexity and the difficulty of writing declarative specifications using basic LTL operators. To address the first challenge, researchers have suggested the GR(1) fragment of LTL, which has an efficient polynomial time symbolic synthesis algorithm. To address the second challenge, specification languages include higher-level constructs that aim at allowing engineers to write succinct and readable specifications. One such construct is the triggers operator, as supported, e.g., in the Property Specification Language (PSL).**

In this work we introduce triggers into specifications for reactive synthesis. The effectiveness of our contribution relies on a novel encoding of regular expressions using symbolic finite automata (SFA) and on a novel semantics for triggers that, in contrast to PSL triggers, admits an efficient translation into GR(1). We show that our triggers are expressive and succinct, and prove that our encoding is optimal.

We have implemented our ideas on top of the Spectra language and synthesizer. We demonstrate the usefulness and effectiveness of using triggers in specifications for synthesis, as well as the challenges involved in using them, via a study of more than 300 triggers written by undergraduate students who participated in a project class on writing specifications for synthesis.

To the best of our knowledge, our work is the first to introduce triggers into specifications for reactive synthesis.

I. Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. Two of the main challenges in bringing reactive synthesis to practice are its high worst-case complexity and the difficulty of writing specifications using basic LTL operators.

To address the first challenge, Piterman et al. [10], [41] have presented GR(1), an assume/guarantee fragment of LTL, with an efficient symbolic synthesis algorithm. GR(1) specifications include assumptions and guarantees about what needs to hold on all initial states, on all states and transitions (safety), and infinitely often on every run (justice). GR(1) has been used in several application domains, e.g., to specify and implement autonomous robots [26], [29], control protocols for smart camera networks [40], distributed control protocols for aircraft vehicle management systems [39], and device drivers [45]. It is supported by several tools [8], [15], [33], [52].

To handle the second challenge, specification languages include higher-level constructs that aim at allowing engineers write succinct and readable specifications. Two of these are regular expressions (REs) and triggers, as supported, e.g., in the Property Specification Language (PSL) [16], [23], an IEEE standard widely used in industrial formal verification tools. REs have many applications in computer science in general and in programming and specification languages in particular. The triggers operator, $L|{=>}R$, roughly specifies that whenever a prefix of a run satisfies the RE on the left, the run starting immediately after that should satisfy the expression on the right. Triggers are considered useful and intuitive to read and write. Live sequence charts [21] and triggered scenarios [46] can be viewed as special cases of triggers.

In this work we introduce REs and triggers into specifications for reactive synthesis. Specifically, we define a rich RE language including Boolean operators, grouping, quantification, and wildcards, where 'characters' are propositional formulas over the variables in the specification, and use REs as operands for the triggers operator. Using our REs and triggers, one can easily express many natural temporal properties that are very difficult to express directly in either LTL or GR(1).

Consider a typical Obstacle Evasion specification, which has been used as a benchmark in the literature. The general setting is an $n \times m$ grid world with a robot moving between cells, as illustrated in Fig. 1. A Spectra specification for an 8×8 setting is shown in Lst. 1. The robot (position encoded by variable `rob`) must make sure to evade a larger, moving obstacle (variable `obs`) that occupies $2{\times}2$ grid cells. In each move, the obstacle and the robot can stay in place or move to any empty adjacent cell. The robot is more agile and can do two steps upon each step of the obstacle.

The engineers developing the specification want to add a guarantee that the robot patrols between *pos1*, *pos2*, and *pos3* in that order. This corresponds to an instance of the *Strictly Ordered Patrolling* pattern from [37] and not directly supported by GR(1) synthesis. It can however be expressed by the trigger shown in lines 41-42. Since the empty word satisfies the left side RE $[\mathbf{true}]^*$ of this trigger, it immediately requires the RE on the right to be satisfied. Thus, the robot will not be at any of the named designated positions until it reaches

pos1, *pos2*, and finally *pos3*. After reaching *pos3* the right side RE is satisfied and the execution may satisfy the left side again, restarting the process.

As another example, the trigger in lines 43-44 expresses a violation (**false**) if the system does not charge for 20 consecutive states, or if the load is high and it does not charge for 14 consecutive states. Note the use of rich assertions, e.g., *load*=*HIGH* **&** **!***charge*, and powerful RE operators, e.g., Kleene star or union (operator |). Of course, both sides of a trigger may consist of REs as in the final example in lines 45-46, expressing that whenever loaded and called, the system will only move or charge until it is unloaded. Finally, note that the triggers are specified in addition to other elements in the specification, i.e., different **alw** (always, G) and **alwEv** (always eventually, GF) assumptions and guarantees. Thus, our work is not about synthesis from specifications made only of triggers but from GR(1) specifications with added triggers.

The use of REs as building blocks allows the expression of a wide range of properties, from very simple to most complex. We present the formal syntax and semantics of our REs and triggers, with examples of the properties they can express.

The effectiveness and significance of our contribution relies on three novelties. First, rather than a naive translation of REs over assertions to automata, which would result in automata that are linear in the number of assertion valuations, we use a pure symbolic translation into symbolic finite automata (SFA) [50], which is linear in the number of assertions.

Second, the semantics of our triggers is simpler than that of PSL triggers. Intuitively, while the simultaneous semantics of PSL triggers requires tracking all (overlapping) occurrences of the trigger's left operand in parallel to its right operand, our non-simultaneous semantics does not track additional occurrences of the left until the right has been satisfied. Importantly, this difference in semantics results in a lower complexity. While synthesis from PSL triggers is double-exponential [27], like that of LTL [43], synthesis from our non-simultaneous triggers requires only one exponent and thus enables efficient embedding into GR(1).

```
1  sys Int(0..7)[2] rob; //robot position
2  sys boolean charge;
3  sys {STOP, UP, DOWN, LEFT, RIGHT} moveCmd;
4  env Int(0..7)[2] obs; //obstacle position
5  env {NONE, LOW, HIGH} load;
6  env boolean called;
7  env boolean obsPause;
8
9  define pos1 := rob[0]=4 & rob[1]=0;
10 define pos2 := rob[0]=0 & rob[1]=3;
11 define pos3 := rob[0]=5 & rob[1]=5;
12 define atPos := pos1|pos2|pos3;
13 define loaded := load!=NONE;
14 define move := moveCmd!=STOP;
15
16 predicate evade(Int(0..7) rx, Int(0..7) ry,
17                 Int(0..7) ox, Int(0..7) oy) {
18   (rx = ox | rx = ox+1) -> ry !=oy & ry !=ry+1
19 }
20 predicate moveOnGrid(Int(0..7) x, Int(0..7) y) {
21   (moveLOne(x) & stay(y) | moveLOne(y) & stay(x))
22 }
23 predicate moveLOne(Int(0..7) d) {
24   next(d) = d+1 | next(d) = d | next(d) = d-1
25 }
26 predicate stay(Int(0..7) d) {
27   next(d)=d
28 }
29 asm ini !loaded;
30 asm ini obs[0]=0 & obs[1]=0;
31 asm alw next(obsPause) =! obsPause;
32 // obs moves on grid but pauses every second step
33 asm alw moveOnGrid(obs[0], obs[1]) &
34   (obsPause | (stay(obs[0]) & stay(obs[1])));
35 asm trig [true]* |=>
36             [obs[0]=0 & obs[1]=0][true]*
37             [obs[0]=7 & obs[1]=7];
38 asm alwEv loaded;
39 gar alw moveOnGrid(rob[0],rob[1]);
40 gar alw evade(rob[0], rob[1], obs[0], obs[1]);
41 gar trig [true]* |=>
42     [!atPos]*[pos1][!atPos]*[pos2][!atPos]*[pos3];
43 gar trig [true]*([!charge]{20} |
44     [load=HIGH &!charge]{14}) |=> [false];
45 gar trig [true]*[loaded & called] |=>
46     [move | charge]*[!loaded];
```

Listing 1: An obstacle evasion specification with triggers.

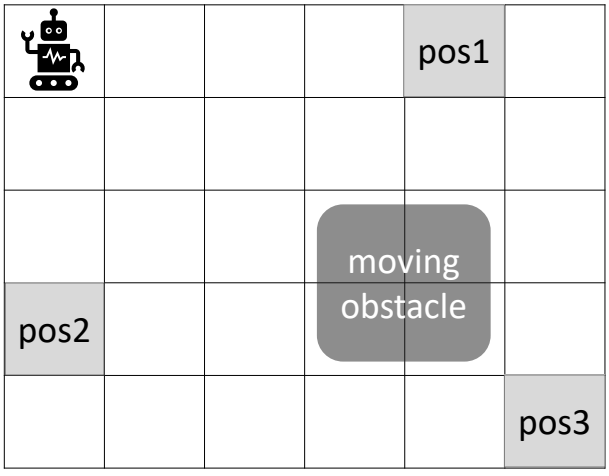

Figure 1: An illustration of the Obstacle Evasion example in a grid world with a robot to visit dedicated cells, e.g., *pos*1, while avoiding a 2x2 moving obstacle.

Third, we prove that expressing a trigger property in GR(1) requires the addition of fresh auxiliary variables and that our encoding is optimal as it adds only the minimal number of variables required. We further present properties that are expressible using a trigger and cannot be expressed in LTL without the addition of new variables, and properties that are expressible using a rather simple trigger but cannot be expressed in LTL without the nested use of the until operator. These theoretical results explain why expressing trigger properties directly in GR(1) is very difficult to do manually, and thus further strengthen the significance of our work.

We have implemented support for REs and triggers on top of Spectra [1], [33], integrating into the language and the synthesis IDE. In particular, the reduction to GR(1) is done internally and is completely transparent to the engineers who

write the specification. They never see the auxiliary variables that the reduction to GR(1) adds to the problem.

For evaluation, we report on more than 300 triggers from 30 specifications written by 20 third-year CS undergraduate students who participated in a semester-long project class on writing specifications for synthesis, as well as more than 100 meaningful mentions of triggers from the class's Slack communications. We use all these to demonstrate the potential usefulness of triggers in expressing various properties for synthesis, as well as the limitations and remaining challenges to make triggers more useful. See Sect. V.

Many have discussed the challenge of writing formal specifications, in particular in LTL context, see, e.g., [44]. To our knowledge, we are the first to efficiently introduce triggers into reactive synthesis specifications. See related work in Sect. VI.

II. Preliminaries

We use standard definitions for languages, regular expressions, DFAs and NFAs from [22] as well as for linear temporal logic (LTL) [42], infinite path, and realizability, e.g., as found in [10]. We provide background on GR(1), Spectra, and symbolic finite automata.

a) GR(1) and Spectra: LTL synthesis is computationally expensive (2EXPTIME-complete [43]). Thus, authors have suggested LTL fragments with efficient synthesis algorithms.

GR(1) is a fragment of LTL with an efficient symbolic synthesis algorithm [10], [41], whose expressive power covers most of the well-known LTL specification patterns [14], [28]. GR(1) specifications include assumptions and guarantees about what needs to hold on all initial states, on all states and transitions (safety), and infinitely often on every run (justice). A GR(1) synthesis problem consists of the following elements [10]:

- $\mathcal{X}$ is a set of input variables controlled by the environment;
- $\mathcal{Y}$ is a set of output variables controlled by the system;
- θ^e is an assertion, i.e., a propositional logic formula, over $\mathcal{X}$ characterizing initial environment states;
- θ^s is an assertion over $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$ characterizing initial system states;
- ρ^e is an assertion over $\mathcal{V} \cup \mathcal{X}'$, with $\mathcal{X}'$ a primed copy of variables $\mathcal{X}$; given a state, ρ^e restricts the next input;
- ρ^s is an assertion over $\mathcal{V} \cup \mathcal{V}'$, with $\mathcal{V}'$ a primed copy of variables $\mathcal{V}$; given a state and input, ρ^s restricts the next output;
- $J^e_{i\in 1..n}$ is a set of assertions over $\mathcal{V}$ for the environment to satisfy infinitely often (called justice assumptions);
- $J^s_{j\in 1..m}$ is a set of assertions over $\mathcal{V}$ for the system to satisfy infinitely often (called justice guarantees).

A GR(1) synthesis problem is strictly realizable[1] iff the following LTL formula is realizable:

$$\varphi^{sr} = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s)) \wedge (\theta^e \wedge \mathbf{G}\rho^e \rightarrow (\bigwedge_{i\in 1..n} \mathbf{GF}J^e_i \rightarrow \bigwedge_{j\in 1..m} \mathbf{GF}J^s_j))$$

Spectra [1], [33] is a specification language and a synthesis environment that includes a GR(1) synthesizer. Spectra extends GR(1)'s Boolean variables to finite-type variables, e.g., enumerations and bounded integers. Beyond GR(1) with several performance heuristics [18], it includes extensions that are reduced into GR(1), e.g., patterns [28]. It comes with several analyses, e.g., well-separation detection [30] and repairs [34].

b) Symbolic Finite Automata (SFA): An *assertion* over a set of variables $\mathcal{V}$ is a Boolean valued expression over variables in $\mathcal{V}$ and their respective operators, e.g., $load$**=**$HIGH$ **&** **!**$charge$. $A(\mathcal{V})$ is the set of all assertions over $\mathcal{V}$. A *valuation* val over $\mathcal{V}$ assigns each $v \in \mathcal{V}$ a value $val(v)$ in v's type. $V(\mathcal{V})$ is the set of all valuations over $\mathcal{V}$.

A symbolic finite automaton (SFA) [50] is a finite automaton that employs assertions (rather than symbols) on the transitions. An SFA over a set of variables $\mathcal{V}$ is a tuple $\mathcal{A} = (Q, \Delta, q_0, \mathcal{F})$, where (1) Q is a finite set of automata states; (2) $\Delta \subseteq Q \times (A(\mathcal{V}) \cup \{\varepsilon\}) \times Q$ is a finite transition relation; (3) $q_0 \in Q$ is the initial state; and (4) $\mathcal{F} \subseteq Q$ are the accepting states. An SFA is deterministic (DSFA) if it has no ε-transitions, and for any two transitions $(q, assrt_1, q_1), (q, assrt_2, q_2)$ with $q_1 \neq q_2$, $\forall val \in V(\mathcal{V})$, $\neg(val \models assrt_1 \wedge assrt_2)$. We use δ to denote the transition relation of a DSFA. Variants for standard constructions on NFAs and DFAs exist for SFAs [50].

III. Syntax and Semantics

We describe the syntax and semantics of our REs and triggers. The syntax and semantics we present match the way we have implemented these in Spectra.

A. Regular Expressions Syntax and Semantics

The basic 'characters' of the REs we deal with are **assertions** over the variables that appear in the specification. Assertions are written inside [..] brackets. Naturally, the semantics of an assertion is the language $L([assrt]) = \{s \in V(\mathcal{V}) : s \models assrt\}$, whose words are sequences of valuations. On top of the basic assertions, REs are defined by the following operators and semantics:
(**concatenation**) $L((re_1)(re_2)) = L(re_1)L(re_2)$;
(**union**) $L((re_1)|(re_2)) = L(re_1) \cup L(re_2)$;
(**intersection**) $L((re_1)\&(re_2)) = L(re_1) \cap L(re_2)$;
(**kleene-star**) $L((re_1)^*) = \bigcup_{i=0}^{\infty}(L(re_1))^i$;
(**plus**) $L((re_1)^+) = \bigcup_{i=1}^{\infty}(L(re_1))^i$;
(**zero-or-one**) $L((re_1)?) = \{\varepsilon\} \cup L(re_1)$;
(**exact-repetitions**) $L((re_1)\{k\}) = (L(re_1))^k$;
(**at-least-repetitions**) $L((re_1)\{k,\}) = \bigcup_{i=k}^{\infty}(L(re_1))^i$;
(**ranged-repetitions**) $L((re_1)\{k,m\}) = \bigcup_{i=k}^{m}(L(re_1))^i$;
(**negation**) $L(\sim(re_1)) = (V(\mathcal{V}))^* \setminus L(re_1)$.

[1]Note that many authors present the simpler and more intuitive formula for implication realizability rather than the one for strict realizability. For a comparison and a reduction between two see [10], [25].

Example 1: The RE from the left side of the trigger shown in Lst. 1 in lines 43-44 includes the operators Kleene star, concatenation, exact repetition, and union.

Note that the RE operators we support are rich and enable the expression of many other operators, including, e.g., SERE non-length matching and PSL's variants of consecutive and non-consecutive counting [16].

B. Triggers Syntax and Semantics

For REs $regexp_1$ and $regexp_2$, a trigger is written $regexp_1|=>regexp_2$. $regexp_1$ is the *prefix* RE of the trigger, and $regexp_2$ is the *suffix* RE of the trigger. Roughly, triggers express a conditional satisfaction between two REs: whenever $regexp_1$ holds, $regexp_2$ should hold afterwards.

To formally define the semantics of the triggers operator, we start by defining the *first-of* operator. Given a language L, the language *first-of* L, denoted $first(L)$, is the language that includes all words in L that have no proper prefix in L. For a language defined by an RE, the *first-of* operator intuitively yields only those words that first match the RE.

Definition 1 (first of L): A word u is a proper prefix of w, if $w = uv$ for some word $v \neq \varepsilon$. For a language L, $first(L) = \{w \in L : w \text{ has no proper prefix in } L\}$.

Example 2: Consider the following variants of RE examples from Lst. 1 and their corresponding *first-of* languages:

- $first(L([!charge]^+)) = L([!charge])$
- $first(L([\texttt{true}]^*[loaded])) = L([!loaded]^*[loaded])$
- $first(L([\texttt{true}]^*)) = \varepsilon$

To decide if a path satisfies a trigger $regexp_1|=>regexp_2$, one can seek, consecutively, words in $first(L(regexp_1))$ and in $first(L(regexp_2))$. The path satisfies the trigger if either this process never ends, or it ends while seeking a word in $first(L(regexp_1))$. Formally:

Definition 2 (trigger semantics): Let $\pi \in (V(\mathcal{V}))^\omega$, and let $regexp_1$ and $regexp_2$ be REs over $2^{\mathcal{V}}$. We say that π satisfies the trigger $regexp_1|=>regexp_2$, and write $\pi \models regexp_1|=>regexp_2$, if one of the following holds:

1) $\varepsilon \in L(regexp_2)$.
2) $\varepsilon \notin L(regexp_2)$ and $\pi = w_0 w_1 \ldots$, where for each i, $w_{2i} \in first(L(regexp_1))$ and $w_{2i+1} \in first(L(regexp_2))$.
3) $\varepsilon \notin L(regexp_2)$ and $\pi = w_0 w_1 \ldots w_{2j+1} w'$, where:
   a) For each $i \leq j$: $w_{2i} \in first(L(regexp_1))$ and $w_{2i+1} \in first(L(regexp_2))$.
   b) No prefix of w' belongs to $first(L(regexp_1))$.

Example 3: The trigger from Lst. 1 lines 41-42 must be satisfied via case (2) where each $w_{2i} = \varepsilon$ and each w_{2i+1} visits $pos1$ to $pos3$. The trigger from lines 45-46 must be satisfied via case (3) where $\pi = w'$ ($[\texttt{false}]$ is not in $V(\mathcal{V})$).

Note that the semantics of our triggers is slightly different than that of PSL triggers. Specifically, in contrast to PSL, we consider a *non-simultaneous* semantics: whenever $regexp_1$ is satisfied, our semantics ignores $regexp_1$ until $regexp_2$ is satisfied. This simpler semantics enables an efficient encoding of triggers into the GR(1) fragment, as we show in Sect. IV-B. We further discuss PSL triggers vs. ours in Sect. VI.

Finally, any trigger can be added to a specification as an assumption or as a guarantee (the full specification of our example in Lst. 1 uses triggers as assumptions and guarantees).

C. Expressiveness and Succinctness

We now show that triggers are both expressive and succinct. They can express properties that cannot be expressed in LTL, and express complicated LTL properties in a succinct manner.

First, it is well known that no LTL formula can express the property "*proposition a holds at least in every second step*" [19] (without the addition of variables). Importantly, the trigger $[\texttt{true}]| => [a]$ expresses this property. This demonstrates the eminent expressive power of triggers.

Second, triggers can easily express properties that are very complex to express in LTL. Specifically, expressing the property *"there is no subsequence of the run in which a holds k-times, but b does not hold"* in LTL is fundamentally difficult, as it cannot be expressed without k nested instances of the until (**U**) operator [17]. Nevertheless, this property is captured by the rather simple trigger $[\texttt{true}]^*((([\texttt{true}]^*[a])\{k\})\&([!b]^*))|=>[\texttt{false}]$. This demonstrates the succinctness of triggers.

IV. Encoding

As both REs and triggers are not directly expressible in GR(1), to integrate them into the synthesis problem we have to find an encoding that preserves their semantics. We present their encoding through finite automata.

A naive approach could translate REs into finite automata using an existing state-of-the-art library like brics [38]. Indeed, brics supports a translation from REs to NFA, as well as minimization and determinization of NFAs. In our context, however, where the REs are defined over assertions, this approach would be computationally costly: in this naive approach, the size of the generated NFA would be linear in the number of valuations of assertions. Therefore, instead, we use a translation of REs into SFAs, which preserves the symbolic nature of our specification, and involves automata whose size is linear in the number of assertions, not valuations.

Below we present the encoding of REs as SFAs. We then present the encoding of triggers as SFAs.

A. Encoding REs

The key to the symbolic approach is to transform each assertion $[assrt]$ in the RE into an SFA consisting of two states and a single transition between them: an initial state with a single transition labeled by $assrt$, leading to an accepting state.

Example 4: As an example, the assertion $[load{=}HIGH \texttt{ \& !}charge]$ is translated into the two-state SFA shown in Fig. 2.

After the translation of each of the assertions into SFAs, we perform a construction over these SFAs, induced by the RE operators, following [22].

Finally, we determinize and minimize the resulting SFA by applying algorithms from [50]. Note that the complexity of the translation of REs into SFAs is linear. Also note that the

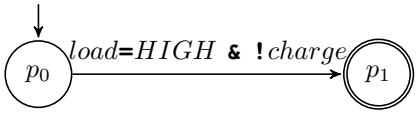

Figure 2: A two-state SFA for the assertion $[load{=}HIGH \texttt{ \& } \texttt{!}charge]$

complexity of SFA determinization and minimization does not depend on the alphabet size as in the classical case [22], but it involves Boolean operations and satisfiability checks over assertions – operations that are natural and very efficient when using symbolic BDD-based encodings.

Determinization is crucial for the correct encoding of triggers into GR(1). Minimization is important in order to improve the performance as it reduces the number of variables added for GR(1) synthesis with triggers. See next.

B. Encoding Triggers

We encode triggers into the specification via the addition of an auxiliary variable, several safety guarantees, and a justice formula. The construction applies to triggers in general, the only difference between their use as assumptions or guarantees is in adding the justice formulas to J^e or J^s.

1) Overview: Our translation of a trigger $trig = regexp_1 |{=}{>} regexp_2$ consists of the three steps shown in Fig. 6. First, in step I, we construct DSFAs for $first(L(regexp_1))$ and $first(L(regexp_2))$, $\mathcal{A}^1$ and $\mathcal{A}^2$, respectively. Then, in step II, we construct a DSFA, $\mathcal{A}_{trig}$, which tracks past valuations. As we show, the path satisfies the trigger iff the DSFA $\mathcal{A}_{trig}$ reaches an accepting automata state infinitely often, i.e., iff a Büchi acceptance condition [19, Chapter 1] over the DSFA is met. Finally, in step III below, we add an auxiliary variable aux to the synthesis problem, to track the automata state of the DSFA $\mathcal{A}_{trig}$, as done, e.g., in [28]. Since $\mathcal{A}_{trig}$ is deterministic, this added variable is indeed an auxiliary variable, i.e., its value is determined by the values of all other variables. By the property that the constructed DSFA satisfies, the trigger holds iff $\texttt{GF}(aux \in \mathcal{F}_{trig})$ holds, where $\mathcal{F}_{trig}$ is the set of the DSFA $\mathcal{A}_{trig}$ accepting states.

2) Construction: We now describe the construction of the trigger DSFA $\mathcal{A}_{trig}$. We fix a synthesis problem $GS = (\mathcal{X}, \mathcal{Y}, \theta^e, \theta^s, \rho^e, \rho^s, J^e, J^s)$, and a trigger formula that serves as a guarantee or an assumption, $trig = regexp_1 |{=}{>} regexp_2$.

Step I: DSFAs for $first(L(regexp_1))$ and $first(L(regexp_2))$

First, we construct DSFAs for the languages $first(L(regexp_1))$ and $first(L(regexp_2))$, $\mathcal{A}^1 = \{Q^1 = \{q^1_0, \ldots, q^1_{k_1}\}, \delta^1, q^1_0, \mathcal{F}^1\}$ and $\mathcal{A}^2 = \{Q^2 = \{q^2_0, \ldots, q^2_{k_2}\}, \delta^2, q^2_0, \mathcal{F}^2\}$, resp., following the construction we described in Sect. IV-A. [2]

[2]After constructing DSFAs for $regexp_1$ and $regexp_2$, we just remove outgoing edges with accepting sources to obtain DSFAs for $first(L(regexp_1))$ and $first(L(regexp_2))$.

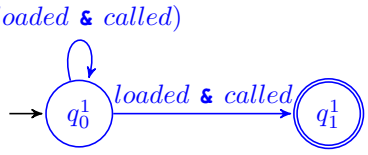

Figure 3: The deterministic SFA for the *first-of* language of $regexp_1 = [\texttt{true}]^*[loaded \texttt{ \& } called]$

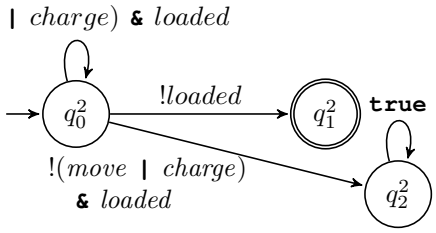

Figure 4: The deterministic SFA for the *first-of* language of $regexp_2 = [move \texttt{ | } charge]^*[\texttt{!}loaded]$

Example 5: Figure 3 shows the deterministic SFA resulting from the translation of $regexp_1 = [\texttt{true}]^*[loaded \texttt{ \& } called]$. The deterministic SFA for $regexp_2 = [move \texttt{ | } charge]^*[\texttt{!}loaded]$ from the same trigger is structurally similar with an additional sink state for unrecoverable violations of $regexp_2$. See Fig. 4.

Step II: Constructing $\mathcal{A}_{trig}$

Given the two DSFAs, $\mathcal{A}^1$ and $\mathcal{A}^2$, we construct a DFSA $\mathcal{A}_{trig}$ by concatenating them as follows. Whenever $\mathcal{A}^1$ accepts, we go to the initial state of $\mathcal{A}^2$, and whenever $\mathcal{A}^2$ accepts, we go to the initial state of $\mathcal{A}^1$. A path π satisfies the trigger iff its computation (a) traverses between $\mathcal{A}^1$ and $\mathcal{A}^2$ infinitely often, or (b) from some point on, never leaves $\mathcal{A}^1$ (compare to cases (2) and (3) in Def. 2). Hence, we redirect transitions to $\mathcal{F}^1$ into q^2_0, redirect transitions to $\mathcal{F}^2$ into q^1_0, and mark all states of $\mathcal{A}^1$ as accepting. [3] Formally: Let

- $\delta^1_{\mathcal{F}^1} = \{(q, assrt, q') \in \delta^1 : q \in Q^1 \setminus \mathcal{F}^1, q' \in \mathcal{F}^1\}$,
- $\delta^1_{q^2_0} = \{(q, assrt, q^2_0) : \exists q'((q, assrt, q') \in \delta^1_{\mathcal{F}^1})\}$,
- $\delta^2_{\mathcal{F}^2} = \{(q, assrt, q') \in \delta^2 : q \in Q^2 \setminus \mathcal{F}^2, q' \in \mathcal{F}^2\}$, and
- $\delta^2_{q^1_0} = \{(q, assrt, q^1_0) : \exists q'((q, assrt, q') \in \delta^2_{\mathcal{F}^2})\}$.

Then, $\mathcal{A}_{trig} = (Q_{trig} = (Q^1 \cup Q^2) \setminus (\mathcal{F}^1 \cup \mathcal{F}^2), \delta_{trig} = ((\delta^1 \setminus \delta^1_{\mathcal{F}^1}) \cup \delta^1_{q^2_0}) \cup ((\delta^2 \setminus \delta^2_{\mathcal{F}^2}) \cup \delta^2_{q^1_0}), q_0 = q^1_0, \mathcal{F}_{trig} = Q^1 \setminus \mathcal{F}^1)$.

Example 6: Both REs of the trigger below, from Lst. 1, have identical structures with different assertions, see Example 5:

$$[\texttt{true}]^*[loaded \texttt{ \& } called] \; |{=}{>} [move \texttt{ | } charge]^*[\texttt{!}loaded]$$

Figure 5 shows $\mathcal{A}_{trig}$ obtained from Step II. The blue states and transitions originate from the left RE and the black ones from the right RE.

The single accepting state q^1_0 of $\mathcal{A}_{trig}$ matches our intuition: either the robot is never loaded and called or it must move or charge until it is unloaded.

[3]Slightly different constructions are used in the special cases where $\varepsilon \in L(\mathcal{A}^1)$ or $\varepsilon \in L(\mathcal{A}^2)$. See [4].

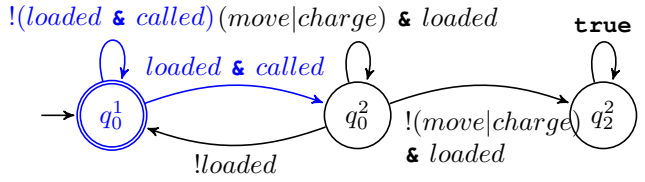

Figure 5: The deterministic SFA for $\mathcal{A}_{trig}$

```
sys {q1, q2, q3} aux;
gar aux = q1;
gar G
 (aux=q1 & !(loaded & called) & next(aux=q1)) |
 (aux=q1 & (loaded & called) & next(aux=q2)) |
 (aux=q2 & (move | charge) & loaded & next(aux=q2))
     |
 (aux=q2 & !(move | charge) & loaded & next(aux=q3)
    ) |
 (aux=q2 & !loaded & next(aux=q1)) |
 (aux=q3 & next(aux=q3));
gar GF aux = q1;
```

Listing 2: Spectra code of GR(1) translation (Example 7) of $\mathcal{A}_{trig}$ from Example 6

Importantly, note that $\mathcal{A}_{trig}$ is a DSFA; since we change transition targets but not sources, the transition relation remains deterministic. In step III, see below, we add an auxiliary variable aux that tracks the state of $\mathcal{A}_{trig}$. If we would have used an NSFA, we could have tracked via aux an "incorrect path" which does not traverse $\mathcal{F}_{trig}$ infinitely often. Then, we could have wrongly decide that the path does not satisfy $trig$.

Step III: Updating the synthesis problem

Given $\mathcal{A}_{trig}$, we augment the synthesis problem with a new auxiliary variable aux that tracks the state of $\mathcal{A}_{trig}$. aux stores values from Q_{trig}, and its value is uniquely determined at each step, based on its old value, the values of the other variables, and the matching transition of $\mathcal{A}_{trig}$.

Formally, we update the synthesis problem into $(\mathcal{X}, \mathcal{Y} \cup \{aux\}, \theta^e, \hat{\theta}^s, \rho^e, \hat{\rho}^s, \hat{J}^e, \hat{J}^s)$, where[4] $\hat{\theta}^s = \theta^s \wedge (aux = q_0)$,

$$\hat{\rho}^s = \rho^s \wedge \bigvee_{(q, assrt, q') \in \delta_{trig}} (aux = q \wedge assrt \wedge aux' = q')$$

and $\hat{J}^e = J^e \cup \{aux \in \mathcal{F}_{trig}\}$ if the trigger is an assumption, or $\hat{J}^s = J^s \cup \{aux \in \mathcal{F}_{trig}\}$ if the trigger is a guarantee.

Example 7: For $\mathcal{A}_{trig}$ from Example 6, Step III adds the elements shown in Lst. 2 (in Spectra code) where values `q1`, `q2`, and `q3` of variable `aux` correspond to states q_0^1, q_0^2, and q_2^2 respectively.

This completes the construction. Note that the construction is linear in the sizes of $\mathcal{A}^1$ and $\mathcal{A}^2$. We provide a rigorous proof of correctness in [4].

C. Optimality of the Encoding

Our encoding adds auxiliary variables to the synthesis problem. Specifically, for a trigger $re_1 |=> re_2$, our encoding adds up to $\log(|Q^1| + |Q^2| - 2)$ Boolean variables, where Q^1, Q^2 are the sets of states of the DFAs for re_1, re_2 resp.

We now claim that our encoding is optimal: $\log(Q^1 + Q^2 - 2)$ is both an upper bound and a (worst-case) lower bound on the number of auxiliary variables required to encode triggers.

Theorem 1: No GR(1) encoding of triggers adds less than $\log(|Q^1| + |Q^2| - 2)$ auxiliary variables in the worst-case.

Proof sketch. For $\mathcal{V} = \{v\}$ and $k > 1$, use the trigger $[\texttt{true}]\{2^k - 1\}|=>[v]$, which formulates the requirement that v must hold every 2^k steps. Show that in this case our encoding adds k variables, and that every GR(1) encoding adds at least k variables. The complete proof appears in [4].

V. Implementation, Validation, Evaluation

Implementation We have implemented and integrated REs and triggers in the Spectra language and synthesis environment, along with their translation into GR(1), as elaborated in Sect. IV. Specifically, to support SFAs in Spectra, we implemented automata with transitions equipped by BDDs [13] via the CUDD 3.0 package [48], following the algorithms in [50]. We implemented our pure symbolic technique, in which we transform the first and second REs into SFAs, as described in Sect. IV-A, determinize and minimize the SFAs [50], and then apply the construction elaborated in Sect. IV-B.

The addition of triggers is integrated to the Spectra code which is available on https://github.com/SpectraSynthesizer.

Validation To validate our implementation's correctness, we created an alternative implementation using brics [38], a well-known library for finite-state automata and REs, to be used as a test oracle. We randomly generated hundreds of REs and triggers, and verified that (1) brics's NFA and our SFA constructions output equivalent automata; and that (2) our two trigger construction implementations output the same trigger SFA. Furthermore, over small synthetic triggers, we manually verified that the constructed trigger SFA is as expected.

Evaluation We consider the following research questions.

RQ1 Can triggers be useful in expressing properties in real-world setting?

RQ2 What are some benefits, limitations, and remaining challenges to make triggers more useful?

RQ3 What is the computational cost of supporting triggers?

User study setting: The Spectra Workshop Class

As the main instrument for answering our RQs, we used data collected in a workshop class in our university. All the participants were senior CS undergraduate students with a strong background in programming and computer sciences: they had already completed classes with Python, Java, and C projects, and took courses in data structures and algorithms. We chose to conduct our study on students since reactive synthesis and Spectra are not widely used in the industry yet, and since studies show that in many cases, students are not a significant threat in empirical software engineering studies [24]. Moreover, more than 50% of the participants already have industry student positions as software engineers.

[4]We abuse notation of assertions and values of variable aux to avoid clutter.

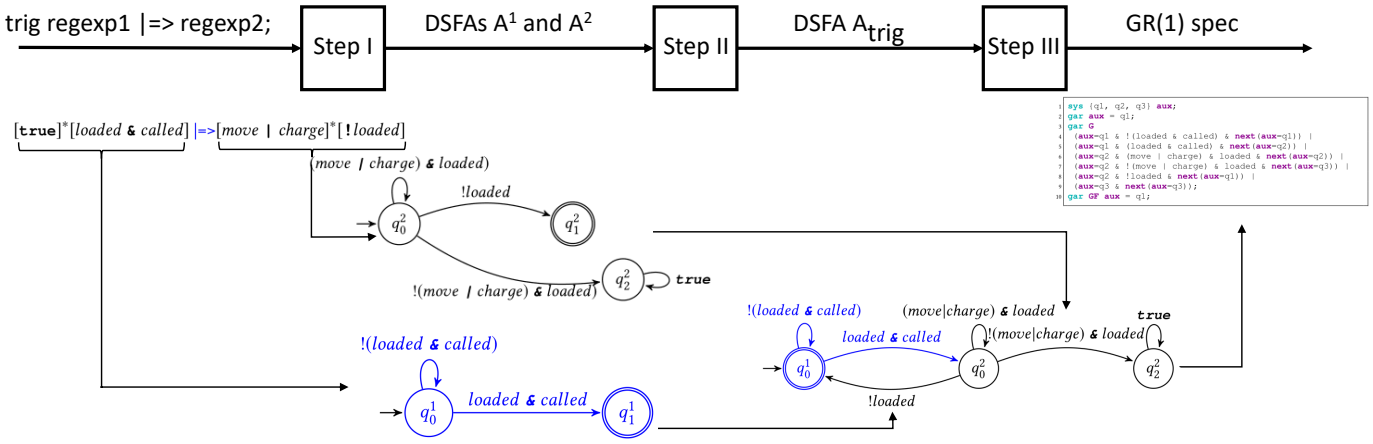

Figure 6: An overview of the encoding of triggers into GR(1)

Twenty out of the twenty-two students who registered to the workshop class signed the consent forms and agreed to participate in this study. As compensation, and with approval of the university's IRB, we offered four bonus points for the workshop's final grade for those agreeing to participate in our study. Participants were aware that we used data collected in the workshop to evaluate Spectra; however, they were unaware of which language features or analyses we evaluated.

The workshop spanned over the 13 weeks of a semester. In the first four weeks, we introduced the students to reactive synthesis and most of Spectra's features. We taught the students the syntax and semantics of triggers the same way we taught all other language features, e.g., counters and patterns. Based on our experience from pilot studies conducted in previous years, we improved the documentation of using triggers, e.g., we gave participants Lst. 3, a list of idioms for using triggers, which contains examples of possible usage. As homework practice, we asked the students to view and complete the exercises of the Spectra hands-on online tutorial [32].

After the first four weeks, we asked the students to complete six development tasks in teams of two students each. The development tasks were diverse, in different levels of difficulty, and were based, in part, on common specification tasks in the literature [37]. We described them at a rather high, semi-formal level, in natural language. Students were free to implement their specifications in any way they found appropriate, without us guiding them to use or not to use certain Spectra language features. In total, we collected at this stage 60 specifications, written by 10 teams. On average, the length of a specification was 141 lines (excluding blank lines and comments), and it contained 16 environment variables, 35 system variables (excluding auxiliary variables generated internally as part of encoding), 10 assumptions, and 24 guarantees.

RQ1: Usefulness of triggers in expressing properties

We observed extensive use of triggers in specifications written by the workshop participants. In total, we found **285 different instances of triggers in 30 different specifications written by 10 different teams (i.e., in 50% of the specifications)** (for comparison, the Dwyer et al. patterns [14], [28], another Spectra language feature, appeared only in 19 different specifications, i.e, in 31% of the specifications). Considering triggers were introduced to the students as any other language feature, we conclude that triggers can be appropriate to express different properties.

We will show a few examples of triggers written by workshop participants alongside a text description of the property they intend to describe (based on the documentation written by the participants in the specification). The complete set of triggers is available as part of the supplementary material package of the paper [3], [4].

Property 1

Whenever there is an engine problem, the environment turns on goHome and does not report any more engine problems until the robot is at home, and there will not be another engine problem for at least the next 16 states.

T2 (we will use Tx to refer to team-x) used 2 triggers responding to an engine problem to specify this property:

```
asm trig [true]*[engineProblem] |=>
   [goHome & !engineProblem]*[home];

asm trig [true]*[engineProblem]|=>
   [!engineProblem]{16,};
```

Property 2

When a car arrives in a traffic light, the junction will eventually have a green light.

T10 specified this property using the following trigger:

```
gar trig [true]*[vehiclesNorth[0] = CAR] |=>
    [true]*[greenNorthVehicles[0]];
```

This is an example of a "response" pattern. We have seen many instances of this pattern in the specifications written by the students.

Property 3

The robot does not stall too much when it needs to go back to start.

T4 specified this property using the following trigger:

```
gar trig [true]*[needToVisitStart] |=>
    [true]{0, 9}[atStart];
```

This is an example of a "bounded eventuality" pattern, i.e., describing that some state occurs at no more than a certain number of steps after another state. It is an instance of the one before last idiom in Lst. 3.

Property 4

After every 5 consecutive states in the orange zone, the robot should stay in place for at least 2 states and then wait for a green light before it continues to move.

T1 specified this property using the following trigger:

```
gar trig [start]*[orangeZone]{5} |=>
 [shouldStay]{2,}[!shouldStay -> greenLight];
```

Property 5

When a robot visits a location, the location color is changed to red for between 3 to 5 turns.

T5 specified this property using the following trigger:

```
gar trig [true]*[at_targetA & !engine_problem] |=>
 [targetA_color = RED]{2,4}[targetA_color = GREEN];
```

Property 6

When the robot visits a target location, it should stay there for at least 5 states (for cleaning) before it may move to another cell.

T1 specified this property using the following trigger:

```
gar trig [start]*[atTarget] |=> [shouldStay]{5,};
```

Finally, as with any language feature, we observed some "smells", i.e., usages of triggers that negatively impact design quality. We demonstrate two of them. First, consider the following property:

Property 7

The robot should rotate if and only if it is in the orange zone.

T2 specified this property using the following trigger:

```
gar trig [true]*[(rotate & !atOrange) | (!rotate &
    atOrange)] |=> [false];
```

However, a much simpler formulation of this property, which involves no triggers, is of course available in Spectra:

```
gar alw rotate <-> atOrange;
```

Second, consider the following property:

Property 8

The robot should never stay in the orange zone for more than 5 consecutive steps.

T3 specified this property using a combination of triggers:

```
gar trig[true]*[orange_zone_counter = 0][
    at_Orange_Zone and orange_zone_counter != 1]
    |=> [false];
gar trig[true]*[orange_zone_counter = 1][
    at_Orange_Zone and orange_zone_counter != 2]
    |=> [false];
....
gar trig[true]*[orange_zone_counter = 5][
    at_Orange_Zone and orange_zone_counter != 6]
    |=> [false];
gar trig[true]*[orange_zone_counter = 6][
    at_Orange_Zone and greenLight = false and
    orange_zone_counter != 6] |=> [false];
```

That is, they used triggers to explicitly count the number of steps in the orange zone, in a variable named `orange_zone_counter`. However, a much simpler solution, which does not involve explicit counting and uses only one trigger can be used, i.e., an instance of the second idiom in Lst. 3:

```
gar trig [true]*[at_Orange_Zone]{6} |=> [false];
```

Note that from a preliminary manual inspection of the specifications, we did not observe a significant number of smelly usages of triggers. Still, identifying such smells is important in order to improve the documentation and teaching of triggers in the future.

RQ2: Benefits and limitations of triggers

The central communication platform with workshop participants was Slack. Using Slack, participants reported their weekly progress in tasks, asked questions on language features, and shared with the workshop teams insights regarding different aspects of Spectra.

We found 103 meaningful mentions of triggers in the Slack communication with the workshop participants. To draw meaningful conclusions from them, we performed a thematic analysis [12], [11] and identified advantages, disadvantages, and usability issues with triggers. We started by identifying all the text fragments mentioning triggers. Then, with multiple open-coding sessions, we identified different themes and discussed them between the authors until agreement. We will now present the main themes that emerged from our analysis.

1) Triggers ease the expression of some properties: Our analysis reveals that in many cases, participants found triggers as a more intuitive and elegant solution to express some properties which they considered to be easier to describe using regular expressions rather than other language constructs (T1,T2,T5,T6,T10). For example:

> "*...It is extremely helpful in this kind of development (where we want to react to what is happening instead of just following a simple procedure like in common program development), it is easy to read and understand, and most importantly, it was easy to translate our thoughts and intentions to a regular expression.*" **(T2)**

```
// regexp should not hold from the beginning
 trig regexp |=> [false];

// regexp should never hold
 trig [true]* regexp |=> [false];

// regexp should hold at most k consecutive times
 trig [true]* (regexp){k + 1} |=> [false];

// regexp should hold at most k times
 trig ([true]* regexp){k + 1} |=> [false];

// regexp should hold infinitely often
 trig [true]* |=> [true]* regexp;

// regexp should hold repeatidly
 trig [true]* |=> regexp;

// regexp should hold in intervals of k states
 trig [true]{k} |=> regexp;

// if re2 holds after re1, the distance
// between them must not exceed k states
 trig [true]* re1 [true]{k+1} re2 |=> [false];

// whenever regexp1 holds,
// regexp2 should eventually hold
 trig [true]* regexp1 |=> [true]* regexp2;

// whenever regexp1 holds,
// regexp2 should hold after exactly k states
 trig [true]* regexp1 |=> [true]{k} regexp2;

// whenever regexp1 holds,
// regexp2 should hold after at most k states
 trig [true]* regexp1 |=> [true]{0,k} regexp2;

// once regexp1 holds, regexp2 should never hold
 trig [true]* regexp1 [true]* regexp2 |=> [false];
```

Listing 3: Examples of idiomatic triggers in Spectra, as presented to the workshop's participants

> "*Triggers were relatively intuitive and helped us a lot in variant 3.*" **(T5)**

In addition, participants who changed properties that they originally specified without triggers to use triggers, describe, in some cases, that the new formulation of a property is simpler, shorter, and more elegant (T2,T5,T9). For example, T9 described changing certain property specification from using a counter to a trigger:

> "*. . . this is also a nicer and shorter solution compared to the counter . . .*" **(T9)**

T10 described a development strategy in which they use a trigger to define the general behavior of a system, and in order to exclude some special cases, use other language constructs:

> "*We've decided to deal with it with a trigger that guarantees that when an emergency vehicle approaches, it'll wait no longer than 2 states until it gets green light, which is the best the system can actually guarantee (excluding specific situations like the freezeMode and constructions on the north-east road). In order to deal with the special situations, we've excluded them from the trigger and made sure in other cases that always eventually all vehicles will have a chance to cross the road.*" **(T10)**

2) Perceived limitations of triggers: Despite the many benefits of triggers, we found out that in some cases, participants considered triggers to have some expressiveness limitations.

For example, participants wanted triggers to allow to use Spectra's **`next()`** operator (T4, T5):

> "*During our work we encounter a language limitation - it's impossible to combine future variables inside triggers sometimes it could be useful to use it though. For example when we want to guarantee the robot will stay in the same positioni after there was no target for 8 states we tried to use trigger with our pattern and we received that Regular expressions can't have primed* **`next()`** *variables.*" **(T4)**

As another example, T2 and T9 mentioned a problem to maintain a variable value false outside the scope of a trigger:

> "*The original idea was to use a trigger which on the left side had something like [true]*([!finishedCleaning]*[finishedCleaning]){5}, and on its right side marks the tank as full and makes sure the robot heads home. It seemed like a convenient way to solve this, but we found out there was no easy way to keep fullTank false when we don't want it to be true.*" **(T2)**

As there are simple solutions to these perceived limitations, our main conclusion from the above is the need to improve the documentation and teaching of triggers in the future.

3) Performance issues while using triggers: Although many participants considered triggers intuitive and capable of clearly describing various properties, all complained that adding many triggers slows down the synthesis times.

> "*In the context of runtime, the use of a lot of triggers makes the spec very heavy . . .*" **(T2)**

We agree with the need for performance improvements, but we do not think that slow performance is directly caused by the use of triggers. Rather, the slow performance is due to the need to express complex properties, regardless of how they are expressed. As we showed in Sect. IV-C, our encoding for triggers is optimal, so there is no general way to express the same properties more efficiently. Future studies should improve the performance of synthesis, regardless of the use of any specific language construct. Another possible cause for the slow performance is the suboptimal use of excessive triggers, as in the smells we report about above. Smells problems may be addressed in the future with better presentation and teaching of triggers and with studies aimed to (1) automatically identify such smells and (2) propose automatic refactoring solutions to replace such smells with more efficient representations.

RQ3: Computation cost of triggers

For all the 285 trigger examples we have collected during the user study, we computed the number of auxiliary variables that each trigger adds to the synthesis problem. Obviously, in theory, one can use arbitrarily complex REs and triggers,

which will translate into an arbitrary large number of additional variables. As we observe, however, in practice, this was not the case. **We found out that on average, triggers added only 2.3 auxiliary variables (with SD of 1.12, and median of 2), and that none of these triggers added more than 5 auxiliary variables to the synthesis problem**.

Given that the Spectra synthesizer already handles specifications with 30-50 variables and more [18], [33], we consider a cost of 2-5 additional variables per trigger to be reasonable, as it is comparable to the cost of adding instances of the Dwyer et al. patterns to GR(1), see [14], [28].

We further report performance results for the 30 specifications that included at least one trigger instance. Means to reproduce the experiments are available in the supplementary material package of the paper [3], [4]. For the experiments we used an AWS t3.2xlarge instance (representing an ordinary laptop with up to 3.1GHz CPU and 32GB RAM). Note that CUDD uses only one core of the CPU. We repeated each experiment 10 times, to address for variance due to the use of CUDD and Java garbage collection. We used a timeout of 10 minutes. 25 of the 30 specifications (83%) completed realizability checking within the 10 minutes timeout. For these 25 specifications, realizability checking time average was 116 seconds, median was 45 seconds. 24 of these specifications were realizable. For these 24 specifications, just-in-time synthesis [36] time average was 32 seconds, median was 19 seconds. We consider these times to be reasonable.

To answer RQ3, we conclude that support for triggers does not come for free, however, its computational cost in terms of additional variables is reasonable. Recall that by Theorem 1, our encoding adds the minimal number of variables required to express the trigger property. Thus, any alternative, automatic or manual way of expressing the same property, will not add less variables to the synthesis problem.

Threats to validity First, as our implementation of the three steps of the encoding defined in Sect. IV-B may have bugs, we validated it using an alternative implementation, see above. The fact that tens of class participants (in the class we report on and in previous ones) have already used our implementation and did not complain about unexpected behavior, reduces this threat further.

Second, w.r.t. the use of students rather than experienced software engineers we refer the reader to the first paragraph under "User study setting".

Third, one may question the correctness of the specifications written by the students. Indeed, although some of the specification tasks were in part based on examples from literature, all tasks were rather open in their formalization and allowed many variants, i.e., correctness would have to be decided case-by-case. Still, the specifications were correct in the sense that students were able to check their realizability, synthesize a controller, and use this controller to execute simulations to their satisfaction. Note that students were obviously motivated to write correct specifications. The class instructors checked the specifications and executed the synthesized controllers.

Fourth, we collected specifications for systems that were defined as class projects; as we reported above, the specifications are not small and not trivial, they intend to model real-world systems, but they are not set up in industry. As we are the first to introduce triggers to specifications for synthesis, existing benchmarks do not include them. In the future we hope to report on industrial case studies that use triggers for synthesis.

VI. Related Work and Discussion

PSL Triggers, LTL, and GR(1). The Property Specification Language (PSL) [16], [23], an IEEE standard widely used in industrial formal verification tools, includes a triggers operator. According to several authors, the PSL triggers operator is the most commonly used PSL operator [16], [27], [51]. A variant of PSL triggers appears also in Intel's ForSpec [6]. Kupferman and Vardi [27] investigated the problem of synthesis for a subset of PSL consisting of triggers only. Since their semantics allows simultaneous occurrences, their synthesis problem is 2EXPTIME-complete, like that of LTL. Their work is theoretical and to our knowledge, has not been implemented.

In contrast, we chose a non-simultaneous semantics, where after the left hand side operand of the trigger is satisfied, it is ignored and not followed until the right hand side operand has been satisfied. This allows us to reduce triggers to GR(1) (while adding auxiliary variables) at the cost of only one exponent, enjoy GR(1)'s efficient symbolic algorithm, and take advantage of existing synthesis tools like Spectra.

Note that a reduction of the original simultaneous PSL triggers into GR(1) (and an implementation in, e.g., Spectra) is possible, albeit at a double-exponential cost [27]. We chose to avoid it by suggesting a non-simultaneous semantics. In Sect. III-C we showed that our triggers can express properties that cannot be expressed in LTL, and that they can express complicated LTL properties in a succinct manner. These theoretical results, and our evaluation, provide some evidence that non-simultaneous triggers are potentially useful.

Beyond the difference of simultaneous vs. the non-simultaneous semantics, in PSL, the triggers right operand is, by itself, any PSL formula, i.e., an RE or an LTL formula. Again, we chose to limit both operands to REs to avoid the double exponential complexity.

Interestingly, PSL includes an overlapping trigger variant, denoted $|->$, where the last state of the left side RE and the first state of the right side RE overlap. It is possible to efficiently encode a non-simultaneous version of this variant in GR(1). We deffer the formal definition as well as the implementation of this variant in Spectra to future work.

Finally, our choice to extend Spectra and not other GR(1) synthesizers, e.g., [8], [15], was motivated by Spectra's performance in comparison to these tools [18], its relatively friendly development environment, the fact that it already includes support for higher-level constructs like patterns [14], [28], and the many specifications it comes with. Still, our main contribution in this paper is relevant to GR(1) synthesis in general and is not limited to Spectra.

Triggers, Scenarios, and the Dywer et al. patterns [14]. Live sequence charts [21], modal sequence diagrams [20], assume-guarantee scenarios [35], and triggered scenarios [2], [46], can all be viewed as variants of triggers. Yet, our triggers are more general in two ways. First, the use of REs to express the left and right parts of the trigger, rather than, typically in the above listed works, only a partial order on events execution. Second, these scenario-based specification languages are mostly limited to event-based models (over an alphabet of events) and allow limited use of assertions, if any. In particular, the last two triggers in our running example as well as, e.g., Property 4 mentioned above, cannot be directly expressed using these scenario-based languages. Some of the above scenario-based formalisms have existential variants [7], [20], [47], [49]. An existential interpretation is impossible to express in LTL (and thus impossible to express in GR(1)); it requires the use of CTL. It could be that GR(1)* [5], as supported in Spectra, can be used or extended for this purpose and so we leave it for future work. We view the presentation of all these triggered scenario variants in the literature as additional evidence for the intuitiveness and potential usefulness of triggers.

Dywer et al. [14] have presented a well-known set of LTL specification patterns, to help engineers write common complex temporal properties. While the patterns are listed in a fixed catalog, triggers is a language construct allowing the specification of arbitrary properties. Moreover, on the one hand, some properties cannot be expressed with the Dwyer et al. patterns but can be easily expressed with triggers (e.g., relating to what happens starting from the beginning of the run, counting, and min/max/range of distance between assertions, all of which we list as example idiomatic triggers in Lst. 3). On the other hand, the other direction of what may be expressible using the Dwyer et al. patterns and not with triggers alone, is not interesting because, as shown in [28], 52 of the 55 patterns are expressible in GR(1). As the translation of the patterns into GR(1) is already implemented in Spectra, there is no need to use triggers to express the Dwyer et al. patterns.

Other Extensions of GR(1). While the GR(1) fragment is expressive, it is not always easy to write the intended assumptions and guarantees using its restricted syntax. Thus, authors have suggested additional, higher-level elements, which are intuitive to read and write and are still automatically reducible to GR(1), with the addition of auxiliary variables. These extensions include, e.g., past LTL operators (already in [10]), the Dwyer et al. patterns [14], [28], monitors, and bounded counters [33]. Our addition of REs and triggers to Spectra in this paper follows this direction.

Finally, while the title of [9] hints that the authors' synthesizer supports PSL, this work presents no support for REs or triggers. To our knowledge, our work is the first to efficiently use REs and triggers in reactive synthesis.

VII. Conclusion

We introduced triggers into specifications for reactive synthesis. We presented a novel encoding of REs and triggers, based on symbolic finite automata (SFA) and a novel non-simultaneous semantics, which enable an efficient translation and minimal addition of auxiliary variables to the synthesis problem. We demonstrated the use of triggers to express useful properties and showed the limited additional computational cost they incur on the synthesis problem.

To our knowledge, our work is the first to define, implement, and evaluate triggers in the context of reactive synthesis. Many authors describe the challenge of writing specifications in general and for reactive synthesis in particular, e.g., [27], [31], [37], [44]. One may expect that adding REs and triggers to the specification language will help engineers in writing specifications for synthesis. While there is still a long way to go to make reactive synthesis usable in practice, we believe that our work makes a step forward in this direction.

Future work. First, the idiomatic triggers from Lst. 3 call for the definition of reusable parametric specification patterns that involve REs and triggers. These can be added to Spectra as simple syntactic sugars or using a library (as in [28]). For example, allow one to write $gar\ never(regexp)$; as a syntactic sugar for $gar\ trig\ [\texttt{true}]^*\ regexp\ |=>\ [\texttt{false}]$;. Second, opportunities for optimization based on reuse of auxiliary variables between triggers, i.e., when the same RE appears in multiple triggers. Finally, further evaluation of triggers usefulness, in a user study to compare the ability of users to read and write various properties, with and without triggers, e.g., in terms of reading and writing correctness and time.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (grant No 101069165, SYNTACT).

References

[1] Spectra Website. https://smlab.cs.tau.ac.il/syntech/spectra/.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning from vacuously satisfiable scenario-based specifications. In *FASE*, pages 377–393, 2012.

[3] G. Amram, D. Ma'ayan, S. Maoz, O. Pistiner, and J. O. Ringert. Supporting artifact, 2023. https://doi.org/10.5281/zenodo.7568415.

[4] G. Amram, D. Ma'ayan, S. Maoz, O. Pistiner, and J. O. Ringert. Supporting materials website, 2023. https://smlab.cs.tau.ac.il/syntech/triggers/.

[5] G. Amram, S. Maoz, and O. Pistiner. GR(1)*: GR(1) specifications extended with existential guarantees. *Formal Aspects Comput.*, 33(4-5):729–761, 2021.

[6] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In J. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[7] S. Ben-David, M. Chechik, A. Gurfinkel, and S. Uchitel. CSSL: a logic for specifying conditional scenarios. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 37–47. ACM, 2011.

[8] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - A new requirements analysis tool with synthesis. In *CAV*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[9] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

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

[11] R. E. Boyatzis. *Transforming qualitative information: Thematic analysis and code development*. sage, 1998.

[12] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

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

[14] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM, 1999.

[15] R. Ehlers and V. Raman. Slugs: Extensible GR(1) synthesis. In *CAV*, volume 9780 of *LNCS*, pages 333–339. Springer, 2016.

[16] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[17] K. Etessami and T. Wilke. An until hierarchy and other applications of an ehrenfeucht-fraissé game for temporal logic. *Inf. Comput.*, 160(1-2):88–108, 2000.

[18] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for GR(1) synthesis and related algorithms. *Acta Informatica*, 57(1-2):37–79, 2020.

[19] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.

[21] D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling*, 2(2):82–107, 2003.

[22] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[23] IEEE Standards. IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL). *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pages 1–184, June 2012.

[24] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[25] U. Klein and A. Pnueli. Revisiting synthesis of GR(1) specifications. In S. Barner, I. G. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, volume 6504 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2010.

[26] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Trans. Robotics*, 25(6):1370–1381, 2009.

[27] O. Kupferman and M. Y. Vardi. Synthesis of trigger properties. In *LPAR*, volume 6355 of *LNCS*, pages 312–331. Springer, 2010.

[28] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *ESEC/FSE*, pages 96–106. ACM, 2015.

[29] S. Maoz and J. O. Ringert. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015*, volume 202 of *EPTCS*, pages 58–72, 2015.

[30] S. Maoz and J. O. Ringert. On well-separation of GR(1) specifications. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 362–372. ACM, 2016.

[31] S. Maoz and J. O. Ringert. On the software engineering challenges of applying reactive synthesis to robotics. In F. Ciccozzi, D. D. Ruscio, I. Malavolta, P. Pelliccione, and A. Wortmann, editors, *Proceedings of the 1st International Workshop on Robotics Software Engineering, RoSE@ICSE 2018, Gothenburg, Sweden, May 28, 2018*, pages 17–22. ACM, 2018.

[32] S. Maoz and J. O. Ringert. Reactive synthesis with spectra: A tutorial. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 320–321, 2021.

[33] S. Maoz and J. O. Ringert. Spectra: A specification language for reactive systems. *Software and Systems Modeling*, 2021.

[34] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for GR(1) specifications. In G. Mussbacher, J. M. Atlee, and T. Bultan, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1016–1026. IEEE / ACM, 2019.

[35] S. Maoz and Y. Sa'ar. Assume-guarantee scenarios: Semantics and synthesis. In *MODELS*, volume 7590 of *LNCS*, pages 335–351. Springer, 2012.

[36] S. Maoz and I. Shevrin. Just-in-time reactive synthesis. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 635–646. IEEE, 2020.

[37] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10):2208–2224, 2021.

[38] A. Møller. dk.brics.automaton: Package Release 1.12-1. https://www.brics.dk/automaton/, 2017.

[39] N. Ozay, U. Topcu, and R. M. Murray. Distributed power allocation for vehicle management systems. In *CDC-ECC*, pages 4841–4848. IEEE, 2011.

[40] N. Ozay, U. Topcu, R. M. Murray, and T. Wongpiromsarn. Distributed synthesis of control protocols for smart camera networks. In *ICCPS'11*, pages 45–54. IEEE Computer Society, 2011.

[41] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[42] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[43] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *POPL*, pages 179–190. ACM Press, 1989.

[44] K. Y. Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In *VSTTE*, volume 9971 of *LNCS*, pages 8–26, 2016.

[45] L. Ryzhyk and A. Walker. Developing a practical reactive synthesis tool: Experience and lessons learned. In R. Piskac and R. Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, volume 229 of *EPTCS*, pages 84–99, 2016.

[46] G. E. Sibay, V. A. Braberman, S. Uchitel, and J. Kramer. Synthesizing modal transition systems from triggered scenarios. *IEEE Trans. Software Eng.*, 39(7):975–1001, 2013.

[47] G. E. Sibay, S. Uchitel, and V. A. Braberman. Existential live sequence charts revisited. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 41–50. ACM, 2008.

[48] F. Somenzi. CUDD: CU Decision Diagram Package Release 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf, 2015.

[49] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th international conference on Software Engineering*, pages 34–43. IEEE Computer Society, 2007.

[50] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 498–507. IEEE Computer Society, 2010.

[51] Wikipedia. Property Specification Language — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Property_Specification_Language, 2023. [Online; accessed 3-February-2023].

[52] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning. In *Proce. of the 14th Int. Conf. on Hybrid Systems: Computation and Control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.