GR(1) Synthesis for LTL Specification Patterns

Shahar Maoz
School of Computer Science
Tel Aviv University, Israel

Jan Oliver Ringert
School of Computer Science
Tel Aviv University, Israel

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 software engineering practice are its very high worst-case complexity – for linear temporal logic (LTL) it is double exponential in the length of the formula, and the difficulty of writing declarative specifications using basic LTL operators. To address the first challenge, Piterman et al. have suggested the General Reactivity of Rank 1 (GR(1)) fragment of LTL, which has an efficient polynomial time symbolic synthesis algorithm. To address the second challenge, Dwyer et al. have identified 55 LTL specification patterns, which are common in industrial specifications and make writing specifications easier.

In this work we show that almost all of the 55 LTL specification patterns identified by Dwyer et al. can be expressed as assumptions and guarantees in the GR(1) fragment of LTL. Specifically, we present an automated, sound and complete translation of the patterns to the GR(1) form, which effectively results in an efficient reactive synthesis procedure for any specification that is written using the patterns.

We have validated the correctness of the catalog of GR(1) templates we have created. The work is implemented in our reactive synthesis environment. It provides positive, promising evidence, for the potential feasibility of using reactive synthesis in practice.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Languages, Design

Keywords

Linear temporal logic, synthesis, specification patterns

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786824

1. INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [35]. Rather than manually constructing a system and using model checking to verify its compliance with its specification, synthesis offers an approach where a correct implementation of the system is automatically obtained for a given specification, if such an implementation exists. In the case of reactive synthesis, an implementation is typically given as an automaton that accepts input from the environment (e.g., from sensors) and produces the system's output (e.g., on actuators). By construction the input and output assignments of every infinite run of the automaton satisfy the specification it was synthesized from.

Two of the main challenges in bringing reactive synthesis to software engineering practice are its very high worst-case complexity – for linear temporal logic (LTL) it is double exponential in the length of the formula, and the difficulty of writing declarative specifications using basic LTL operators.

To address the first challenge, Piterman et al. [7, 34] have suggested the General Reactivity of Rank 1 (GR(1)) fragment of LTL, which has an efficient polynomial time symbolic synthesis algorithm. GR(1) is a strict assume/guarantee subset of LTL, comprised of constraints for initial states θ, safety propositions ρ over the current and successor state, and justice constraints J_i (i.e., assertions about what should hold infinitely often). Intuitively, if the assumptions θ^e, ρ^e, and J_i^e are satisfied by the environment the system has to satisfy the guarantees θ^s, ρ^s, and J_i^s, i.e., valid runs satisfy

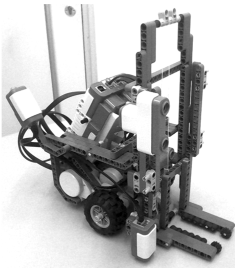
$$(\theta^e \wedge \mathtt{G}\rho^e \wedge \bigwedge_{0<i\leq j} \mathtt{GF}J_i^e) \rightarrow (\theta^s \wedge \mathtt{G}\rho^s \wedge \bigwedge_{0<i\leq k} \mathtt{GF}J_i^s).$$

GR(1) synthesis has been used in various application domains and contexts, including robotics [25], scenario-based specifications [32], aspect languages [31], and event-based behavior models [12], to name a few.

To address the second challenge, Dwyer et al. [16] have identified 55 LTL specification patterns, which are common in industrial specifications and make writing specifications easier. The patterns are organized by kind, e.g., absence or existence of properties, and are ordered by scope, e.g., globally or before. Their semantics is defined by a mapping to basic LTL formulas. An example pattern of kind existence with scope between is

p `occurs between` q `and` r

where p, q, and r are parameters of the pattern, which can be instantiated with non-temporal propositions. This pattern is



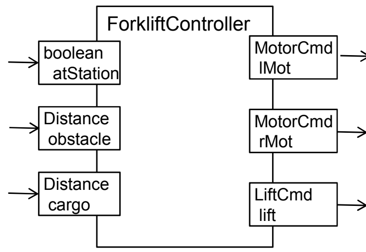


Figure 1: A forklift and its controller component `ForkliftController`

numbered P09 and its semantics expressed in LTL according to [16] is

```
[](q & !r -> (!r W (p & !r))).
```

In this work we show that almost all of the 55 LTL specification patterns identified by Dwyer et al. can be used as assumptions and guarantees in the GR(1) fragment of LTL. Specifically, we present an automated, sound and complete translation of the patterns to the GR(1) form, which effectively results in an efficient reactive synthesis procedure for any specification that is written using the patterns.

Technically, the translation starts from the LTL formula of the pattern, translates it to a minimal deterministic Büchi automaton (DBW), if one exists, and then translates the automaton to a GR(1) assumption or guarantee formula, while possibly adding auxiliary variables to the GR(1) synthesis problem. In case no DBW exists, the pattern is not supported.

Critical to the usefulness of our approach is that the costly translation of LTL to DBW is done only once for every pattern. In fact, we have already done it and saved the result as a set of templates inside our synthesis tool. This works because patterns are instantiated only with propositions (not with nested temporal operators). We further show that patterns can even be instantiated with past LTL formulas, but not with nested future temporal operators.

To summarize our contribution, our work answers the following three questions: (1) is GR(1) expressive enough to support the Dwyer et al. patterns, which are well-recognized as common in industrial specifications?, (2) can the translation be done automatically (and correctly)?, and (3) what's the extra cost of doing it (e.g., in number of auxiliary variables)?

To answer the first two questions, we have implemented and automated the translation, and our findings show that 52 of the patterns from the original work of Dwyer et al. [16] can be expressed as assumptions and guarantees in the GR(1) fragment. Thus, we have indeed embedded the results of the translation into our reactive synthesis environment. Moreover, as our translation is complete, our work shows that the remaining 3 patterns are indeed not expressible as assumptions or guarantees in the GR(1) fragment by our approach.

To answer the third question, our pattern representation in GR(1) requires at most 3 auxiliary variables per pattern instance. This gives an upper bound for the complexity of a GR(1) synthesis problem where patterns are used as assumptions or guarantees. Note that this is a very satisfying result, since based on the translation via a DBW, one could expect in the worst case an exponential number of auxiliary variables per pattern.

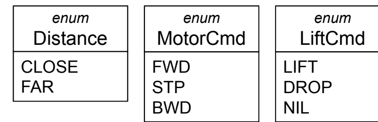


Figure 2: Enumeration datatypes of inputs and outputs of component `ForkliftController`

The remainder of this paper is structured as follows. In Sect. 2 we start with a running example. Sect. 3 provides required formal background on LTL and GR(1). Sect. 4 presents the main technical contribution of our work, i.e., the translation from LTL specification patterns to equivalent expressions in the GR(1) fragment. Sect. 5 describes the implementation and Sect. 6 presents our results. The paper concludes with a discussion in Sect. 7, related work in Sect. 8, and future work in Sect. 9.

2. RUNNING EXAMPLE

We start off with a running example, adapted from our specification of a Lego forklift, shown in Fig. 1[1], see [30]. The forklift has 3 sensors: one sensor to determine whether it is at a station and two distance sensors to detect obstacles and cargo. It also has 3 motors, to turn the left and right wheels and to lift the fork. Values read by the sensors are provided as inputs to component `ForkliftController` and its outputs are commands controlling the motors. All inputs and outputs are typed, e.g., the input `obstacle` has type `Distance`. The datatypes are defined as enumerations in Fig. 2.

An engineer specifies the behavior of the forklift controller to synthesize an implementation. A simple safety guarantee is that if the forklift detects an obstacle, both motors are stopped (see ll. 1-2 in Listing 1). Another part of the specification is the liveness guarantee to always eventually deliver cargo (expressed as `lift=DROP` in ll. 4-5).

A property more complicated to express in pure LTL is that the forklift has to leave its pick-up station between lifting and dropping cargo. The engineer expresses this guarantee for the controller in ll. 7-9 using a higher-level specification pattern instantiated with values read by the sensors.

A second engineer assists in defining assumptions on the environment of the forklift. One assumption is that going forward with both motors will lead to reaching a station unless the motors are not going forward anymore (see pattern used in ll. 11-13). It turns out that in order to satisfy this assumption an adversary environment can prevent the forklift from reaching a cargo station by presenting obstacles forcing the forklift to stop (ll. 1-2). Hence, an additional assumption for a well-behaved environment is added by the engineers: in the given setting it is reasonable to expect that between two stations, the forklift may be blocked by obstacles at most twice. This assumption is expressed using another specification pattern, shown in ll.15-17 of Listing 1.

The patterns used by the engineers in the forklift specification are LTL specification patterns as identified by Dwyer et al. [16]. Their formal semantics, in LTL, taken from [16],

[1] Note that this is a real Lego robot that we have built. We use our synthesis tool and code generation to run it.

```
                                              Specification
 1 GUARANTEE -- always stop if detect obstacle
 2   G (obstacle=CLOSE -> (lMot=STP & rMot=STP));
 3
 4 GUARANTEE -- keep on delivering cargo
 5   GF (lift=DROP);
 6
 7 GUARANTEE -- be not at station before delivery
 8   (!atStation) occurs between
 9                  (lift=LIFT) and (lift=DROP);   --P09
10
11 ASSUMPTION -- forwarding leads to station
12   Globally (lMot=FWD & rMot=FWD) leads to
13     (atStation | !(lMot=FWD & rMot=FWD));       --P26
14
15 ASSUMPTION -- at most blocked twice
16   After (!atStation) have at most two
17     (obstacle=CLOSE) until (atStation);         --P15
```

Listing 1: Excerpt of a specification for the forklift controller

is given in Eqn. 1-3 below. However, most importantly for our work, is that these LTL formulas are not written in the syntactically restricted fragment of GR(1).

$$p \texttt{ occurs between } q \texttt{ and } r := \texttt{[](q \& !r -> (!r W (p \& !r)))} \quad (1)$$

$$\texttt{Globally } p \texttt{ leads to } q := \texttt{[](p -> <>q)} \quad (2)$$

$$\texttt{After } q \texttt{ have at most two } p \texttt{ until } r := \texttt{[](q -> ((!p \& !r) U (r | ((p \& !r) U (r | ((!p \& !r) U (r | ((p \& !r) U (r | (!p W r) | []p)))))))))} \quad (3)$$

The engineers are interested in using efficient GR(1) synthesis to obtain a correct-by-construction controller for the forklift robot, one which satisfies all guarantees if all assumptions are satisfied by the environment. Can a specification including these patterns be written in a way that allows the application of GR(1) synthesis?

Our work gives a positive answer to the above question. Indeed, we show that all patterns used in the specification of the forklift in Listing 1 (and actually almost all patterns from [16], see Sect. 6), can be translated to the GR(1) fragment. The translation is fully automated and requires the addition of auxiliary Boolean variables to the synthesis problem, a total of 6 variables in our example. Although the translation time may be double exponential in the length of the formula, we have already done it, off line, so that the patterns are added to our specification language and their instantiation in a given specification takes constant time in our tool.

As an example, the GR(1) representation of pattern P09 (Eqn. 1) requires two auxiliary variables to encode a new variable `s` and is shown in Listing 2 in SMV-like syntax as used by our implementation.

3. PRELIMINARIES

We repeat some of the standard definitions of Büchi automata and linear temporal logic (LTL), e.g., as found in [7, 26]. Büchi word automata are finite automata that accept infinite words over a finite alphabet.

Definition 1. A Büchi automaton is a tuple $\mathcal{B} = (Q, \Sigma, \delta, I, F)$ where:

```
                                                         SMV
 1 VAR -- auxiliary variables: states of DBW
 2   s : {S1, S2, bot};
 3 INIT -- initial assignments: initial state
 4   s=S1;
 5 TRANS -- safety this and next state
 6  ((s=S1 & (lift =DROP | atStation | lift!=LIFT)
 7                                      & X s=S1) |
 8    (s=S1 & (lift!=DROP & !atStation & lift=LIFT)
 9                                      & X s=S2) |
10    (s=S2 & (lift!=DROP & !atStation  ) & X s=S2) |
11    (s=S2 & (lift =DROP               ) & X s=bot)|
12    (s=S2 & (lift!=DROP &  atStation  ) & X s=S1) |
13    (s=bot& (TRUE                     ) & X s=bot));
14 LTLSPEC -- justice part: accepting states
15   G F (s=S1 | s=S2);
```

Listing 2: The instance of LTL specification pattern P09 from Listing 1, ll. 8-9, written in GR(1) SMV-like syntax as used in our implementation

- Q is the set of states
- Σ is the alphabet
- $\delta : Q \times \Sigma \times Q$ is the transition relation
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of repeated states (Büchi condition)

A Büchi automaton is deterministic (a DBW) iff $|I| = 1$ and $\forall q, u : |\{q' \text{ s.t. } (q, u, q') \in \delta\}| \leq 1$. A Büchi automaton is complete iff $\forall q, u\ \exists q' : (q, u, q') \in \delta$.

An accepting run of $\mathcal{B}$ on a word $u_0 u_1.. \in \Sigma^\omega$ is a sequence $q_0 q_1..$ of states from Q such that $q_0 \in I$, $\forall i \geq 0 : (q_i, u_i, q_{i+1}) \in \delta$, and some $q \in F$ appears infinitely often in the run. The set of words accepted by $\mathcal{B}$ is denoted $L(\mathcal{B})$.

We now repeat the definition of LTL, a modal temporal logic with modalities referring to time. LTL allows engineers to express properties of executions of reactive systems. The syntax of LTL formulas is typically defined over a set of atomic propositions AP with the future temporal operators `X` (next) and `U` (until) and the past time temporal operators `Y` (previous) and `S` (since).

Definition 2. The syntax of LTL formulas over AP is $\varphi ::= \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathtt{X}\varphi \mid \varphi\mathtt{U}\varphi \mid \mathtt{Y}\varphi \mid \varphi\mathtt{S}\varphi$ for $p \in AP$.

For $\Sigma = 2^{AP}$ a computation $u = u_0 u_1.. \in \Sigma^\omega$ is a sequence where u_i is the set of atomic propositions that hold at the i-th position. For position i we use $u, i \models \varphi$ to denote that φ holds at position i, inductively defined as:

- $u, i \models p$ iff $p \in u_i$
- $u, i \models \neg\phi$ iff $u, i \not\models \phi$
- $u, i \models \varphi_1 \vee \varphi_2$ iff $u, i \models \varphi_1$ or $u, i \models \varphi_2$
- $u, i \models \mathtt{X}\varphi$ iff $u, i{+}1 \models \varphi$
- $u, i \models \varphi_1 \mathtt{U} \varphi_2$ iff $\exists k \geq i$: $u, k \models \varphi_2$ and $\forall j, i \leq j < k$: $u, j \models \varphi_1$
- $u, i \models \mathtt{Y}\varphi$ iff $u, i{-}1 \models \varphi$
- $u, i \models \varphi_1 \mathtt{S} \varphi_2$ iff $\exists k, 0 \leq k \leq i$: $u, k \models \varphi_2$ and $\forall j, k < j \leq i$: $u, j \models \varphi_1$

We denote $u, 0 \models \varphi$ by $u \models \varphi$. Additional LTL operators are defined as abbreviations of the above:

- $\mathtt{F}\varphi := \mathtt{true}\mathtt{U}\varphi$ (finally)
- $\mathtt{G}\varphi := \neg\mathtt{F}\neg\varphi$ (globally)
- $\varphi_1 \mathtt{W} \varphi_2 := (\varphi_1 \mathtt{U} \varphi_2) \vee \mathtt{G}\varphi_1$ (weak until)
- $\mathtt{H}\varphi := \neg(\mathtt{true}\mathtt{S}\neg\varphi)$ (historically)

LTL formulas can be used as specifications of reactive systems where atomic propositions are interpreted as environment (input) and system (output) variables.

An LTL specification φ is realizable if a fairness-free automaton (Büchi automaton without acceptance condition) exists such that all runs of the automaton are accepted by φ [7]. This automaton is called a controller. The goal of LTL synthesis is, given an LTL specification, to find a controller that realizes it, if such a controller exists.

GR(1) synthesis [7] handles a fragment of LTL where specifications contain assertions over initial states, safety constraints relating the current and next state, and justice goals requiring that an assertion holds infinitely many times during a computation. A GR(1) synthesis problem is defined as a game between a system player and an environment player, with the following game structure [7]:

- $\mathcal{X}$ input variables controlled by the environment
- $\mathcal{Y}$ output variables controlled by the system
- θ^e assertion over $\mathcal{X}$ characterizing initial states of the environment
- θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial states of the system
- $\rho^e(\mathcal{X} \cup \mathcal{Y}, \mathcal{X})$ transition relation of the environment
- $\rho^s(\mathcal{X} \cup \mathcal{Y}, \mathcal{X} \cup \mathcal{Y})$ transition relation of the system
- $\varphi = \mathtt{GF}J^e \rightarrow \mathtt{GF}J^s$ winning condition as implication between justice goals J^e of the environment and J^s of the system.

The game has a winning strategy for the system, i.e., the system player can always win following this strategy, iff the following LTL specification φ_G is realizable [7]:

$$\varphi_G = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathtt{G}((\mathtt{H}\rho^e) \rightarrow \rho^s)) \wedge (\theta^e \wedge \mathtt{G}\rho^e \rightarrow \varphi).$$

Specifications for GR(1) synthesis have to be expressible in the above game structure and thus do not cover the complete LTL. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis have been presented in [7, 34].

4. FROM LTL SPECIFICATION PATTERNS TO GR(1)

We are now ready to present the main technical contribution of our work, i.e., the translation from LTL specification patterns to equivalent expressions in the GR(1) fragment, if such an expression exists.

The main idea of our approach is to translate the LTL formula of a specification pattern to a DBW, if such DBW exists. We then translate the DBW to an LTL formula in the GR(1) fragment. This translation may add auxiliary variables. This approach enables us to use existing algorithms for GR(1) synthesis to solve the syntactically enriched synthesis problem of specifications containing LTL specification patterns as assumptions and guarantees.

We show that our translation, if DBWs for all patterns exist, is correct for the synthesis problem realizing the specification

$$(\theta^e \wedge \mathtt{G}\rho^e \wedge \bigwedge_{0<i\leq j} \mathtt{GF}J^e_i \wedge \bigwedge_{j<i\leq m} \psi^e_i) \rightarrow (\theta^s \wedge \mathtt{G}\rho^s \wedge \bigwedge_{0<i\leq k} \mathtt{GF}J^s_i \wedge \bigwedge_{k<i\leq n} \psi^s_i) \quad (4)$$

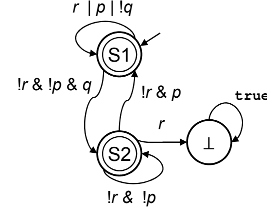


Figure 3: DBW for the LTL specification pattern P09 in Eqn. 1

where θ^e,θ^s, ρ^e, and ρ^s are in the GR(1) fragment, J^e_i are j justice assumptions, J^s_i are k justice guarantees, ψ^e_i are $m{-}j$ LTL specification patterns used as assumptions, and ψ^s_i are $n{-}k$ patterns used as guarantees.

In our running example, the specification shown in Listing 1 is represented in the above format where the first guarantee (l. 2) is a conjunct of ρ^s, the second guarantee (l. 5) is a J^s_i, the pattern of the third guarantee (ll. 8-9) is a ψ^s_i, and the two patterns used as assumptions (ll. 11-17) appear as two ψ^e_i.

We show the translation, prove its correctness, and discuss the time complexity for solving the synthesis problem using existing algorithms, which depends on the auxiliary variables added during our translation.

4.1 Translation

Our translation of the specification given in Eqn. 4 combines two constructions from [7]: (a) for solving the GR(1) implication specification, i.e., Eqn. 4 without LTL specification patterns ψ^e_i and ψ^s_i, and (b) for solving the implication specification between DBWs with additional variables representing their states. Construction (a) adds two Boolean monitor variables sf^e (observing historical satisfaction of θ^e and ρ^e) and sf^s (observing θ^s and ρ^s), the justice assumption $\mathtt{GF}\ sf^e$, and the justice guarantee $\mathtt{GF}\ sf^s$ [7, Sect. 5.1]. Intuitively this construction turns safety violations into justice violations. We apply (a) directly to Eqn. 4 without the patterns ψ^e_i and ψ^s_i, which are handled by (b). Construction (b) adds variables representing the states of each complete DBW, its initial states, transitions, and acceptance condition to the game construction [7, Sect. 5.3].

To apply construction (b) we first translate each LTL specification pattern into a corresponding DBW (if such a DBW exists for the pattern), and then transform this DBW into the initial, safety, and justice constraints for the construction of the GR(1) game. The problem of translating an LTL formula into a DBW (if one exists) is well-studied and algorithms and tools are available [2, 15]. As an example the LTL specification pattern in Eqn. 1 has a corresponding DBW with three states shown in Fig. 3. We now show how to translate a DBW into the parts required for the GR(1) construction.

4.1.1 DBW2GR1 Construction

Given a DBW $\mathcal{B} = (Q, \Sigma, \delta, \{q_I\}, F)$ with alphabet $\Sigma = 2^{\mathcal{X}\cup\mathcal{Y}}$ we obtain an LTL formula φ in the GR(1) fragment over the alphabet $\Sigma' = 2^{\mathcal{X}\cup\mathcal{Y}\cup V}$ where $\mathcal{X}$ and $\mathcal{Y}$ are sets of Boolean input and output variables resp. and V is a set of auxiliary Boolean variables added by the translation to encode the states of the DBW. Our translation creates φ using three subformulas where θ encodes the initial state,

```
SMV
1  VAR -- auxiliary variables: states of DBW
2    s : {S1, S2, bot};
3  INIT -- initial assignments: initial state
4    s=S1;
5  TRANS -- safety this and next state
6    ((s=S1  & ( r |  p | !q) & X s=S1) |
7     (s=S1  & (!r & !p &  q) & X s=S2) |
8     (s=S2  & (!r & !p      ) & X s=S2) |
9     (s=S2  & ( r           ) & X s=bot)|
10    (s=S2  & (!r &  p      ) & X s=S1) |
11    (s=bot & (TRUE         ) & X s=bot));
12 LTLSPEC -- justice part: accepting states
13   G F (s=S1 | s=S2);
```

Listing 3: Output of DBW2GR1, for the DBW of the LTL specification pattern P09 in Eqn. 1, shown in Fig. 3

ρ encodes the transition relation as safety properties over the current and next state, and J encodes the acceptance condition of the DBW as a justice constraint. The injective function $\mathtt{map}_V : Q \to 2^V$ maps states of the DBW to valuations of the auxiliary variables introduced by the translation. Formally:

VAR V is set of $k = \lceil log_2|Q|\rceil$ fresh Boolean variables

INIT

$$\theta := \mathtt{map}_V(q_I)$$

TRANS

$$\rho := \bigvee_{(q,u,q')\in\delta} (\mathtt{map}_V(q) \wedge u \wedge \mathtt{X}\ \mathtt{map}_V(q'))$$

JUSTICE

$$J := \bigvee_{q\in F} \mathtt{map}_V(q)$$

The LTL formula that characterizes the acceptance of the DBW $\mathcal{B}_i^\alpha$ for $\alpha \in \{e, s\}$ resulting from the pattern ψ_i^α, is the conjunction $\varphi_i^\alpha = \theta_i^\alpha \wedge \mathtt{G}\rho_i^\alpha \wedge \mathtt{GF}J_i^\alpha$ obtained from the translation above. Note that this translation, from DBW to GR(1), is linear in the size of the DBW.

In our running example, the DBW shown in Fig. 3, which represents the pattern P09, is translated to a formula with the structure of φ_i^α in the GR(1) fragment, as shown in Listing 3 in the SMV-like syntax used by our implementation. The first part of the translation creates a set of auxiliary variables to represent the states of the DBW. In Listing 3 we use a single variable `s` (l. 2) for better readability instead of two Boolean variables encoding the values of `s`. The translation part **INIT** sets the initial assignment of the auxiliary variable (θ encoded in l. 4) to represent the initial state of the DBW from Fig. 3. The part **TRANS** lists the transition relation δ as constraints over current state, current input, and next state ($\mathtt{G}\rho$ encoded in ll.6-11). Finally the acceptance of the DBW is translated in part **JUSTICE** to the disjunction of always eventually visiting one of the accepting states ($\mathtt{GF}J$ encoded in l. 13).

Note that the result of DBW2GR1 as shown in Listing 3 is a template where variables `p`,`q`, and `r` can be instantiated with non-temporal assertions. This is the same instantiation mechanism used for the original LTL patterns.

4.1.2 Game Construction

The game for GR(1) synthesis based on this translation, combining constructions (a) and (b), has the following structure for specifications as shown in Eqn. 4:

- $\mathcal{X}' = \mathcal{X}$
- $\mathcal{Y}' = \mathcal{Y} \cup \{sf^e, sf^s\} \cup \bigcup_{j<i\leq m} V_i^e \cup \bigcup_{k<i\leq n} V_i^s$
- $\theta^e = \mathtt{true}$
- $\theta^s = (\theta^e \leftrightarrow sf^e) \wedge (\theta^s \leftrightarrow sf^s) \wedge \bigwedge_{j<i\leq m} \theta_i^e \wedge \bigwedge_{k<i\leq n} \theta_i^s$
- $\rho^e = \mathtt{true}$
- $\rho^s = ((\rho^e \wedge sf^e) \leftrightarrow \mathtt{X}sf^e) \wedge ((\rho^s \wedge sf^s) \leftrightarrow \mathtt{X}sf^s) \wedge \bigwedge_{j<i\leq m} \rho_i^e \wedge \bigwedge_{k<i\leq n} \rho_i^s$
- $\varphi = (\mathtt{GF}sf^e \wedge \mathtt{GF}J^e \wedge \bigwedge_{j<i\leq m} \mathtt{GF}J_i^e) \to (\mathtt{GF}sf^s \wedge \mathtt{GF}J^s \wedge \bigwedge_{k<i\leq n} \mathtt{GF}J_i^s)$.

We use the GR(1) synthesis algorithm of [7] to solve this game. Theorem 1 (below) states that a winning strategy for the system player in the above game indeed implements the specification in Eqn. 4, and that such a strategy is found if it exists.

Note that for each specification pattern, the translation above depends on the existence of a DBW for the pattern. Some LTL formulas cannot be expressed as a DBW [26] and our translation thus handles only a subset of LTL. As we show in Sect. 6, almost all LTL patterns from [16] do have a corresponding DBW, and hence, are supported by our translation.

4.2 Translation Correctness

For two words $w \in \Sigma^\omega$ and $v \in (2^V)^\omega$ we denote as $w \oplus v \in (\Sigma \cup 2^V)^\omega$ the word where $(w \oplus v)_i = w_i \cup v_i$.

Lemma 1 (DBW2GR1 is correct)**.** Given a complete DBW $\mathcal{B}$, the LTL formula $\varphi = \theta \wedge \mathtt{G}\rho \wedge \mathtt{GF}J$ resulting from the translation in Sect. 4.1 is in the GR(1) fragment and satisfies: $\forall w \in \Sigma^\omega : w \in L(\mathcal{B}) \leftrightarrow \exists v \in (2^V)^\omega : w \oplus v \models \varphi$.

Proof. By construction θ and J have no temporal operators and ρ uses only the next operator, so φ is in the supported GR(1) fragment. We show bidirectional acceptance:

"$\to$": $\forall w = w_0w_1.. \in \Sigma^\omega : w \in L(\mathcal{B})$ the DBW has by definition of determinism a unique run $q = q_0q_1.. \in Q^\omega$ with $q_I = q_0$ and $\forall i : (q_i, w_i, q_{i+1}) \in \delta$. For $v \in (2^V)^\omega$ with $\forall i : v_i = \mathtt{map}_V(q_i)$ we have: $w \oplus v \models \theta$ by construction and definition of v_0. Analogously, $\forall i : (q_i, w_i, q_{i+1}) \in \delta$ yields $(w \oplus v), i \models \rho$, thus $(w \oplus v) \models \mathtt{G}\rho$. Since some $q_i \in F$ appears infinitely often in the run q it is encoded infinitely often in v and thus $w \oplus v \models \mathtt{GF}J$.

"$\leftarrow$": Given $\forall w = w_0w_1.. \in \Sigma^\omega : \exists v \in (2^V)^\omega : w \oplus v \models \varphi$: from $w \oplus v \models \theta$ we know that $\mathtt{map}_V^{-1}(v_0) = q_I$ and from $\forall i : (w \oplus v), i \models \rho$ we know that v must encode an infinite run q of $\mathcal{B}$ with $\forall i : q_i = \mathtt{map}_v^{-1}(v_i)$ and $(q_i, w_i, q_{i+1}) \in \delta$. Due to determinism of $\mathcal{B}$ and injectiveness of $\mathtt{map}_V$ the run is unique and exists iff $(w \oplus v) \models \mathtt{G}\rho$. Since the run is unique from $(w \oplus v) \models \mathtt{GF}J$ we know that a state in F of $\mathcal{B}$ is visited infinitely often and $\mathcal{B}$ accepts w. □

Theorem 1. Given a specification in the form of Eqn. 4 where ψ_i^e and ψ_i^s can be translated into DBWs our construction synthesizes a controller implementing the specification, if one exists.

Proof. The correctness of construction (a) is shown in [7, Thm. 5]. The correctness of construction (b) is shown in

[7, Thm. 7]. The proof of (b) requires completeness and determinism of the Büchi automata. The DBW we use are complete and in Lemma 1 we showed that our translation from Sect. 4.1 correctly represents the DBW in the GR(1) fragment. □

4.3 Complexity Analysis

It is known that the worst-case time complexity for the synthesis of a controller for an open reactive system from a general LTL specification φ is double exponential in the length of the input formula, i.e., in $O(2^{2^{|\varphi|}})$ [35]. In contrast, the worst-case time complexity of synthesizing a controller for a GR(1) specification is in $O(nmN^2)$ where n and m are the number of justice goals of the environment and system players respectively, and N is the size of the state space.

Since we use GR(1) synthesis, the algorithm's complexity is not changed. However, our construction increases the state space for the algorithm. Instead of a state space N of size $O(2^{|\mathcal{X}|+|\mathcal{Y}|})$, where $\mathcal{X}$ and $\mathcal{Y}$ are the sets of Boolean environment and system variables, we have a state space of size $O(2^{|\mathcal{X}|+|\mathcal{Y}|+|V|})$, where $|V|$ is the total number of additional, auxiliary variables.

Each supported pattern may add a justice goal to either m or n, and $\lceil log_2|Q|\rceil$ variables where Q is the set of states of the DBW corresponding to the pattern. In general, the size of a DBW corresponding to an LTL formula, if one exists, may be double exponential in the length of the formula [28]. Thus, the number of variables added per pattern, is in the worst case exponential in the length of the pattern.

Most importantly however, this high worst-case complexity does not apply to the LTL patterns of [16]. Our results, presented in Sect. 6, show that 52 of these patterns have a corresponding DBW with at most 8 states (the remaining 3 patterns do not have a corresponding DBW at all), so only at most 3 variables need to be added for each of these patterns.

5. IMPLEMENTATION

5.1 Overview

We have implemented the translation `DBW2GR1` and set up a toolchain to (1) check for all patterns from [16] whether they can be supported by our approach and at what cost in terms of auxiliary variables, and (2) provide fully automated support of patterns in our GR(1) synthesis environment.

The cost of supporting an LTL specification pattern ψ in GR(1) synthesis, following the translation in Sect. 4.1, depends on the size of the DBW found for ψ. Given a pattern we are thus interested in a smallest DBW representing it. However, the smallest DBW is not necessarily unique and DBW minimization is NP-hard [38].

Given the high cost of obtaining a GR(1) representation of an LTL pattern we decided to do this computation *offline*, i.e., *independent* of the instantiation of a pattern and the specification it is used in. This works because patterns are only instantiated with non-temporal propositions (in Sect. 7.3 we second this limitation and show that they can be instantiated also with past LTL formulas), and their GR(1) representations can thus be used as templates. Thus, the output of applying our toolchain to the 55 patterns from [16] is a catalog of GR(1) templates, available from [40], together with the DBW2GR1 tool. When preparing a pattern-based

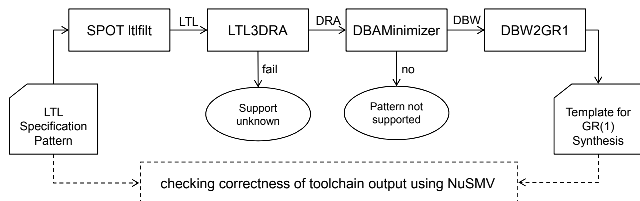


Figure 4: The toolchain we have used, per pattern, to create the GR(1) templates (the dashed parts are used for validation, see Sect. 6.2)

specification for synthesis, we do not need to recompute the translation of each pattern to an equivalent GR(1) formula but only to look up the GR(1) template that corresponds to the pattern in the catalog.

Below we give additional details about the toolchain we have used to create the catalog.

5.2 Toolchain

To decide whether an LTL specification pattern can be supported and to find the minimal DBW we use a combination of tools. Fig. 4 shows our toolchain. The input is an LTL specification pattern and the output is either, a template for synthesis in GR(1) format, a negative result that no such template exists, or an indeterminate result that the existence of such a template is unknown.

We start with the tool `ltlfilt` of `SPOT` [15], to syntactically preprocess the LTL formula of the pattern to a format readable by the next tool, `LTL3DRA`. We use `LTL3DRA` [2] to compute a deterministic Rabin automaton (DRW) from an LTL formula. `LTL3DRA` is a competitor of `LTL2DSTAR` [23] that often yields smaller DRW for LTL formulas, e.g., when trying the two tools, we saw that a DRW for pattern P45 found by `LTL2DSTAR` (version 0.5.1 with `LTL3BA` [3]) has 2184 states while a DRW for the same pattern found by `LTL3DRA` has 40 states. Both tools rely on heuristics to create smaller DRWs. Next, given a DRW, we use `DBAMinimizer` [17] to determine whether a corresponding DBW exists and to find a minimal one. `DBAMinimizer` uses a SAT solver to find a minimal DBW. Finally, we feed the minimal DBW found by `DBAMinimizer` to our tool `DBW2GR1`, to obtain a template for instantiating LTL specification patterns in GR(1) specifications as described in Sect. 4.1.

6. RESULTS AND OBSERVATIONS

Table 1 presents our main result: **52 of the 55 patterns of [16] are supported by our approach. The cost per supported pattern instance is at most 3 auxiliary variables. The remaining 3 patterns P23, P50, and P55, do not have a corresponding DBW.**

The table contains all LTL patterns from [16] grouped by kind, e.g., absence or existence, and ordered by scope (second column), e.g., globally or before, following the classification presented in [16]. For each pattern we report the length (third column) of the LTL formula that defines the semantics of the pattern (number of characters including spaces, e.g., P01 defined as `[](!p)` has length 6). The fourth column of the table reports the number of states of a minimal DBW for the pattern if one exists and `n.a.` otherwise; the fifth indicates whether the minimal DBW has a sink state (a complete DBW without a sink state expresses a pure liveness

Table 1: The 55 LTL specification patterns from [16], with LTL formula length, number of states of minimal DBW, whether DBW has a sink, and number of auxiliary variables in GR(1) template.

P#	Scope	LTL $\|\psi\|$	DBW min $\|Q\|$	DBW Sink?	GR(1) $\|V\|$
Absence: p is false					
P01	Globally	6	2	1	1
P02	Before r	15	4	1	2
P03	After q	15	3	1	2
P04	Between q and r	30	4	1	2
P05	After q until r	22	3	1	2
Existence: p becomes true					
P06	Globally	5	2	0	1
P07	Before r	13	3	1	2
P08	After q	20	3	0	2
P09	Between q and r	29	3	1	2
P10	After q until r	29	3	1	2
Bounded Existence: p-states occur at most 2 times					
P11	Globally	30	6	1	3
P12	Before r	93	8	1	3
P13	After q	50	7	1	3
P14	Between q and r	103	8	1	3
P15	After q until r	101	7	1	3
Universality: p is true					
P16	Globally	5	2	1	1
P17	Before r	14	4	1	2
P18	After q	14	3	1	2
P19	Between q and r	29	4	1	2
P20	After q until r	21	3	1	2
Precedence: s precedes p					
P21	Globally	6	3	1	2
P22	Before r	21	4	1	2
P23	After q	23	n.a.	n.a.	n.a.
P24	Between q and r	36	4	1	2
P25	After q until r	28	3	1	2
Response: s responds to p					
P26	Globally	12	2	0	1
P27	Before r	33	4	1	2
P28	After q	21	3	0	2
P29	Between q and r	48	4	1	2
P30	After q until r	42	4	1	2
Precedence Chain: s, t precedes p					
P31	Globally	34	4	1	2
P32	Before r	40	5	1	3
P33	After q	56	5	1	3
P34	Between q and r	50	5	1	3
P35	After q until r	51	4	1	2
Precedence Chain: p precedes s, t					
P36	Globally	28	4	1	2
P37	Before r	51	5	1	3
P38	After q	54	5	1	3
P39	Between q and r	61	5	1	3
P40	After q until r	69	4	1	2
Response Chain: p responds to s, t					
P41	Globally	33	4	1	2
P42	Before r	48	6	0	3
P43	After q	46	5	0	3
P44	Between q and r	59	8	0	3
P45	After q until r	96	6	0	3
Response Chain: s, t responds to p					
P46	Globally	22	5	0	3
P47	Before r	45	5	1	3
P48	After q	31	5	1	3
P49	Between q and r	56	5	1	3
P50	After q until r	74	n.a.	n.a.	n.a.
Constrained Chain: s, t without z responds to p					
P51	Globally	32	6	1	3
P52	Before r	57	5	1	3
P53	After q	40	5	1	3
P54	Between q and r	68	5	1	3
P55	After q until r	96	n.a.	n.a.	n.a.

property). Finally, the last column reports on the number of auxiliary variables necessary to support the pattern following our DBW2GR1 construction described in Sect. 4.

Some examples of the LTL patterns and the GR(1) templates we have computed for them are detailed in Table 2. Based on this table, our synthesis tool accepts specifications with assumptions and guarantees written using patterns, as shown in the second column of the table, and automatically instantiate them in a form that allows GR(1) synthesis based on the template in the fourth column. List. 1 is an example of a specification handled by our environment.

6.1 Unsupported Patterns

The three LTL patterns where a DBW does not exist are

- the *precedence* pattern P23 with scope *after q* and LTL semantics `[]!q | <>(q & (!p W s))`,
- the *response chain* pattern P50 with scope *after q until r* and LTL semantics `[] (q -> (p -> (!r U (s & !r & X(!r U t)))) U (r | [] (p -> (s & X<> t))))`, and
- the *constrained chain* pattern P55 with scope *after q until r* and LTL semantics `[] (q -> (p -> (!r U (s & !r & !z & X((!r & !z) U t)))) U (r || [] (p -> (s & !z & X(!z U t)))))`.

It is important to note, however, that these three patterns, which we cannot support, are among the least frequent patterns according to the survey reported in [16]. Specifically, out of 555 pattern instances examined in [16], only one was in the form of P23. The other two patterns, P50 and P55, are listed in [16] in order to make the matrix of pattern/scope combinations complete, but in the survey of [16], were not found at all. This further strengthens the positive, promising nature of our results.

6.2 Validation

The correctness of the GR(1) templates resulting from LTL patterns relies on a toolchain with four tools of which some are prototypical implementations and proofs of concept. These may have bugs or we might have used them incorrectly. It is thus advised to validate the correctness of the generated GR(1) templates. To address this, we have implemented a correctness check that takes the original LTL pattern and the generated GR(1) template, augments it for verification, and automatically checks correctness using the model checker `NuSMV` [10].

We generate two checks for each generated GR(1) template. The first check asserts the equivalence of the satisfaction of the generated `JUSTICE` formula J and the satisfaction of the original LTL semantics of the pattern ψ with GR(1) representation $\theta \wedge \mathtt{G}\rho \wedge \mathtt{GF}J$. By construction of our translation (see, e.g., List. 3), this check relies on the assignments to auxiliary variables in the blocks `INIT` (encoding θ) and `TRANS` (encoding $\mathtt{G}\rho$). The construction of [7] for supporting a DBW in GR(1) specifications requires that the DBW is complete, i.e., that the automaton has an enabled transition for every source state and input. We thus generate a second check, which asserts that all combinations of values assigned to variables representing the parameters of the pattern are accepted in every state of the automaton defined in the generated GR(1) template.

Initially, our validation reported incorrect templates for three out of the 52 supported patterns and their 41 sup-

ported negations (see Sect. 7.4), specifically for pattern P51 and the negations of P15 and P42. Inspection revealed incorrect translation by LTL3DRA. For at least 3 out of the 110 provided LTL formulas, LTL3DRA produced an incorrect DRW! This issue appears in version 0.1.1 (2013-09-09), used for our experiments, and persists in version 0.2.1 (2015-02-08) of LTL3DRA.[2] To bypass this bug in LTL3DRA, we reran all non-validated LTL to DRW translations using LTL2DSTAR [23], for the 3 incorrect DRWs and the 14 cases where an equivalent DBW could not be found. This second experiment confirmed the non-existence of a DBW in all cases established using LTL3DRA and produced correct DRWs, so we were finally able to successfully validate our GR(1) templates for the three remaining cases.

Thus, we have successfully validated the final output of our toolchain for all templates produced and embedded in our tool. The generated templates and checks used for verification are available for inspection and reproduction from [40].

6.3 Threats to Validity

We briefly discuss threats to the internal and external validity of our results.

First, the computation of the results that we present in Table 1 relies on a toolchain with four tools of which some are prototypical implementations and proofs of concept. These may have bugs or we might have used them incorrectly. To mitigate this and validate the correctness of our results and generated catalog of GR(1) templates, we validated our results as described in Sect. 6.2 above.

Second, the origin of the specification patterns of Dwyer et al. may be viewed as another threat to the validity of our approach. These patterns were extracted from industrial and academic specifications used for model checking, not for synthesis. It may be the case that specifications written for synthesis have (or should better have) different characteristics, or use other patterns. We are unaware of any comparable studies for synthesis or studies analyzing the difference between specifications for verification and for synthesis. Thus, we believe, using the Dwyer et al. patterns in our context is reasonable.

7. DISCUSSION AND EXTENSIONS

We now discuss several important features as well as extensions of our work.

7.1 Completeness

We are interested in the completeness of results provided by our toolchain, i.e., if a DBW for an LTL pattern exists, is it found by our toolchain?

The first tool in our toolchain, SPOT, performs purely syntactic preprocessing and preserves completeness.

While a DRW can express any LTL formula [26], the second tool in our toolchain, LTL3DRA, only supports the translation of a fragment of LTL to DRW [2]. Thus, in theory, the second step is incomplete: if the translation fails it is not known whether the pattern can be supported or not (see Fig. 4). However, as our results show (see Sect. 6), for all patterns from [16] this case never happened; LTL3DRA always produced a DRW.

In the third step in the toolchain, if a DBW exists for the given DRW, it is found by DBAMinimizer. In case DBAMinimizer gives a negative result, no DBW exists and the pattern cannot be supported. In our results, this negative result was returned for only three of the patterns, as presented in Sect. 6.

Finally, our own tool, DBW2GR1, always computes a GR(1) template for a DBW. Thus, overall, although the toolchain is theoretically incomplete (due to the second tool LTL3DRA), on our data of 55 patterns it was complete.

Two more remarks are of interest. First, to address the potential incompleteness of the analyses by LTL3DRA, e.g., if using the toolchain with additional LTL formulas as input, one can use general LTL synthesis tools such as Lily [21] or Acacia+ [8] to synthesize a DBW directly. This DBW can again be minimized using DBAMinimizer. In practice, we did not need this alternative toolchain, because for all the patterns from [16], our toolchain provided a definite result.

Second, from a practical point of view, if a DBW is found, it might be too big for efficient analysis by DBAMinimizer. We have encountered this problem for patterns P49 and P54, where the SAT formula generated by DBAMinimizer in DIMACS CNF format was exceeding a size of 110GB. To solve this problem, we modified the search strategy and DBW reconstruction of DBAMinimizer to handle these cases. Technically the modification allowed us to guess and check a small size DBW resulting in smaller SAT problems, small enough for efficient analysis by DBAMinimizer.

7.2 Generalized Rabin(1) Synthesis and Remaining Patterns

Ehlers [18] has extended GR(1) synthesis, where assumptions and guarantees can be expressed as DBWs, to Generalized Rabin(1) synthesis, where assumptions and guarantees can be expressed as DRWs with one acceptance pair. DRWs with one acceptance pair are more expressive than DBWs. We now discuss how our results from Sect. 6 directly transfer to GRabin(1) synthesis.

The complexity of GRabin(1) synthesis depends on the size of the DRWs used as assumptions and guarantees. For 52 of the 55 patterns from [16] we have computed minimal DBWs, which directly translate to DRWs of the same structure and size by adapting their acceptance condition [26]. DRWs are DBW-type [27], i.e., given a DRW for language L, if a DBW for L exists, the DBW has the same structure as the DRW. Thus, a minimal DBW has the size of a minimal DRW accepting the same language since the existence of smaller DRW would contradict minimality of the DBW. Thus, for the 52 supported patterns our approach provides not only minimal DBWs but also minimal DRWs for GRabin(1) synthesis.

7.3 Incorporating the Past

In [7] it was shown how to incorporate past LTL formulas into the GR(1) fragment. Specifically, this is done by translating each past LTL formula into a deterministic temporal tester [7, Sect. 5.2]. This requires the addition of auxiliary variables to encode the states of the temporal tester and its acceptance, i.e., satisfaction of the past LTL formula. The acceptance expression is a non-temporal formula that replaces the past LTL subformula. The approach is compo-

[2]A formula outside the LTL fragment supported by LTL3DRA (see [2]) might result in an incorrect DRW. Starting from version 0.2.2 (2015-06-15) the fragment is checked.

Table 2: Patterns P09, P15, and P26 used in the forklift specification shown in Listing 1, their LTL semantics (from [16]), and the corresponding GR(1) templates we have generated for them.

Pattern	English (kind and scope)	LTL semantics ψ	GR(1) template
P09	p occurs between q and r	`[](q & !r -> (!r W (p & !r)))`	$\theta :=$ `s=S1` $\rho :=$ `((s=S1 & ( r | p | !q) & X s=S1) | (s=S1 & (!r & !p & q) & X s=S2) | (s=S2 & (!r & !p) & X s=S2) | (s=S2 & (r) & X s=bot)| (s=S2 & (!r & p) & X s=S1) | (s=bot & (TRUE) & X s=bot))` $J :=$ `(s=S1 | s=S2)`
P15	After q have at most two p until r	`[](q -> ((!p & !r) U (r | ((p & !r) U (r | ((!p & !r) U (r | ((p & !r) U (r | (!p W r) | []p)))))))))`	$\theta :=$ `s=S1` $\rho :=$ `((s=S1 & (!q & !r | r) & X s=S1) | (s=S1 & (q & !p & !r) & X s=S2) | (s=S1 & (q & p & !r) & X s=S3) | (s=S2 & (r) & X s=S1) | (s=S2 & (!p & !r) & X s=S2) | (s=S2 & (p & !r) & X s=S3) | (s=S3 & (r) & X s=S1) | (s=S3 & (p & !r) & X s=S3) | (s=S3 & (!p & !r) & X s=S4) | (s=S4 & (r) & X s=S1) | (s=S4 & (!p & !r) & X s=S4) | (s=S4 & (p & !r) & X s=S5) | (s=S5 & (r) & X s=S1) | (s=S5 & (p & !r) & X s=S5) | (s=S5 & (!p & !r) & X s=S6) | (s=S6 & (r) & X s=S1) | (s=S6 & (!p & !r) & X s=S6) | (s=S6 & (p & !r) & X s=bot) | (s=bot & (TRUE) & X s=bot))` $J :=$ `(s=S1 | s=S2 | s=S3 | s=S4 | s=S5 | s=S6)`
P26	Globally p leads to q	`[](p -> <>q)`	$\theta :=$ `s=S1` $\rho :=$ `((s=S1 & (p & q | !p) & X s=S1) | (s=S1 & (p & !q) & X s=S2) | (s=S2 & !q & X s=S2) | (s=S2 & q & X s=S1))` $J :=$ `s=S1`

sitional as long as only past LTL operators are nested within a GR(1) formula.

It is important to note that the incorporation of the past extends into our support for patterns. Specifically, one may use past LTL formulas within patterns. As an example, consider a specification that says that the forklift has loaded cargo when it leaves a station and does not drop it until it arrives at a station. Our specification expresses this guarantee using pattern P20, instantiated in the first parameter with the past LTL formula `lift!=DROP S lift=LIFT`:

```
Globally (lift!=DROP S lift=LIFT) after
          (!atStation) until (atStation) --P20
```

The past subformula is satisfied iff cargo has not been dropped since it was last lifted. Technically, our construction shown in Sect. 4.1 is extended to support past by translating past formulas to temporal testers and then instantiating the pattern of each template with the expression denoting the acceptance of the tester. We omit the details from this version of the paper.

7.4 Pattern Negations and Boolean Combinations

Our translation goes through a DBW, but DBWs are not closed under complement [26], i.e., the existence of a DBW for pattern ψ gives no information about the existence of a DBW for its negation $\neg\psi$. Thus, it is interesting to check whether these DBWs exist and whether our framework may support the negation of patterns.

We examined the negation of all 55 patterns using our toolchain and found that 41 negated patterns have a DBW and thus a GR(1) template representation. The maximal size of the minimal DBW for all pattern negations is again 8 states and so at most 3 auxiliary variables are required to support a negated pattern instantiation. Pattern P23, without a corresponding DBW, is supported in its negated form. For the two other unsupported patterns, P50 and P55, the negation does not have a corresponding DBW. So, in our synthesis tool, we support 41 pattern negations.

Finally, as DBWs are closed not only under intersection (conjunction) but also under union (disjunction), our work opens the way to support specifications that include intersections and unions over the 52 supported patterns and 41 supported pattern negations. We omit the details on how this can be done from this version of the paper.

Conjunction, disjunction, and negation of patterns allow for much flexibility and expressiveness in writing the specification on the way to the symbolic GR(1) synthesis.

As an example consider an extended forklift with an emergency off switch and the conditional guarantee to always eventually get to a station if the emergency off switch is never pressed between stations. The absence of the property `emgOff` between stations can be expressed using pattern P04, and the above guarantee can be written as an implication between an instance of pattern P04 and pattern P26:

```
(emgOff) never occurs between
      (!atStation) and (atStation) --P04
                  IMPLIES
Globally (!atStation) leads to (atStation) --P26
```

Note that the implication is supported by our approach because it translates to a disjunction of the negation of pattern P04, which has a corresponding GR(1) template, and pattern P26.

8. RELATED WORK

Specification patterns aim to assist engineers in the difficult task of formally writing a specification. The work of Dwyer et al. [16] on temporal property specifications, which we relate to in this paper, is the most well-known work in this area. The patterns have been used and extended in many tools and contexts, including, e.g., property elucidation and natural language interfaces in Smith et al. work [39], runtime verification in Bauer et al. work [4], and OCL in Dou et al. work [13], to list a few. Related and extended patterns have been investigated and proposed for service-based applications by Bianculli et al. [5], for real-time specifications by Konrad and Cheng [24], and for probabilistic specifications by Grunske [20]. A comprehensive framework for all of these patterns have been recently suggested by Autili et al. [1].

Several LTL synthesis tools were presented in recent years, e.g., Lily [21], Anzu [22], RATSY [6], Unbeast [19], and Acacia+ [8]. Some of these tools handle general LTL specifications while others focus on the GR(1) fragment. Our own implementation of GR(1) synthesis is written on top of JTLV [36], but the contribution of our present work does not depend on a specific implementation of GR(1) synthesis. Some of these tools support a combination of LTL formulas and automata as input. To the best of our knowledge, none of these synthesis tools and the works using them supports high-level specification patterns as input.

GR(1) synthesis has been used and extended in different contexts and for different application domains, including robotics [25], scenario-based specifications [32], aspect languages [31], and event-based behavior models [12], to name a few.

Many works, including some by the first listed author, have presented case studies based on GR(1) or dealt with unrealizability in the context of GR(1), e.g., [11, 22, 33, 37] and some of the works listed in the previous paragraph. All these works claimed that GR(1) is expressive enough to support most specifications written in practice. To the best of our knowledge, our work is the first to examine this claim against a concrete list of well-known patterns and thus to strengthen it with evidence and practically integrate these patterns into a synthesis tool.

9. CONCLUSION AND FUTURE WORK

In this paper we have showed that almost all of the LTL specification patterns of Dwyer et al. [16] can be used as assumptions and guarantees in GR(1) specifications, which have an efficient polynomial symbolic synthesis algorithm. We have automated the process of translating the patterns into corresponding GR(1) templates, and have integrated them into our synthesis tool. We further proved that the translation is correct, showed that it is complete for the 55 patterns of Dwyer et al., and validated its results using model-checking. The work provides evidence for the strength of the GR(1) fragment.

We consider the following future work directions. First, we plan to systematically examine possible support for additional specification patterns on top of the GR(1) fragment, by reusing our toolchain and template generator and applying them to other forms of LTL specifications. These include, for example, extensions of the Dwyer et al. patterns, counting patterns ("after q, p becomes true in at most / exactly / at least k steps / occurrences of r"), trigger patterns (which use regular expressions, as defined in Kupferman and Vardi's trigger logic [29]), different variants of scenarios [32], etc.

Furthermore, many recent works in the area of synthesis deal with quantitative and probabilistic specifications, see, e.g., [9, 14]. We plan to investigate if and how can quantitative and probabilistic patterns, as identified, e.g., in [20, 24], be supported within the symbolic algorithm of GR(1).

Our work is part of a larger project on bridging the gap between the theory and algorithms of reactive synthesis on the one hand and software engineering practice on the other. In this project, we are building engineer-friendly tools around reactive synthesis, for example to identify and fix unrealizability (as in, e.g., [11, 33]) and to provide two-way traceability between the assumptions and guarantees in the specification and the states and transitions of the synthesized implementation. Both problems, dealing with unrealizability and providing traceability, become more challenging in the presence of patterns in the language used for specification.

The DBW2GR1 tool, the generated patterns catalog, and the means to validate its correctness using model-checking, all available from [40], have been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. We thank Dafna Sadeh for her help with the implementation of DBW2GR1. Jan O. Ringert acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

11. REFERENCES

[1] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Trans. Softw. Eng.*, 2015.

[2] T. Babiak, F. Blahoudek, M. Kretínský, and J. Strejcek. Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment. In *ATVA*, volume 8172 of *LNCS*, pages 24–39. Springer, 2013.

[3] T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. In *TACAS*, volume 7214 of *LNCS*, pages 95–109. Springer, 2012.

[4] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[5] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. Specification patterns from research to industry: A case study in service-based applications. In *ICSE*, pages 968–976. IEEE, 2012.

[6] 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.

[7] 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.

[8] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for LTL synthesis. In *CAV*, volume 7358 of *LNCS*, pages 652–657. Springer, 2012.

[9] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, volume 6174 of *LNCS*, pages 380–395. Springer, 2010.

[10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[11] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI*, volume 4905 of *LNCS*, pages 52–67. Springer, 2008.

[12] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9, 2013.

[13] W. Dou, D. Bianculli, and L. C. Briand. OCLR: A more expressive, pattern-based temporal extension of OCL. In *ECMFA*, volume 8569 of *LNCS*, pages 51–66. Springer, 2014.

[14] K. Dräger, V. Forejt, M. Z. Kwiatkowska, D. Parker, and M. Ujma. Permissive controller synthesis for probabilistic systems. In *TACAS*, volume 8413 of *LNCS*, pages 531–546. Springer, 2014.

[15] A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized büchi automata. In *MASCOTS*, pages 76–83. IEEE Comp. Soc., 2004.

[16] 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.

[17] R. Ehlers. Minimising deterministic Büchi automata precisely using SAT solving. In *SAT*, volume 6175 of *LNCS*, pages 326–332. Springer-Verlag, 2010.

[18] R. Ehlers. Generalized Rabin(1) synthesis with applications to robust system synthesis. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 101–115. Springer, 2011.

[19] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *TACAS*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.

[20] L. Grunske. Specification patterns for probabilistic quality properties. In *ICSE*, pages 31–40. ACM, 2008.

[21] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124. IEEE, 2006.

[22] B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, volume 4590 of *LNCS*, pages 258–262. Springer, 2007.

[23] J. Klein and C. Baier. On-the-fly stuttering in the construction of deterministic *omega* -automata. In *CIAA*, volume 4783 of *LNCS*, pages 51–61. Springer, 2007.

[24] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE*, pages 372–381. ACM, 2005.

[25] 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.

[26] O. Kupferman. Automata Theory and Model Checking. In E. Clarke, T. A. Henzinger, and H. Veith, editors, *Handbook of Theoretical Computer Science*, chapter 7. Springer, 2015.

[27] O. Kupferman, G. Morgenstern, and A. Murano. Typeness for omega-regular automata. *Int. J. Found. Comput. Sci.*, 17(4):869–884, 2006.

[28] O. Kupferman and A. Rosenberg. The blowup in translating LTL to deterministic automata. In *6th Int. Work. on Model Checking and AI (MoChArt)*, volume 6572 of *LNCS*, pages 85–94. Springer, 2010.

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

[30] S. Maoz and J. O. Ringert. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *Proc. 4th Workshop on Synthesis (SYNT), co-located with CAV*, 2015. To appear.

[31] S. Maoz and Y. Sa'ar. AspectLTL: an aspect language for LTL specifications. In P. Borba and S. Chiba, editors, *AOSD*, pages 19–30. ACM, 2011.

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

[33] S. Maoz and Y. Sa'ar. Counter play-out: executing unrealizable scenario-based specifications. In *ICSE*, pages 242–251. IEEE / ACM, 2013.

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

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

[36] A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV: A framework for developing verification algorithms. In *CAV*, volume 6174 of *LNCS*, pages 171–174. Springer, 2010.

[37] V. Raman and H. Kress-Gazit. Explaining impossible high-level robot behaviors. *IEEE Trans. Robotics*, 29(1):94–104, 2013.

[38] S. Schewe. Beyond Hyper-Minimisation—Minimising DBAs and DPAs is NP-Complete. In *FSTTCS*, volume 8 of *LIPIcs*, pages 400–411, 2010.

[39] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. In *ICSE*, pages 11–21. ACM, 2002.

[40] Supporting Materials Website. `http://smlab.cs.tau.ac.il/syntech/patterns/`.