Inherent Vacuity for GR(1) Specifications

Shahar Maoz
Tel Aviv University
Israel

Rafi Shalom
Tel Aviv University
Israel

ABSTRACT

Vacuity is a well-known quality issue in formal specifications, studied mostly in the context of model checking. Inherent vacuity is a type of vacuity that applies to specifications, without the context of a model. GR(1) is an expressive assume-guarantee fragment of LTL, which enables efficient symbolic synthesis.

In this work we investigate inherent vacuity for GR(1) specifications. We define several general types of inherent vacuity for GR(1), including specification element vacuity and domain value vacuity. We detect vacuities using a reduction to LTL satisfiability, specialized for the context of GR(1). We further extend vacuity detection to handle GR(1) specifications that are enriched with past LTL, monitors, and patterns. Finally, we define a novel notion of vacuity core, which provides means to localize the cause of vacuity.

We implemented our work and evaluated it on benchmarks from the literature. The evaluation shows that vacuities are indeed common in GR(1) specifications, and that we are able to efficiently detect them and effectively localize their causes. Moreover, our evaluation shows that removal of vacuous specification elements may significantly reduce synthesis time.

CCS CONCEPTS

• **Software and its engineering** → *Formal methods.*

KEYWORDS

reactive synthesis, GR(1), vacuity

ACM Reference Format:
Shahar Maoz and Rafi Shalom. 2020. Inherent Vacuity for GR(1) Specifications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409669

1 INTRODUCTION

Vacuity is a well-known quality issue in formal specifications. Vacuity has been studied extensively and implemented in the context of model checking, where one checks whether a system model satisfies a specification and vacuity usually means that some elements of the specification play no role in that satisfaction. Inherent vacuity is a type of vacuity that applies to specifications without the context of an existing model, e.g., ones used for the purpose of synthesis. Roughly, a specification is inherently vacuous if one or more of its elements is logically redundant. Inherent vacuity has been studied theoretically, very generally, in the context of LTL specifications [20]. However, to the best of our knowledge, it has not yet been defined, implemented, and evaluated in any concrete synthesis setup.

GR(1) is an assume-guarantee fragment of Linear Temporal Logic (LTL) that has an efficient symbolic synthesis algorithm [10] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [16, 30]. 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 recently used in several application domains, e.g., to specify and implement autonomous robots [27, 31], control protocols for smart camera networks [37], distributed control protocols for aircraft vehicle management systems [36], and device drivers [39]. Several tools support GR(1) synthesis [7, 18, 33, 43].

In this work we investigate inherent vacuity for GR(1) specifications. Our first contribution consists of the definition of several types of vacuity for GR(1) specifications, including specification elements vacuities and domain value vacuities. Specifically, we take advantage of the structure of GR(1) specifications and present case-based definitions of vacuous elements and vacuous domain values. Roughly, a specification element is vacuous if removing it will not change the semantics of the specification, and a domain value is vacuous if it is unreachable in any run that satisfies the specification. The formal definitions of these vacuity types appear in Sect. 4. We further show how to efficiently detect these vacuity types using a reduction to a satisfiability problem over formulas consisting of elements from GR(1) specifications and of their negation. See Sect. 5.

It is important to note that inherent vacuity (unlike, e.g., unrealizability, non-well-separation) is defined not only with regard to the semantics of the specification but fundamentally also with regard to its concrete syntax. Thus, two semantically equivalent specifications may exhibit different vacuities. As a simple example, although the formulas $p \rightarrow q$ and $(p \rightarrow q) \wedge (\neg p \vee q)$ are semantically equivalent, the first has no vacuous elements while the second has. Indeed, our definitions and algorithms for vacuity detection consider not only the semantics of the GR(1) specification but also its syntax. We consider it to be a unique and interesting aspect of our work.

Detecting vacuity is important, but by itself not informative enough. Thus, **as a second contribution we present means for localizing the cause of the vacuity, by computing what we call a vacuity core, a locally minimal subset of the specification that is necessary and sufficient for the detected vacuity.** We compute vacuity cores using a delta debugging approach [44]. See Sect. 6.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409669

Finally, we extend the scope of inherent vacuity definitions and detection algorithm to unrealizable GR(1) specifications as well as to specifications that include, beyond pure GR(1) elements, also past LTL operators, monitors, and patterns. These extensions are important because many specifications written during a development process may be unrealizable and because past LTL operators, monitors, and patterns, help engineers write more concise and readable specifications [33]. See Sect. 7.

We have implemented all our ideas as an extension of Spectra [33, 42], an open source specification language and tool set for reactive synthesis. We present an evaluation over benchmarks from the literature. **Our evaluation shows (1) that vacuity indeed commonly occurs in GR(1) specifications, (2) that our algorithms effectively and efficiently detect the different types of vacuity, (3) that the vacuity core is effective in localizing the cause of vacuity, and (4) that controller synthesis running times are in many cases significantly reduced when a vacuous element is removed from a specification**. See Sect. 8.

Vacuity has been studied in the literature and implemented in model checkers (e.g., [4, 5, 13, 14, 23, 24, 28, 40]). In the context of synthesis, however, we are only aware of the theoretical framework of inherent vacuity [20]. In particular, to our knowledge, our work is the first to define, examine, and evaluate inherent vacuity for GR(1) specifications. Note that in the context of model checking, vacuity is a property of a specification and a model. The model may satisfy the specification in a vacuous way. Our context is fundamentally different. We deal with *inherent* vacuity, which is a property of the specification alone. Vacuity checks are now a standard component in commercial model checkers [14]. We believe they will become standard components in future synthesizers. We discuss related work in Sect. 9.

2 RUNNING EXAMPLE

As a running example for this paper, we use a lift specification (see List. 1), which has appeared in several variants in previous GR(1)-related papers [2, 10, 11, 38]. The specification is written in Spectra format [33, 42]. We adapted the example by using an integer value to represent the floor number (i.e., variable `f` in line 5) instead of one Boolean variable per floor [2], in order to make the example more concise, intelligible, and scalable. The example is small and simple, to fit the paper presentation. In our evaluation we have used larger and more complex specifications, taken from benchmarks.

The specification models a controller for a three floors lift. The lift has three request buttons, one on each floor. Requests are represented by environment variables `b1`, `b2`, and `b3`, which may be independently true or false. The current floor of the lift is represented by the system variable `f`. The environment is required to initially have no requests (line 8), turn off any granted request at the next step (lines 11-13), and keep ungranted requests (lines 16-18). The system is required to start the lift on the first floor (line 21), and to disallow the lift to move more than one floor at a time (line 24). The system is also required not to move up when there are no requests (line 27), to eventually grant every request (lines 30-32), and to make sure every floor is visited infinitely often (lines 35-37).

Are there vacuous assumptions or guarantees in our example specification? Our tool finds four vacuous guarantees: the three

```
1  env boolean b1;
2  env boolean b2;
3  env boolean b3;
4
5  sys Int(1..3) f;
6
7  // No buttons are initially pressed
8  asm !b1 and !b2 and !b3;
9
10 // Request is removed when satisfied
11 asm G ( (b1 and f=1) -> next(!b1));
12 asm G ( (b2 and f=2) -> next(!b2));
13 asm G ( (b3 and f=3) -> next(!b3));
14
15 // Request must remain while unsatisfied
16 asm G ( (b1 and f!=1) -> next(b1));
17 asm G ( (b2 and f!=2) -> next(b2));
18 asm G ( (b3 and f!=3) -> next(b3));
19
20 // Lift is initially at lowest floor
21 gar f=1;
22
23 // Always stay at the same floor or move to an adjacent floor
24 gar G (f>=next(f)-1 and f<=next(f)+1);
25
26 // Do not move up when there are no requests
27 gar G (f<next(f)) ->(b1 or b2 or b3);
28
29 // Eventually grant each request
30 gar GF (b1 -> f=1);
31 gar GF (b2 -> f=2);
32 gar GF (b3 -> f=3);
33
34 // Visit every floor infinitely often
35 gar GF f=1;
36 gar GF f=2;
37 gar GF f=3;
```

Listing 1: Lift controller specification, adopted from [2]

```
1 // Don't go down when at first floor
2 gar G f=1 -> next(f)>=f;
```

Listing 2: Trivial vacuity example

guarantees in lines 30-32, and the guarantee in line 36. Intuitively, removing any single one of these guarantees, does not change the specification's semantics.

Why are these guarantees vacuous? Our tool also provides a cause for each vacuity in the form of a locally minimal subset of assumptions and guarantees that imply it. Specifically, each of the vacuous guarantees in lines 30-32 is directly implied by one of the guarantees in lines 35-37. The guarantee in line 36 is implied by the conjunction of the guarantees in lines 24, 35, and 37. Indeed, if the lift must be on the first and on the third floors infinitely often, it must also be infinitely often on the second floor.

Finally, we demonstrate an additional vacuity by adding List. 2, which contains a guarantee that prevents the lift from going down when it is on the first floor. This guarantee is vacuous because the variable `f` already has 1 as a minimal value, regardless of any assumption or guarantee. We call such vacuities trivial vacuities. Our tool detects it as a trivial vacuity.

3 PRELIMINARIES

3.1 Linear Temporal Logic (LTL)

We use a standard definition of *linear temporal logic (LTL)*, e.g., as found in [10], over present-future temporal operators **X** (next), **U** (until), **F** (finally), and **G** (globally), and past temporal operator **H** (historically). For a finite set of Boolean variables $\mathcal{V}$, a *computation* $\sigma = s_0 s_1 .. \in (2^{\mathcal{V}})^{\omega}$ is an infinite sequence of *states*, i.e., of truth assignments s_i to $\mathcal{V}$. We use $\sigma, i \models \psi$ to denote that the LTL formula ψ holds at position $i \geq 0$ of σ.

We denote $\sigma, 0 \models \psi$ by $\sigma \models \psi$, and say that σ satisfies ψ. Two LTL formulas φ and ψ are LTL equivalent, denoted $\varphi \equiv_{\text{LTL}} \psi$, iff for all computations σ, $\sigma \models \varphi$ iff $\sigma \models \psi$. One LTL formula φ implies another LTL formula ψ, denoted $\varphi \rightarrow_{\text{LTL}} \psi$, iff for all computations σ, if $\sigma \models \varphi$ then $\sigma \models \psi$. An LTL formula φ is satisfiable iff there is a computation σ s.t. $\sigma \models \varphi$.

3.2 GR(1) and GR(1) Realizability

LTL formulas can be used as specifications of reactive systems, where atomic propositions are interpreted as environment (input) and system (output) variables. An assignment to all variables is called a state.

A strategy for an LTL specification φ prescribes the outputs of a system that from its winning states for all environment choices lead to computations that satisfy φ. A specification φ is called realizable if a strategy exists such that for all initial environment choices the initial states are winning states. The goal of LTL synthesis is, given an LTL specification, to find a strategy that realizes it, if one exists.

GR(1) synthesis [10] handles a fragment of LTL where specifications contain initial assumptions and guarantees over initial states, safety assumptions and guarantees relating the current and next state, and justice assumptions and guarantees requiring that an assertion holds infinitely many times during a computation. A GR(1) specification $\mathcal{S}$ consists of the following elements [10]:

- $\mathcal{X}$ input variables controlled by the environment;
- $\mathcal{Y}$ output variables controlled by the system;
- $\mathcal{X}'$ and $\mathcal{Y}'$ copies of input and output variables at next step
- θ^e assertion over $\mathcal{X}$ characterizing initial environment states;
- θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial system states;
- $\rho^e(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}')$ transition relation of the environment;
- $\rho^s(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}')$ transition relation of the system;
- $J^e_{i \in 1..n}$ justice goals of the environment;
- $J^s_{j \in 1..m}$ justice goals of the system.

A GR(1) specification is realizable, i.e., allows an implementation, iff the following LTL formula is realizable[1]:

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

Roughly, φ^{sr} means that if the environment keeps all initial assumptions then the system should keep all initial guarantees, as long as the environment keeps all safety assumptions the system should keep all safety guarantees, and in all infinite plays, if the environment keeps all justice assumptions the system should keep all justice guarantees.

Specifications for GR(1) synthesis have to be expressible in the above structure and thus do not cover the complete LTL. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis have been presented in [10, 38]. Specifically, realizability checking is done in time $O(nmN^2)$, where N is the size of the state space $2^{\mathcal{X} \cup \mathcal{Y}}$. The algorithm of [10, 38] computes winning states for the system, i.e., states from which the system can realize φ^{sr}.

[1]We use the definition of strict realizability, as defined in [10] and implemented in several GR(1) synthesis tools, including Slugs [18] and Spectra [33].

The semantics of a specification in the context of inherent vacuity was defined as the set of its implementations [20]. We use here LTL semantics, which for GR(1) is the set of computations that φ^{sr} satisfies, formally $\{\sigma | \sigma \models \varphi^{sr}\}$. This semantics is finer as it determines the set of implementations [20, 22], but not vice versa.

We use a parametrized version of φ^{sr} for our proofs, namely $\varphi^{sr}[p_1, p_2, p_3, p_4, p_5, p_6] = (p_1 \rightarrow p_2) \wedge (p_1 \rightarrow \mathbf{G}((\mathbf{H}p_3) \rightarrow p_4)) \wedge (p_1 \wedge \mathbf{G}p_3 \rightarrow (p_5 \rightarrow p_6))$.

Thus, $\varphi^{sr} = \varphi^{sr}[\theta^e, \theta^s, \rho_e, \rho_s, \bigwedge_{i \in 1..n} \mathbf{GF} J^e_i, \bigwedge_{j \in 1..m} \mathbf{GF} J^s_j]$. We also use a dot notation to avoid rewriting replacements that remain the same, e.g., $\varphi^{sr}[\alpha, \cdot, \cdot, \cdot, \beta, \gamma] = (\alpha \rightarrow \theta^s) \wedge (\alpha \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s)) \wedge (\alpha \wedge \mathbf{G}\rho^e \rightarrow (\beta \rightarrow \gamma))$.

3.3 Abstract Syntax of a Specification

Since vacuity depends not only on semantics but also on syntax, in our context it is important to consider variables with multi-valued domains and to handle individual specification elements (e.g., initial assertions) separately. We provide an abstract syntax definition of a GR(1) specification (inspired by Spectra [42]).

Definition 1 (Abstract syntax of a specification). *A GR(1) specification is a tuple* $Spec = \langle V_e, V_s, D, M_e, M_s \rangle$, *where* V_e *and* V_s *are sets of environment and system variables respectively,* $D : V_e \cup V_s \rightarrow Doms$ *assigns a finite domain to each variable*[2], *and* M_e *and* M_s *are the environment and system modules. A module is a triplet* $M = \langle I, T, J \rangle$ *that contains sets of initial assertions* $I = \{I_n\}^i_{n=1}$, *safety assertions* $T = \{T_n\}^t_{n=1}$, *and justice assertions* $J = \{J_n\}^j_{n=1}$ *of the module, where* $i = |I|$, $t = |T|$ *and* $j = |J|$. *The set of elements of module* $M = \langle I, T, J \rangle$ *is* $B_M = I \cup \{\mathbf{G}\, T_i\}^t_{i=1} \cup \{\mathbf{GF}\, J_i\}^j_{i=1}$.

3.4 Basic Symbolic Algorithms

We describe two basic algorithms [21] that we implemented symbolically and use in our work, namely reachability (Alg. 1), and a generalized Buchi winning region computation (Alg. 2).

We write here in μ-calculus notation the formulas for the two algorithms. The modal μ-calculus is a fixed-point logic [26].

$$R(\rho, F) = [\![\mu Y.F \vee \diamond Y]\!] \quad (2)$$

$$W(\rho, \{J_i\}^n_{i=1}) = [\![\nu Z. \bigwedge_{i=1}^{n} \mu Y.(J_i \wedge \diamond Z) \vee \diamond Y]\!] \quad (3)$$

Equ. 2 describes states from which one can reach states in F with a finite path using the transition relation ρ. Equ. 3 describes states from which there is a path according to ρ that reaches all sets $\{J_i\}^n_{i=1}$ infinitely often[3]. Both μ-formulas in Equ. 2 and 3 are interpreted over the transition system with states $2^{\mathcal{X}}$ and transition relation $\rho \subseteq 2^{\mathcal{X}} \times 2^{\mathcal{X}}$. We denote with $\diamond X$ states that have a transition entering a state in X according to ρ. Explicitly, the semantics of $\diamond X$ is the set $\{s | \exists s' \in X \rho(s, s')\}$.

Symbolic algorithms in our context operate on sets of states and transitions over Boolean variables $\mathcal{X}$ and their primed copy $\mathcal{X}'$ instead of on their explicit representations. We operate on assertions using the usual Boolean operators. All operations used in our

[2]We use formulas with non-binary variables without explicitly translating them, e.g., we may write $\mathbf{GF}\ x \geq 2$ where $D(x) = \{1, 2, 3\}$ without using two binary variables for x. Our implementation fully supports multi-valued variables.

[3]If the set of justices is empty, we define one justice that contains all states.

Algorithm 1 Reach computes all states that have a path to a set of states given a transition relation.

Require: A transition relation ρ, and a set of states X
Ensure: returns states that reach X
1: $R \leftarrow X$
2: $R' \leftarrow \top$
3: **while** $R \neq R'$ **do**
4: $\quad R' \leftarrow R$
5: $\quad R \leftarrow X \vee pred(\rho, R)$
6: **end while**
7: return R

Algorithm 2 BuchiWinRegion computes a winning region of a generalized Buchi condition given a transition relation and a set of justices.

Require: A transition relation ρ, and a set of justices $J = \{J_i\}_{i=1}^{n}$
Ensure: returns winning region of the generalized Buchi condition
1: **if** $J = \emptyset$ **then**
2: $\quad J \leftarrow \{\top\}$
3: **end if**
4: $W \leftarrow \top$
5: $W' \leftarrow \bot$
6: **while** $W \neq W'$ **do**
7: $\quad W' \leftarrow W$
8: $\quad$ **for** $J_i \in J$ **do**
9: $\quad\quad start \leftarrow pred(\rho, W) \wedge J_i$
10: $\quad\quad W \leftarrow W \wedge \mathbf{Reach}(\rho, start)$
11: $\quad$ **end for**
12: **end while**
13: return W

algorithms are symbolic and have direct implementations using Binary Decision Diagrams (BDDs) with CUDD [41].

We use $pred(\rho, \xi)$, computed symbolically by $\exists \mathcal{X}'(\rho \wedge prime(\xi))$, for states that can enter a state in ξ according to the transition relation ρ. $prime(\xi)$ translates an assertion ξ over $\mathcal{X}$ to an equivalent assertion over $\mathcal{X}'$. Existential quantification over $\mathcal{X}'$ yields an assertion without $\mathcal{X}'$ that holds iff there exists an assignment to variables in $\mathcal{X}'$ s.t. the assertion holds.

4 DEFINING INHERENT VACUITY FOR GR(1)

Intuitively, a vacuous element of a specification is a part of it that is redundant in the sense that if we remove it, the semantics of the specification will not change.

Since we are interested in GR(1) specifications, we choose a pragmatic, case-based approach, which takes advantage of the structure of GR(1) specifications. We require preservation of LTL semantics, which ensures that the set of implementations remains unchanged [20, 22]. We then define several types of vacuities that are applicable to GR(1) specifications. We consider several cases corresponding to different elements (i.e., members of $B_{M_e} \cup B_{M_s}$) of the specification. For each case, we define a set of sub-formulas, which we call a premise-set, and another sub-formula we call a consequent. We will then prove that if the premise-set implies the consequent, the specification element in question is indeed a vacuity.

4.1 Specification Elements Vacuity

We first define an element of a specification as vacuous when its removal from the specification does not change the LTL semantics of the strict realizability formula.

Definition 2 (Vacuous element of a GR(1) specification). *For a specification* $Spec = \langle V_e, V_s, D, M_e, M_s \rangle$ *and an element* $v \in B_{M_e} \cup B_{M_s}$, *let* φ^{sr} *denote the LTL formula for the realizability of Spec (see Equ. 1), and let* $\varphi^{sr}_{\underline{v}}$ *denote the same formula for the specification Spec without* v, *then* v *is vacuous iff* $\varphi^{sr} \equiv_{LTL} \varphi^{sr}_{\underline{v}}$, *i.e., the removal of* v *does not change the LTL semantics of the specification.*

Table 1: Specification element vacuity types

	M	T	v	$ps(v)$	$cons(v)$
1	E	V^e_I	$i \in I_e$	$I_e \setminus \{i\}$	i
2		V^e_S	$\mathbf{G}\, t; t \in T_e$	$T_e \setminus \{t\}$	t
3		V^e_J	$\mathbf{GF}\, j; j \in J_e$	$I_e \cup I_s \cup \{\mathbf{G}\, t \mid t \in T_e \cup T_s\} \cup \{\mathbf{GF}\, x \mid x \in J_e \setminus \{j\}\}$	$\mathbf{GF}\, j$
4	S	V^s_I	$i \in I_s$	$I_e \cup (I_s \setminus \{i\})$	i
5		V^s_S	$\mathbf{G}\, t; t \in T_s$	$T_e \cup (T_s \setminus \{t\})$	t
6		V^s_J	$\mathbf{GF}\, j; j \in J_s$	$I_e \cup I_s \cup \{\mathbf{G}\, t \mid t \in T_e \cup T_s\} \cup \{\mathbf{GF}\, x \mid x \in J_e \cup (J_s \setminus \{j\})\}$	$\mathbf{GF}\, j$

We now define types of vacuities for specification elements. Each element (whether initial, safety, or justice element) in each module (environment or system) may be vacuous. Intuitively, an element is vacuous when a subset of other elements implies it, rendering it redundant. Thus, we will define and detect each vacuity type based on a pair: a premise-set *ps* and a consequent *cons*; whenever the latter is implied by the conjunction of the elements in the former, we have a correct vacuity.

Table 1 lists six types of vacuities, their respective module (E stands for environment and S for system), their type name (under T), and their respective premise-set and consequent. For example, line 5 of Table 1 defines the vacuity of system module safeties V^s_S: A system module safety $\mathbf{G}\, t$ for $t \in T_s$ is vacuous, if the conjunction of the propositional parts of all environment and system safeties, without it, implies its consequent t, i.e., if $(\wedge_{\alpha \in T_e \cup (T_s \setminus \{t\})} \alpha) \rightarrow_{\text{LTL}} t$.

Remark 1. *Note that safety elements vacuities are defined without the temporal operator* $\mathbf{G}$. *Indeed, it would be incorrect to consider implication between safeties with the temporal operator* $\mathbf{G}$, *i.e., it would not define a vacuity. Consider the specification* $\langle \{a, a_1\}, \{b\}, D, M_e, M_s \rangle$, *where* a *and* a_1 *are Boolean environment variables,* $D(b) = \{1, 2, 3\}$, $B_{M_e} = \{\mathbf{G}\, \neg a, \mathbf{G}\, (b = 1 \rightarrow \mathbf{X}a)\}$, *and* $B_{M_s} = \{\mathbf{G}\, (b < 3 \rightarrow (b < \mathbf{X}b \wedge \mathbf{X}b < 3)), \mathbf{G}\, (b = 3), \mathbf{G}\, (b = 3 \rightarrow \mathbf{X}a_1)\}$. *In this specification* $\mathbf{G}\, (b < 3 \rightarrow (b < \mathbf{X}b \wedge \mathbf{X}b < 3)) \rightarrow_{LTL} \mathbf{G}\, (b = 3)$. *The specification is unrealizable, but if we remove* $\mathbf{G}\, (b = 3)$ *from* B_{M_s} *the specification becomes realizable, because now the system can win by setting* $b = 1$ *at the initial state. Clearly, the semantics has changed. See [1] for additional examples that guided our definition of vacuity types.*

Recall the vacuities in the Lift specification from Sect. 2. We can now explain their types, premise-sets, etc.

Example 1 (Lift example vacuities). *The four vacuities in List. 1 are of type* V^s_J. *Their consequent is thus themselves, and their premise-sets are all the elements of the specification excluding themselves. The vacuity in List. 2 is of type* V^s_S. *Its consequent is thus its propositional part, and its premise-set is the set of propositional parts of all other safeties in the specification.*

We now prove the correctness of Table 1, i.e., that for each of the six formulas v in the table, when the premise-set implies its consequent, the formula is a vacuity.

Theorem 1 (Table 1 element removal preserves LTL semantics). *Given a specification* $\langle V_e, V_s, D, M_e, M_s \rangle$ *and* $v \in B_{M_e} \cup B_{M_s}$, *for the six definitions of premise-set* $ps(v)$ *and consequent* $cons(v)$

presented in Table 1, if $(\wedge_{\alpha \in ps(v)} \alpha) \rightarrow_{LTL} cons(v)$*, then* v *is a vacuity, i.e., its removal preserves the LTL semantics of the specification. Formally,* $\varphi^{sr} \equiv_{LTL} \varphi^{sr}_{\underline{v}}$ *as defined in Definition 2.*

Proof. Let $M_e = \langle I_e, T_e, J_e \rangle$ and $M_s = \langle I_s, T_s, J_s \rangle$. The proof handles each type of vacuity separately.

Type V^e_I:
Let $v = i \in B_{M_e}$ be a vacuity of type V^e_I, then $v = i \in I_e$, $\theta^e = \wedge_{x \in I_e} x$ and $\theta^e_{\underline{v}} = \wedge_{x \in I_e \setminus \{i\}} x$. Since v is an initial environment vacuity we know that $\theta^e_{\underline{v}} \rightarrow i$, thus $\theta^e \equiv \theta^e_{\underline{v}} \wedge i \equiv \theta^e_{\underline{v}}$, and therefore $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\theta^e_{\underline{v}}, \cdot, \cdot, \cdot, \cdot, \cdot] \equiv_{\text{LTL}} \varphi^{sr}[\theta^e, \cdot, \cdot, \cdot, \cdot, \cdot] = \varphi^{sr}$.

Type V^s_I:
Let $v = i \in B_{M_s}$ be a vacuity of type V^s_I, then $v = i \in I_s$, $\theta^s = \wedge_{x \in I_s} x$ and $\theta^s_{\underline{v}} = \wedge_{x \in I_s \setminus \{i\}} x$. Since v is an initial system vacuity we know that $\theta^e \wedge \theta^s_{\underline{v}} \rightarrow i$, thus $\theta^e \rightarrow \theta^s \equiv \theta^e \rightarrow \theta^s_{\underline{v}}$, and therefore $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\theta^e, \theta^s_{\underline{v}}, \cdot, \cdot, \cdot, \cdot] \equiv_{\text{LTL}} \varphi^{sr}[\theta^e, \theta^s, \cdot, \cdot, \cdot, \cdot] = \varphi^{sr}$.

Type V^e_S:
Let $v = \mathbf{G}\, t \in B_{M_e}$ be a vacuity of type V^e_S, then $t \in T_e$, $\rho^e = \wedge_{x \in T_e} x$ and $\rho^e_{\underline{v}} = \wedge_{x \in T_e \setminus \{t\}} x$. Since v is a safety environment vacuity we know that $\rho^e_{\underline{v}} \rightarrow t$, thus $\rho^e \equiv \rho^e_{\underline{v}} \wedge t \equiv \rho^e_{\underline{v}}$, and therefore $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\cdot, \cdot, \rho^e_{\underline{v}}, \cdot, \cdot, \cdot] \equiv_{\text{LTL}} \varphi^{sr}[\cdot, \cdot, \rho^e, \cdot, \cdot, \cdot] = \varphi^{sr}$.

Type V^s_S:
Let $v = \mathbf{G}\, t \in B_{M_s}$ be a vacuity of type V^s_S, then $t \in T_s$, $\rho^s = \wedge_{x \in T_s} x$ and $\rho^s_{\underline{v}} = \wedge_{x \in T_s \setminus \{t\}} x$. Proving that $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\cdot, \cdot, \cdot, \rho^s_{\underline{v}}, \cdot, \cdot] \equiv_{\text{LTL}} \varphi^{sr}[\cdot, \cdot, \cdot, \rho^s, \cdot, \cdot] = \varphi^{sr}$, is equivalent to proving that $\mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s) \equiv_{\text{LTL}} \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s_{\underline{v}})$. Note that since $\rho^s \rightarrow \rho^s_{\underline{v}}$, then also $\mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s) \rightarrow_{\text{LTL}} \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s_{\underline{v}})$ holds. Conversely, let σ be a computation such that $\sigma \models \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s_{\underline{v}})$. We now show that for all $i \geq 0$, we have $\sigma, i \models (\mathbf{H}\rho^e) \rightarrow \rho^s$, which proves that $\mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s_{\underline{v}}) \rightarrow_{\text{LTL}} \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s)$. For every $i \geq 0$, either $\sigma, i \not\models \mathbf{H}\rho^e$, thus $\sigma, i \models (\mathbf{H}\rho^e) \rightarrow \rho^s$, and we are done. Otherwise assume $\sigma, i \models \mathbf{H}\rho^e$, then $\sigma, i \models \rho^s_{\underline{v}}$, plus according to the definition of $\mathbf{H}$ we have $\sigma, i \models \rho^e$. Since $v = \mathbf{G}\, t$ is a safety system vacuity then $\rho^e \wedge \rho^s_{\underline{v}} \rightarrow t$, thus $\sigma, i \models \rho^s_{\underline{v}} \wedge t \equiv \rho^s$.

Preliminary considerations for both types V^e_J and V^s_J:
For justice vacuities of both kinds we show bellow that $\varphi^{sr} \equiv_{\text{LTL}} \varphi^{sr}_{\underline{v}}$ by showing both $\varphi^{sr} \rightarrow_{\text{LTL}} \varphi^{sr}_{\underline{v}}$, and $\varphi^{sr}_{\underline{v}} \rightarrow_{\text{LTL}} \varphi^{sr}$, thus we always assume a computation σ such that either $\sigma \models \varphi^{sr}$ or $\sigma \models \varphi^{sr}_{\underline{v}}$ holds, in order to imply the other. Before we discuss both types of vacuities separately, we show that it is enough to consider only computations σ that satisfy $\sigma \models \theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s$.

First note that for all $\sigma \not\models \theta^e \wedge \mathbf{G}\rho^e$, we know that $\sigma \models \varphi^{sr}$ iff $\sigma \models (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s))$ iff $\sigma \models \varphi^{sr}_{\underline{v}}$. Thus we only need to prove that $\varphi^{sr} \equiv_{\text{LTL}} \varphi^{sr}_{\underline{v}}$ for computations σ that satisfy $\sigma \models \theta^e \wedge \mathbf{G}\rho^e$. Also note that when we assume that $\sigma \models \varphi^{sr}_{\underline{v}}$ holds or $\sigma \models \varphi^{sr}$ holds, then $\sigma \models \theta^s$ holds because in both cases $\sigma \models \theta^e \rightarrow \theta^s$, and we focus on computations that satisfy $\sigma \models \theta^e$. Similarly, we know that $\sigma \models \mathbf{G}\rho^s$ because $\sigma \models \theta^e \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s)$ holds in both cases, and we focus on computations for which $\sigma \models \theta^e \wedge \mathbf{G}\rho^e$ holds.

Type V^e_J:
Let $v = \mathbf{GF}\, j \in B_{M_e}$ be a vacuity of type V^e_J. In order to prove that $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\cdot, \cdot, \cdot, \cdot, \wedge_{x \in J_e \setminus \{j\}} \mathbf{GF}\, x, \cdot] \equiv_{\text{LTL}} \varphi^{sr}[\cdot, \cdot, \cdot, \cdot, \wedge_{x \in J_e} \mathbf{GF}\, x, \cdot] = \varphi^{sr}$, it is enough to show that $\sigma \models \wedge_{x \in J_e} \mathbf{GF}\, x$ iff $\sigma \models \wedge_{x \in J_e \setminus \{j\}} \mathbf{GF}\, x$ for all computations $\sigma \models \theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s$. Evidently always $\wedge_{x \in J_e} \mathbf{GF}\, x \rightarrow_{\text{LTL}} \wedge_{x \in J_e \setminus \{j\}} \mathbf{GF}\, x$. Since $\mathbf{GF}\, j$ is an environment justice vacuity, we know that $\theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s \wedge \wedge_{x \in J_e \setminus \{j\}} \mathbf{GF}\, x \rightarrow_{\text{LTL}} \mathbf{GF} j$, thus for all computations $\sigma \models \theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s$, we know that $\sigma \models \wedge_{x \in J_e \setminus \{j\}} \mathbf{GF}\, x$ implies $\sigma \models \wedge_{x \in J_e} \mathbf{GF}\, x$.

Type V^s_J:
Let $v = \mathbf{GF}\, j \in B_{M_s}$ be a vacuity of type V^s_J. For all computations σ such that $\sigma \not\models \wedge_{x \in J_e} \mathbf{GF}\, x$ we know that $\sigma \models \varphi^{sr}_{\underline{v}}$ iff $\sigma \models (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s))$ iff $\sigma \models \varphi^{sr}$.

Otherwise, $\sigma \models \wedge_{x \in J_e} \mathbf{GF}\, x$, and in order to prove that $\varphi^{sr}_{\underline{v}} = \varphi^{sr}[\cdot, \cdot, \cdot, \cdot, \cdot, \wedge_{x \in J_s \setminus \{j\}} \mathbf{GF}\, x] \equiv_{\text{LTL}} \varphi^{sr}[\cdot, \cdot, \cdot, \cdot, \cdot, \wedge_{x \in J_s} \mathbf{GF}\, x] = \varphi^{sr}$, it remains to show that $\sigma \models \wedge_{x \in J_s} \mathbf{GF}\, x$ iff $\sigma \models \wedge_{x \in J_s \setminus \{j\}} \mathbf{GF}\, x$ for all computations that satisfy $\sigma \models \theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s \wedge_{x \in J_e} \mathbf{GF}\, x$. However, since $\mathbf{GF}\, j$ is a system justice vacuity, we know that $\theta^e \wedge \theta^s \wedge \mathbf{G}\rho^e \wedge \mathbf{G}\rho^s \wedge \wedge_{x \in J_e} \mathbf{GF}\, x \wedge \wedge_{x \in J_s \setminus \{j\}} \mathbf{GF}\, x \rightarrow_{\text{LTL}} \mathbf{GF} j$. □

Table 2: Unreachable domain values vacuity types

M	T	v	$ps(v)$	$cons(v)$
E	V^e_D	$dv(var, value); var \in V_e; value \in D(var)$	T_e	$var \neq value$
S	V^s_D	$dv(var, value); var \in V_s; value \in D(var)$	$T_e \cup T_s$	$var \neq value$

4.2 Unreachable Domain Values Vacuity

Second, we consider unreachable domain values, i.e., values that are unreachable in any controller that realizes the specification, as another form of vacuity. Note that this is an extension of the notion of vacuity, which is usually defined only w.r.t. subformulas of a specification.

Definition 3 (Unreachable domain value formula). *An unreachable domain value formula* $dv(var, value)$ *is the formula* $\mathbf{G}\ var \neq value$.

We claim that unreachable domain value vacuities as defined in Table 2 are indeed vacuities by showing that for V^e_D (resp. V^s_D), adding $dv(var, value)$ as an assumption (resp. guarantee) to the specification, does not change its semantics.

Theorem 2 (Semantic equivalence under unreachable domain value restriction). *Given a specification* $\langle V_e, V_s, D, M_e, M_s \rangle$ *and a formula* $v = dv(var, value)$ *of type* V^e_D *(resp.* V^s_D*), such that* $(\wedge_{\alpha \in ps(v)} \alpha) \rightarrow_{LTL} cons(v)$*, adding* v *as an assumption (resp. guarantee) does not change the LTL semantics of the specification.*

Proof. Adding the safety $dv(var, value)$ in both cases would create a new specification in which $dv(var, value)$ is a vacuity of type V^e_S (resp. V^s_S). According to Thm. 1, the new specification is LTL equivalent to the original specification. □

4.3 Unsatisfiability as Vacuity: Avoiding Vacuous Vacuities

Finally, we define unsatisfiable specifications as vacuous. Unsatisfiability is a form of inherent vacuity [20]. Intuitively, if the conjunction of all the elements in the specification is equivalent to false, the specification is considered vacuous. Formally:

Definition 4 (Unsatisfiability as vacuity). *For a specification* $Spec = \langle V_e, V_s, D, M_e, M_s \rangle$*, consider the formula* $\perp$ *with the premise-set* $ps(\perp) = B_{M_s} \cup B_{M_e}$ *and the consequent* $cons(\perp) = \perp$.

Algorithm 3 Sat Satisfiability of initial states, a safety and justices.

Require: θ, ρ, and $J = \{J_i\}_{i=1}^n$
Ensure: returns true iff $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i=1}^n \mathbf{GF}J_i$ is satisfiable
1: return **BuchiWinRegion**$(\rho, J) \wedge \theta \neq \bot$ ▷ Use Alg. 2

Algorithm 4 ImpJust Computes implication of a justice.

Require: θ, ρ, $J = \{J_i\}_{i=1}^n$, and J^{imp}
Ensure: returns true iff $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i=1}^n \mathbf{GF}J_i \rightarrow_{\text{LTL}} \mathbf{GF}J^{imp}$ holds
1: return **Reach**$(\rho,$ **BuchiWinRegion**$(\rho \wedge \neg J^{imp}, J)) \wedge \theta = \bot$ ▷ Use Alg. 1, 2

The specification has a vacuity of type $V_\bot$ if it satisfies the requirement $\wedge_{\alpha \in ps(\bot)} \alpha \rightarrow_{LTL} cons(\bot)$, i.e., if $(\wedge_{\alpha \in B_{M_s} \cup B_{M_e}} \alpha) \rightarrow_{LTL} \bot$.

Defining unsatisfiable specifications as vacuous allows us to avoid the many additional meaningless vacuities one may find in this case. For example, in a specification with seven elements, such that the conjunction of three of which equals false, up to four elements may be trivially found vacuous because of the unsatisfiability of the specification, regardless of their content. To avoid reporting these vacuous vacuities (pun intended), when a specification is unsatisfiable, we define it as vacuous and avoid looking for vacuous elements in it.

5 DETECTING VACUITIES

We now describe how we detect the different types of vacuities defined in the previous section. We continue with the overall algorithm for detecting all the vacuities in a given specification. Finally, we discuss correctness and complexity.

5.1 Detecting Specific Types of Vacuity

5.1.1 Detecting Initial Assertions , Safeties, and Domain Vacuities. The premise-sets and consequents of vacuities of types V_I^e, V_I^s, V_S^e, V_S^s, V_D^e, and V_D^s, namely initial assertions, safeties, and domain vacuities, of both environment and system modules, include no LTL operators. Thus, in order to detect them, we simply use propositional implication, which in our settings is implemented symbolically using BDDs.

5.1.2 Detecting Unsatisfiability. For the unsatisfiability of a specification (vacuity of type $V_\bot$), we consider an algorithm for the satisfiability of formulas of the form $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i \in 1..n} \mathbf{GF}J_i$ over Boolean variables $\mathcal{X}$. This is easily reducible to the Buchi winning region problem in state space $2^{\mathcal{X}}$, by checking that the winning region intersects the initial states, i.e., that $W(\rho, \{J_i\}_{i=1}^n) \cap \theta \neq \emptyset$. This yields Alg. 3.

5.1.3 Detecting Justice Vacuities. For vacuities of types V_J^e and V_J^s, we check implications of the form $(\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i \in 1..n} \mathbf{GF}J_i) \rightarrow_{\text{LTL}} \mathbf{GF}J^{imp}$. This implication is equivalent to the unsatisfiability of $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i \in 1..n} \mathbf{GF}J_i \wedge \mathbf{FG}\neg J^{imp}$, which is reducible to the Buchi winning region problem, by computing states $R(\rho, W(\rho|_{\neg J^{imp}}, \{J_i\}_{i=1}^n))$, and checking that they do not intersect θ, which yields Alg. 4. We denote by $\rho|_{\neg J^{imp}}$ the restriction of ρ to states in $\neg J^{imp}$, namely $\{(s, s') \in \rho | s \notin J^{imp}\}$.

Algorithm 5 Vacuities Finds all vacuities of a specification.

Require: $Spec = \langle V_e, V_s, D, \langle I_e, T_e, J_e \rangle, \langle I_s, T_s, J_s \rangle \rangle$
Ensure: returns the set of all vacuities of $Spec$
1: $\theta_{Spec} \leftarrow \wedge_{\varphi \in I_e \cup I_s} \varphi$
2: $\rho_{Spec} \leftarrow \wedge_{\varphi \in T_e \cup T_s} \varphi$
3: **if** $\neg$**Sat**$(\theta_{Spec}, \rho_{Spec}, J_e \cup J_s)$ **then** ▷ Use Alg. 3
4: return $\{\bot\}$
5: **end if**
6: $vac \leftarrow \emptyset$
7: **for** $\theta \in I_e \cup I_s$ **do**
8: **if** $(\wedge_{\alpha \in ps(\theta)} \alpha) \rightarrow cons(\theta)$ **then** ▷ See Tbl. 1 for ps and $cons$ def.
9: Add θ to vac
10: **end if**
11: **end for**
12: **for** $\rho \in T_e \cup T_s$ **do**
13: **if** $(\wedge_{\alpha \in ps(\mathbf{G}\rho)} \alpha) \rightarrow cons(\mathbf{G}\,\rho)$ **then** ▷ See Tbl. 1 for ps and $cons$ def.
14: Add $\mathbf{G}\rho$ to vac
15: **end if**
16: **end for**
17: **for** $j \in J_e \cup J_s$ **do**
18: $PSJ \leftarrow \begin{cases} J_e \setminus \{j\} & j \in J_e \\ J_e \cup J_s \setminus \{j\} & j \in J_s \end{cases}$
19: **if** **ImpJust**$(\theta_{Spec}, \rho_{Spec}, PSJ, j)$ **then** ▷ Use Alg. 4
20: Add $\mathbf{GF}j$ to vac
21: **end if**
22: **end for**
23: **for** $var \in V_e \cup V_s$ **do**
24: **for** $value \in D(var)$ **do**
25: $dv \leftarrow \mathbf{G}\, var \neq value$
26: **if** $(\wedge_{\alpha \in ps(dv)} \alpha) \rightarrow cons(dv)$ **then** ▷ See Tbl. 2 for ps and $cons$ def.
27: Add dv to vac
28: **end if**
29: **end for**
30: **end for**
31: return vac

5.2 Putting it All Together: Detecting All Vacuities in a Given Specification

Algorithm 5 finds all vacuities in its input specification *Spec*. If the specification is unsatisfiable, we return $\bot$ as the only vacuity (lines 3-5, see Def. 4). Otherwise, we go through all initial assertions (lines 7-11), safety assertions (lines 12-16), justices assertions (lines 17-22), and domain values (lines 23-30), and check for vacuities according to vacuity types defined in Tables 1 and 2.

5.3 Correctness and Complexity

The correctness for detecting initial assertions, safeties, and domain vacuities is immediate.

For the correctness of detecting unsatisfiability and justice vacuities, note that Alg. 1 and 2 are symbolic implementations of well-known algorithms [21]. Alg. 3 is correct because Alg. 2 is correct, and because it verifies that there are states in θ from which we can visit all justices infinitely often. In Alg. 4, we use a direct reduction of the implication of justice J^{imp} to the generalized Buchi condition. Lemma 1 proves the correctness of the reduction.

Lemma 1. *Let θ, J_i for all $1 \leq i \leq n$, and J^{imp} be propositional formulas over variables $\mathcal{X}$, and let ρ be a propositional formula over $\mathcal{X} \cup \mathcal{X}'$. Then*[4], $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i=1}^n \mathbf{GF}J_i \rightarrow_{LTL} \mathbf{GF}\, J^{imp}$ *iff* $R(\rho, W(\rho|_{\neg J^{imp}}, \{J_i\}_{i=1}^n)) \cap \theta = \emptyset$.

[4]We use set and logic notations interchangeably here, e.g., θ is both a formula and the subset of $2^{\mathcal{X}}$ where the formula holds.

Proof. $R(\rho, W(\rho|_{\neg J^{imp}}, \{J_i\}_{i=1}^n)) \cap \theta \neq \emptyset$ iff there is a computation σ such that $\sigma \models \theta$, and $\sigma \models \mathbf{G}\rho$ and $\sigma \models \mathbf{FG}\neg J^{imp} \equiv_{\text{LTL}} \neg\mathbf{GF}J^{imp}$ and $\sigma \models \mathbf{F}(\bigwedge_{i=1}^n \mathbf{GF}J_i) \equiv_{\text{LTL}} \bigwedge_{i=1}^n \mathbf{GF}J_i$ iff there is a computation σ such that $\sigma \models \theta \wedge \mathbf{G}\rho \bigwedge_{i=1}^n \mathbf{GF}J_i$, and $\sigma \not\models \mathbf{GF}J^{imp}$ iff $\theta \wedge \mathbf{G}\rho \wedge \bigwedge_{i=1}^n \mathbf{GF}J_i \nrightarrow_{\text{LTL}} \mathbf{GF}\ J^{imp}$ □

Finally, Alg. 5 is correct because it checks all vacuity types according to their definition.

In terms of complexity, measured in symbolic steps, detecting initial assertions, safeties, and domain vacuities requires a number of symbolic operations linear in the number of elements in the specification $E = |B_{M_e} \cup B_{M_s}|$. Alg. 1 takes $O(N)$ and Alg. 2 takes $O(N^2)$, so Alg. 3 and 4 take $O(N^2)$. For Alg. 5, the complexity, as measured in symbolic steps, is $O(E^2)$ in lines 7-16, $O(E+\Sigma_{var \in V_e \cup V_s}|D(var)|)$ in lines 23-30 (the conjunction of the only two premise-sets requires $O(E)$ symbolic operations to compute), and $O((m+n)N^2)$ in lines 3-5 and 17-22. This is comparable and slightly better than the $O(mnN^2)$ complexity of realizability checking (see Sect. 3.2).

6 VACUITY CAUSE LOCALIZATION

The automatic detection of a vacuity of any type allows the engineer to know that her specification has a redundant element. However, knowing that an element of a specification is redundant may not be enough. To better understand the detected vacuity, and decide what to do with it, we provide the engineer with a means to localize the cause of the vacuity.

Localizing the Cause of Vacuities: Vacuity Core. Given a vacuity, we define its core to be a locally minimal subset of its premise-set that implies its consequent. Formally:

Definition 5 (Vacuity core). *Given a vacuity v, a vacuity core of v is a subset $\Gamma \subseteq ps(v)$ such that $(\wedge_{\alpha\in\Gamma}\alpha) \rightarrow_{LTL} cons(v)$, but for any strict subset $\Gamma' \subset \Gamma$, $(\wedge_{\alpha\in\Gamma'}\alpha) \nrightarrow_{LTL} cons(v)$.*

To compute a vacuity core, we apply DDMin [44], which finds a locally minimal subset of a set for a given monotonic criterion. For each of the vacuities found in lines 4, 9, 14, 20, or 27 of Alg. 5, we minimize its premise-set. The criterion method for minimization is implemented using BDD implication for vacuities found in lines 9, 14 and 27, and using Alg. 3 and Alg. 4 for vacuities found in lines 4 and 20, resp. Unsatisfiability can be viewed as LTL implication of $\bot$, and is thus consistent with our framework. Importantly, note that implication is monotonic in the sense that if v is implied by the conjunction of the elements in Γ, v is also implied by the conjunction of elements in any superset of Γ. This satisfies the monotonicity requirement of the DDMin algorithm.

Example 2. *Recall the vacuity in line 36 of the specification in List. 1. This is a vacuity of type V_J^s. We detect it by checking that it is implied by all other elements of the specification (its premise-set). Then, using the DDMin algorithm, we find that it is implied by the specification elements in lines 24, 35 and 37, and that it is not implied by any strict subset of this set of elements. Thus, this set of three elements constitutes a core for this vacuity.*

A vacuity core is a local minimum. The same vacuity may have different cores of different sizes, and DDMin finds one of them. That said, a vacuity core is indeed a sufficient and necessary subset of the specification that implies the detected vacuity.

Finally, in Sect. 8 we empirically show that vacuity cores are in many cases very effective in localizing the cause of vacuities.

Trivial Vacuities. A vacuity may be trivial, in the sense that it does not depend on any other element in the specification. A simple example for a trivial vacuity is a tautology, e.g., the guarantee $\mathbf{GF}\ \neg(x \leq 3 \ \wedge \ x \geq 4)$. This would be a vacuity in any specification. Interestingly, however, some trivial vacuities are not tautologies. For example, the guarantee in List. 2 is a trivial vacuity for the range 1..3 of f, but it would not be a vacuity at all if the range was 0..3.

Checking whether a vacuity is trivial is done by checking whether the BDD that represents the assertion of the element, and takes into account domain values, is $\top$. An exception is a justice that represents a pattern, see Sect. 7.3.

7 EXTENSIONS

7.1 Vacuities in Unrealizable Specifications

Unrealizability is a well-known problem in specifications for synthesis. Many specifications written during a development process may be unrealizable. Do unrealizable specifications include vacuities?

Defining vacuity for unrealizable specifications requires a careful discussion of the semantics. The semantics of a specification for reactive synthesis is typically defined as its set of implementations [20]. Accordingly, all unrealizable specifications, which have no implementations, are equivalent. Thus, this semantics is not useful for inherent vacuity of unrealizable specifications. Our approach is different.

As we presented in Sect. 3.2, the semantics we use is finer, and is defined not as the set of implementations of φ^{sr} but as the set of computations satisfying φ^{sr}. Since our definition of vacuity preserves the LTL equivalence of φ^{sr} (see Theorems 1 and 2), it transfers seamlessly to the unrealizable case: removing a vacuous element from an unrealizable specification does not change the set of counter-strategies [25, 35] it induces. Intuitively, this means that two such specifications, before and after removal of a vacuity, are different neither in terms of how close they are to realizability, nor by their reasons for unrealizability.

For example, the specification in List. 1, which we have borrowed from [2], is unrealizable, yet all the vacuities we detect, and their causes, may contain valuable information for the engineer. Specifically, without the guarantees in lines 35-37, which cause the vacuities in lines 30-32, the specification becomes realizable, and, in this case, free of vacuities.

Finally, in Sect. 8 we empirically show that many unrealizable specifications taken from benchmarks that appeared in the literature include vacuities.

7.2 Dealing with Auxiliary Variables

GR(1) has been extended with past LTL formulas (already in [10]), patterns [30], and monitors [33]. The extensions are useful and may help engineers write better specifications but their use introduces auxiliary variables that do not appear in the original specification. Auxiliary variables may also be added explicitly.

It is thus important to note that auxiliary variables do not hinder our ability to detect vacuous specification elements, check satisfiability, or compute vacuity cores. Our only special treatment for auxiliary variables is to not check them for domain vacuities, as we consider this case to be irrelevant.

7.3 Dealing with Auxiliary Elements

Patterns [16, 30] allow the engineer to easily express many useful temporal properties as additional assumptions or guarantees, although their LTL formulation is not in GR(1). For example, a response pattern is an LTL formula of the form $response(p, q) = \mathbf{G}\,(p \rightarrow \mathbf{F}\,q)$, which means that whenever p occurs, q will eventually occur as well. It is thus important to consider vacuities in specifications that include patterns.

Defining and detecting vacuity for specifications that use patterns is challenging, because their reduction to GR(1), following [30], adds not only auxiliary variables, but also auxiliary elements to the specification, i.e., additional initial, safety, and justice assumptions or guarantees. These additional elements are not explicit in the specification as written by the engineer and thus any vacuity that may be related to them should not be computed or reported at their level but rather at the level of the pattern that induced them, as written by the engineer.

Specifically, each pattern induces three GR(1) specification elements - a justice which is implemented as a part of the respective module, and two auxiliary elements, one is an initial assertion, and one is a safety.

Our approach handles auxiliary elements and patterns for all aspects of the vacuities we defined: considering patterns as vacuities, inclusion in premise-sets, and vacuity core computations. We make the special treatment for patterns, as detailed below, fully transparent to the engineer.

7.3.1 Patterns as vacuities. For patterns we always use the justice induced by the pattern as the consequent, and its auxiliaries are always a part of its premise-set. Thus, for example, the formula $response(p, p)$, which is a trivial vacuity, is detected as such because the auxiliary elements of the pattern imply its justice in this case.

7.3.2 Premise-sets in the presence of patterns. When we form premise-sets for vacuity element computation, we detect the module of each auxiliary element by tracing to the pattern that induced it. We then classify it based on the premise-set definition. Premise-sets of initial and safety elements (including unreachable domain values) ignore auxiliary elements in their premise-sets.

7.3.3 Vacuity cores in the presence of patterns. When a pattern is a part of a premise-set of another specification element, we bundle all its three elements for core computation. Bundling ensures that the auxiliary elements are counted as one element of the premise-set and are not separated when the premise-set is split into parts by the DDMin algorithm.

Example 3. *List. 3 shows a specification for which our tool detects two vacuities and finds each of their cores.*

One vacuity is the response pattern guarantee `gamma` *in line 7. We detect it because the justice induced by it (which is its consequent) is implied by the premise-set that includes nine elements, namely the two auxiliary elements induced by* `gamma`*, the safety* `alpha` *in line 5, and the six elements induced by the patterns* `beta` *and* `delta` *in lines 6 and 8 respectively. The core computation for this vacuity works on a set that has three elements, namely* `alpha`*, the three elements of* `beta` *bundled together, and the three elements of* `delta` *bundled as well. It finds that given the two auxiliary elements induced by* `gamma`*, which are not considered for minimization,* `alpha` *and* `delta` *(but not any of them alone), are together enough to imply the justice induced by* `gamma`*. Thus, the tool reports* `alpha` *and* `delta` *as a vacuity core of* `gamma`*.*

Another vacuity is the response pattern guarantee `delta` *in line 8. It is a vacuity because of the assumption* `beta` *and the guarantee* `gamma` *in lines 6 and 7 respectively. We detect this vacuity because the justice induced by* `delta` *(which is its consequent) is implied by the premise-set that includes nine elements - the element* `alpha`*, all six elements induced by* `beta` *and* `gamma`*, and the two auxiliary elements induced by* `delta`*. The core computation works on a set that has three elements, one of them is* `alpha`*, and the other two are each a bundle of the three specification elements induced by* `beta` *and by* `gamma`*. The two auxiliary elements of* `delta` *are inside the premise-set yet not considered for minimization. The DDMin algorithm detects* `beta` *and* `gamma` *as a vacuity core of* `delta`*.*

```
1 env boolean x;
2 env boolean y;
3 sys boolean z;
4
5 asm G x;                 // alpha
6 asm pRespondsToS(x, y); // beta
7 gar pRespondsToS(y, z); // gamma, a vacuity
8 gar pRespondsToS(x, z); // delta, a vacuity
```

Listing 3: Response patterns and cores example

8 EVALUATION

We have implemented detection for the different types of vacuity as an extension of Spectra [33, 42], based on CUDD [41] as a BDD library. Our implementation includes also the computation of vacuity core, as an instance of the generic DDMin algorithm implemented in Spectra (with the performance heuristics described in [19]).

Means to run our implementation, all specifications used in our evaluation, and all data we report on below, are available in supporting materials for inspection and reproduction [1]. We encourage the interested reader to try them out.

The following research questions guide our evaluation.

R1 Does vacuity appear in specifications, and are different types more frequent than others?

R2 Can vacuity be computed efficiently during development?

R3 Does vacuity core computation effectively localize its cause?

R4 Does vacuity removal make controller synthesis faster?

Below we report on the experiments we have conducted in order to answer the above questions.

8.1 Corpus of Specifications

We use the benchmark `SYNTECH15` [19], which includes a total of 78 specifications of 6 autonomous Lego robots, written by 3rd year undergraduate computer science students in a project class taught by the authors of [19]. Out of the 78 specifications in `SYNTECH15`, we use 14 unrealizable ones, which we label `SYN15U`, and all 61 realizable ones, which we label `SYN15R`. A similar benchmark from

Table 3: Number of vacuities in SYNTECH specifications

Spec set	S	H	$V_\perp$	V_I^e	V_S^e	V_J^e	V_I^s	V_S^s	V_J^s	V_D^e	V_D^s
SYN15U	14	11	2	0	0	8	3	19	8	0	7
SYN15R	61	51	8	0	0	30	5	102	30	0	11
SYN17U	26	19	4	1	8	22	0	9	16	0	29
SYN17R	118	99	5	15	78	153	0	45	75	21	157

the same authors, SYNTECH17 [33, 42] has 149 specifications. We use all 26 unrealizable specifications, which we label SYN17U, and 118 realizable ones, which we label SYN17R. [5]

We further use 4 different sizes of AMBA [8] (1 to 4 masters), 4 realizable and 12 unrealizable specifications (4 in each of the 3 variants of unrealizability described in [15]). Finally, we use 4 different sizes of GENBUF [9] (5 to 30 senders), 4 realizable and 12 unrealizable specifications (4 in each of the 3 variants of unrealizability described in [15].

8.2 Validation

We have implemented an automatic test that removes vacuities found by our algorithm (or adds a domain restriction element for domain vacuities, see Sect. 4.2), and checks that realizability, satisfiability of both modules, satisfiability of the specification, and well-separation all remain unchanged. The test is reproducible [1], and extensible to any specification in Spectra format. We have also examined dozens of vacuities and their cores and manually checked that their cores imply them and that all of the core elements are indeed required for the implication. These tests have increased our confidence in the validity of our computations.

8.3 Experiments Setup

We run all experiments on an ordinary PC, Intel Xeon W-2133 CPU 3.6GHz, 32GB RAM with Windows 10 64-bit OS, Java 8 64Bit, and CUDD 3 compiled for 64Bit, using only a single core of the CPU.

Times we report are average values of 10 runs, measured by Java in milliseconds. Even though the algorithms we deal with are deterministic, we performed 10 runs since JVM garbage collection and BDD dynamic-reordering add variance to running times.[6]

8.4 Results: Number of Vacuities

Table 3 presents the number of vacuities found in SYNTECH sets of specifications. For each set, column S shows the number of specifications in the set and column H shows the number of specifications that have at least one vacuity. All other columns show the number of vacuities per type. We do not detect (and do not report in the table) the vacuities in unsatisfiable specifications (see Sect. 4.3).

The results show that vacuities are very frequent in SYNTECH specifications. 82% of specifications include at least one vacuity. We observe that the most frequent vacuities are of types V_J^e, V_J^s and V_D^s. Unsatisfiable specifications are rare.

AMBA and GENBUF specifications are all satisfiable and contain no domain value vacuities and no trivial vacuities. This is expected because all the variables are Boolean and because the specifications were carefully written by experts. Interestingly, the specifications do contain many vacuities of type V_S^s and some of type V_J^s. For example, the realizable AMBA specification with 4 masters has 4 vacuities of type V_S^s, and 1 vacuity of type V_J^s.

[5]The above sets exclude 3 unrealizable SYNTECH15 specifications, and 5 realizable SYNTECH17 specifications that are not GR(1) specifications (some have system variables in initial assumptions, and some have primed system variables in safety assumptions).
[6]Since BDD-based implementations' performance is sensitive to variable order, we note that in all our experiments we used CUDD's automatic variable reordering. This is common practice in the literature.

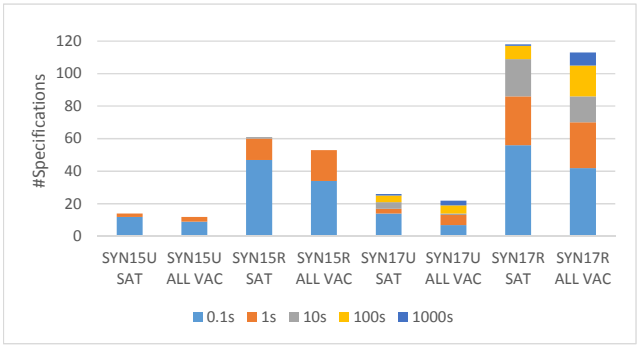

Figure 1: Running times of satisfiability checks (left columns) and of finding all vacuities of satisfiable specifications (right columns), for the SYNTECH sets, grouped by increasing ranges, in seconds.

To answer R1: Vacuities occur frequently in specifications from the literature. Over 82% of SYNTECH specifications have at least one vacuity. Vacuities of types V_J^e, V_J^s, and V_D^s are the most frequent. All AMBA and GENBUF specifications have vacuities.

8.5 Results: Running Times

Figure 1 shows the running times for all four SYNTECH sets of specifications. Each set has two columns showing the number of vacuities found up to a growing time limit starting from 0.1 seconds. The left hand column reports running times of satisfiability checks. The right hand column reports running times needed in order to find all other vacuities of satisfiable specifications.

The results show that for SYN15 specifications, computing satisfiability takes less than 1 second, and finding all vacuities is always done in less than 10 seconds. For SYN17 specifications 100 seconds are enough in order to compute satisfiability for all but 2 specifications out of 144, and in order to find all vacuities for 124 out of 135 satisfiable specifications.

Running times for detecting all vacuities in the 16 AMBA specifications were on average 0.1, 0.8, 18.1, and 304.5 seconds for 1 to 4 masters resp. Running times for detecting all vacuities in the 16 GENBUF specifications were on average 0.3, 2.2, 58.3, 480 seconds for 5 to 30 senders resp. This growth in running times is expected, as the state space in these benchmarks increases exponentially (e.g., in AMBA, by a factor of 64 for each additional master), and as the number of elements in the specification increases linearly (e.g., in AMBA, about 30 new elements for each additional master).

To answer R2: Satisfiability and finding all vacuities, is reached within 10 seconds for all of SYNTECH15 specifications. Within 100 seconds we compute satisfiability and all vacuities for 98% and 91% of SYNTECH17 specifications, resp.

Table 4: Localization effectiveness

Spec set	T	Core size					Reduction ratio		
		1	2	3	4	≥ 5	1/8	1/4	1/2
SYN15U	1	11	18	5	4	8	63%	87%	100%
SYN15R	4	46	80	15	17	24	72%	91%	99%
SYN17U	9	41	23	3	5	8	79%	91%	99%
SYN17R	41	294	120	66	16	12	81%	92%	98%

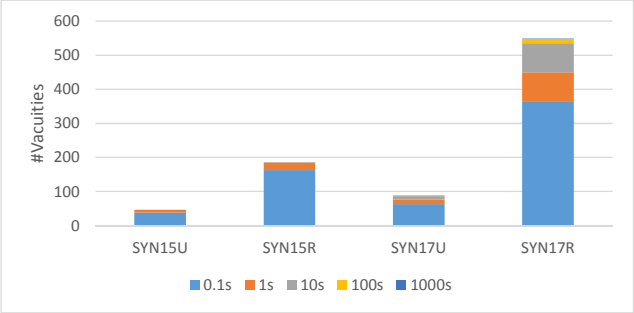

Figure 2: Running time of vacuity localization for SYNTECH sets, grouped by increasing ranges, in seconds.

8.6 Results: Localization Effectiveness

Table 4 reports localization results for SYNTECH specifications.

Given a specification set, for each vacuity we first check if it is a trivial vacuity. For non-trivial vacuities, we find a core (see Sect. 6).

We report under column T the number of trivial vacuities found in each specification set. For non-trivial vacuities, we report the number of vacuities for each core size, with the last column reporting cores of size five or more[7]. For example, the number 80 in the second row under column 2 means that in the SYN15R set, 80 vacuities had a core of size 2.

We further report under reduction ratio, the percentage of vacuities for which the ratio of the core size to the premise-set size[8], was at most 1/2, 1/4, and 1/8. E.g., the value 72% in the second row under column 1/8 means that in the SYN15R set, for 72% of the vacuities, the core size was at most 1/8 of the premise-set size.

The results show that the vacuity core size is less than half of the premise-set size for almost all vacuities, is less than a quarter for more than 87% of the vacuities, and is less than an eighth for more than 63%. Thus, a vacuity core is usually very effective in localizing the cause of the vacuity. In absolute terms, cores of sizes 1 and 2 are the most frequent. Large cores of size 5 and higher exist, but are relatively rare.

Figure 2 shows running times for core computations on SYNTECH specifications. For SYNTECH15 specifications, all core computations take less than 10 seconds. For each of the SYNTECH17 specification sets, all cores are computed within 100 seconds, except one.

[7]Note that cores are a local minimum and our algorithm sometimes finds different cores with different sizes, so we report the median size of the core on 10 runs.
[8]We count patterns as single elements in the premise-set and in the core (see Sect. 7.3)

Table 5: Synthesis Time Reduction

Spec set	#V	V_I^e	V_S^e	V_J^e	V_I^s	V_S^s	V_J^s
SYN15R	54	–	–	7.1%	–	1.4%	25.2%
SYN17R	322	0%	0%	7%	–	0%	32.2%

To answer R3: Vacuity core computations are effective and efficient. In SYNTECH sets, cores are eight times smaller than their respective premise-sets in over 63% of the vacuities. Most of the cores are of size 1 or 2. Core computations take less than 10 seconds for all SYNTECH15, and less than 100 seconds for all vacuities except one in each of the SYNTECH17 sets.

8.7 Results: Synthesis Time Reduction

Table 5 reports median percentage of synthesis running times reduction for all specifications in SYN15R and SYN17R for which original synthesis times in Spectra is over 0.1 seconds, broken down by vacuity types. Column #V shows the number of vacuities for which the original synthesis took over 0.1 seconds. Under each vacuity type we show the median percentage of synthesis running time reduction when the vacuity is removed from the specification. For example, the value 32.2% on the second row and rightmost column means that for SYN17R specifications whose original synthesis time in Spectra was over 0.1 seconds, for at least half of vacuities of type V_J^s, removing the vacuous justice guarantee reduced synthesis time by at least 32.2%. We use '-' to mark cases where there were no relevant vacuities for the vacuity type.

We observe that removing vacuous justice elements, for V_J^e and more so for V_J^s, makes synthesis running times faster. This is expected, as removing justices from the GR(1) winning condition reduces the computations the algorithm has to perform. These results are encouraging, since in realizable specifications, vacuities of these two types are very common (see Sect. 8.4). For other vacuity types, we observed only minor effect on running times, if any. Again, this is expected because, for example, removing vacuous elements of type V_I^e and V_S^e does not change the game's arena and winning conditions.

Results for realizable GENBUF and AMBA specifications, which only have vacuities of types V_S^s and V_J^s (see Sect. 8.4), show a median of 0% and 44% synthesis time reduction for these vacuity types, resp. This is consistent with the results for the SYNTECH specifications.

To answer R4: Removal of vacuities of types V_J^e and V_J^s have medians of 7%-7.1% and 25.2%-44% of synthesis running time reduction respectively. For other vacuity types we observe minor or no difference in synthesis running time.

8.8 Threats to Validity

We discuss threats to the validity of our results. First, symbolic computations are not trivial and our implementation may have bugs. To mitigate, we performed a thorough validation using all specifications available to us, see Sect. 8.2. Second, we have based most of our evaluation on the SYNTECH specifications, which were

created by 3rd year undergraduate CS students with no prior experience in writing LTL specifications. We further examined specifications from the AMBA and GENBUF sets. We do not know if these are representative of specifications engineers would write in practice. Third, we did not perform a user-study, with engineers, to examine whether users will find the reported vacuities useful for improving the quality of the specifications they write.

9 RELATED WORK

Vacuity in Model Checking. Vacuity has been widely studied in the context of model checking [4, 5, 13, 14, 23, 24, 28, 40]. In this context, when one checks whether a system satisfies a specification, vacuity usually means that some elements of the specification play no role in that satisfaction. Many different definitions for vacuity exist in this context, and they cover different cases. A classic example for vacuity in model checking is the case where a request-grant assertion $\mathbf{G}(r \rightarrow \mathbf{X}g)$ is satisfied by a system S because $S \models \mathbf{G}\neg r$, i.e., because a request is never made.

Note that in the context of model checking, vacuity is a property of a specification and a model. The model may satisfy the specification in a vacuous way. Our context is fundamentally different. We deal with inherent vacuity, which is a property of the specification alone.

Some empirical evidence from IBM showed that during the first verification run of a new design, 20% of formulas are found to be trivially valid, and that trivial validity always points to a real problem in either the design or its specification or environment [5]. Similar empirical evidence for the context of synthesis does not exist. Our work provides a step towards collecting such evidence.

Vacuity checks are now a standard component in commercial model checkers [14]. We believe that they will become standard components in future synthesizers.

Vacuity in Specifications for Synthesis. Fisman et al. [20] proposed a comprehensive theoretical framework for vacuity in LTL specifications, called inherent vacuity. The framework defines different aspects for vacuity, relating, e.g., to the setting (closed vs. open systems), and to tightening the specification or cleaning it (strengthening/weakening vs. equivalence). Thus, they do not offer a single definition of inherent vacuity, but, rather, a general theoretical parametrized framework. Finally, they proved that inherent vacuity detection can be reduced to LTL satisfiability or realizability checks, whose complexity is PSPACE-complete and 2EXPTIME-complete, respectively.

Our work is inspired by [20], but follows a pragmatic approach, takes advantage of the structure of GR(1) specifications, and defines a specific set of practical cases of inherent vacuity for GR(1) that allow efficient detection (see Sect. 5.3). Our setup considers open systems and LTL equivalence. The use of equivalence allows us to capture vacuities for both realizable and unrealizable specifications. To our knowledge, our work is the first to define, implement, and evaluate vacuity detection for GR(1) in particular and for specifications for reactive synthesis in general.

Finally, Bloem et al. [6] present a technique to synthesize non-vacuous systems. Given a specification, they show how to synthesize a system that will satisfy the specification non-vacuously. Their work is thus very different from ours. It is not about inherent vacuity. As in the case of model checking, the vacuity they consider is a property of the specification and the (synthesized) model.

Other Quality Aspects of GR(1) Specifications. Some works considered other quality aspects of GR(1) specifications. One example is the detection, explanation, and repair of unrealizability, see, e.g., [11, 25, 29, 34]. Another example is well-separation [32], which detects cases in which the synthesized controller may satisfy the specification by preventing the environment from satisfying the assumptions, without satisfying the guarantees. Finally, another example is the detection of assumptions that are not necessary for realizability [15, 17] or ones that are not weakest [12]. The last example may be viewed as a special, weakening/strengthening case (rather than equivalence case) of inherent vacuity [20].

10 CONCLUSION AND FUTURE WORK

We presented inherent vacuity definitions and algorithms for GR(1) specifications. The types of vacuity we detect include vacuous elements and vacuous domain values. We extended our work with vacuity core, which localizes the cause of vacuity.

We implemented our work, validated its correctness, and evaluated it on benchmarks from the literature, including more than 200 realizable and unrealizable specifications of autonomous Lego robots from the SYNTECH benchmarks. The evaluation shows that the different types of vacuity we consider indeed occur in specifications, that we detect the different types of vacuities efficiently in acceptable times, and that vacuity core is effective in localizing the cause of vacuity. It further shows that removal of vacuous elements from a specification may significantly reduce synthesis times. To the best of our knowledge, our work is the first to define, examine, and evaluate vacuity for GR(1) specifications.

Our work has important implications to anyone using GR(1) specifications for synthesis. First, we observe that vacuities are common and their presence may hint at problems in the specification. Second, we show evidence that removing vacuities of types V_J^e and V_J^s may significantly reduce synthesis time. Thus, as we provide efficient means for vacuities detection and localization, it is recommended to examine them and consider their removal.

We suggest future research as follows. First, consider a finer-grain analysis, down to the level of sub-formulas, i.e., detecting vacuities within specification elements. Second, Amram et al. have recently defined GR(1)* [3], an extension of GR(1) specifications with existential guarantees. GR(1)* expressive power is strictly beyond that of LTL, so our present work cannot be applied to it as is. It would be interesting to extend our work on inherent vacuity to GR(1)*.

ACKNOWLEDGEMENTS

We thank Jan Oliver Ringert and Or Pistiner for helpful discussions and advice. 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).

REFERENCES

[1] [n.d.]. Supporting materials website. http://smlab.cs.tau.ac.il/syntech/vacuity/.

[2] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2013. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*. IEEE, 26–33. http://dx.doi.org/10.1109/FMCAD.2013.6679387

[3] Gal Amram, Shahar Maoz, and Or Pistiner. 2019. GR(1)*: GR(1) Specifications Extended with Existential Guarantees. In *Formal Methods (FM) (LNCS)*, Vol. 11800. Springer, 83–100. https://doi.org/10.1007/978-3-030-30942-8_7

[4] Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer, and Moshe Y. Vardi. 2003. Enhanced Vacuity Detection in Linear Temporal Logic. In *CAV (LNCS)*, Vol. 2725. Springer, 368–380. https://doi.org/10.1007/978-3-540-45069-6_35

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. 2001. Efficient Detection of Vacuity in Temporal Model Checking. *Formal Methods in System Design* 18, 2 (2001), 141–163. https://doi.org/10.1023/A:1008779610539

[6] Roderick Bloem, Hana Chockler, Masoud Ebrahimi, and Ofer Strichman. 2017. Synthesizing Non-Vacuous Systems. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and David Monniaux (Eds.), Vol. 10145. Springer, 55–72. https://doi.org/10.1007/978-3-319-52234-0_4

[7] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. 2010. RATSY - A New Requirements Analysis Tool with Synthesis. In *CAV (LNCS)*, Vol. 6174. Springer, 425–429. https://doi.org/10.1007/978-3-642-14295-6_37

[8] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, Rudy Lauwereins and Jan Madsen (Eds.). EDA Consortium, San Jose, CA, USA, 1188–1193. https://dl.acm.org/citation.cfm?id=1266622

[9] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Specify, Compile, Run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.* 190, 4 (2007), 3–16. https://doi.org/10.1016/j.entcs.2007.09.004

[10] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.* 78, 3 (2012), 911–938. https://doi.org/10.1016/j.jcss.2011.08.007

[11] Davide G. Cavezza and Dalal Alrajeh. 2017. Interpolation-Based GR(1) Assumptions Refinement. In *TACAS (LNCS)*, Vol. 10205. 281–297. https://doi.org/10.1007/978-3-662-54577-5_16

[12] Davide Giacomo Cavezza, Dalal Alrajeh, and András György. 2018. A Weakness Measure for GR(1) Formulae. In *FM (LNCS)*, Vol. 10951. Springer, 110–128. https://doi.org/10.1007/978-3-319-95582-7_7

[13] Hana Chockler, Arie Gurfinkel, and Ofer Strichman. 2013. Beyond vacuity: towards the strongest passing formula. *Formal Methods in System Design* 43, 3 (2013), 552–571. https://doi.org/10.1007/s10703-013-0192-6

[14] Hana Chockler and Ofer Strichman. 2007. Easier and More Informative Vacuity Checks. In *MEMOCODE 2007*. 189–198. https://doi.org/10.1109/MEMCOD.2007.371225

[15] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. 2008. Diagnostic Information for Realizability. In *VMCAI (LNCS)*, Vol. 4905. Springer, 52–67. https://doi.org/10.1007/978-3-540-78163-9_9

[16] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *ICSE*. ACM, 411–420.

[17] Rüdiger Ehlers and Vasumathi Raman. 2014. Low-Effort Specification Debugging and Analysis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014. (EPTCS)*, Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha (Eds.), Vol. 157. 117–133. https://doi.org/10.4204/EPTCS.157.12

[18] Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) Synthesis. In *CAV (LNCS)*, Vol. 9780. Springer, 333–339. https://doi.org/10.1007/978-3-319-41540-6_18

[19] Elizabeth Firman, Shahar Maoz, and Jan Oliver Ringert. 2020. Performance heuristics for GR(1) synthesis and related algorithms. *Acta Inf.* 57, 1-2 (2020), 37–79. https://doi.org/10.1007/s00236-019-00351-9

[20] Dana Fisman, Orna Kupferman, Sarai Sheinvald-Faragy, and Moshe Y. Vardi. 2008. A Framework for Inherent Vacuity. In *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings (Lecture Notes in Computer Science)*, Hana Chockler and Alan J. Hu (Eds.), Vol. 5394. Springer, 7–22. https://doi.org/10.1007/978-3-642-01702-5_7

[21] Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Lecture Notes in Computer Science, Vol. 2500. Springer. https://doi.org/10.1007/3-540-36387-4

[22] Karin Greimel, Roderick Bloem, Barbara Jobstmann, and Moshe Y. Vardi. 2008. Open Implication. In *ICALP (LNCS)*, Vol. 5126. Springer, 361–372. https://doi.org/10.1007/978-3-540-70583-3_30

[23] Arie Gurfinkel and Marsha Chechik. 2004. Extending Extended Vacuity. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings (Lecture Notes in Computer Science)*, Alan J. Hu and Andrew K. Martin (Eds.), Vol. 3312. Springer, 306–321. https://doi.org/10.1007/978-3-540-30494-4_22

[24] Arie Gurfinkel and Marsha Chechik. 2012. Robust Vacuity for Branching Temporal Logic. *ACM Trans. Comput. Log.* 13, 1 (2012), 1:1–1:32. https://doi.org/10.1145/2071368.2071369

[25] Robert Könighofer, Georg Hofferek, and Roderick Bloem. 2013. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT* 15, 5-6 (2013), 563–583. https://doi.org/10.1007/s10009-011-0221-y

[26] Dexter Kozen. 1983. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.* 27 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

[27] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Trans. Robotics* 25, 6 (2009), 1370–1381. https://doi.org/10.1109/TRO.2009.2030225

[28] Orna Kupferman and Moshe Y. Vardi. 2003. Vacuity detection in temporal model checking. *STTT* 4, 2 (2003), 224–233. https://doi.org/10.1007/s100090100062

[29] Aviv Kuvent, Shahar Maoz, and Jan Oliver Ringert. 2017. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *ESEC/FSE*. 362–372. https://doi.org/10.1145/3106237.3106240

[30] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) synthesis for LTL specification patterns. In *ESEC/FSE*. ACM, 96–106. https://doi.org/10.1145/2786805.2786824

[31] Shahar Maoz and Jan Oliver Ringert. 2015. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015 (EPTCS)*, Vol. 202. 58–72. https://doi.org/10.4204/EPTCS.202.5

[32] Shahar Maoz and Jan Oliver Ringert. 2016. On well-separation of GR(1) specifications. In *FSE*. ACM, 362–372. https://doi.org/10.1145/2950290.2950300

[33] Shahar Maoz and Jan Oliver Ringert. 2019. Spectra: A Specification Language for Reactive Systems. *CoRR* abs/1904.06668 (2019). arXiv:1904.06668 http://arxiv.org/abs/1904.06668

[34] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. 2019. Symbolic repairs for GR(1) specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1016–1026. https://dl.acm.org/citation.cfm?id=3339632

[35] Shahar Maoz and Yaniv Sa'ar. 2013. Counter play-out: executing unrealizable scenario-based specifications. In *ICSE*. IEEE, 242–251. http://dl.acm.org/citation.cfm?id=2486821

[36] Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. 2011. Distributed power allocation for vehicle management systems. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference, CDC-ECC 2011, Orlando, FL, USA, December 12-15, 2011*. IEEE, 4841–4848. https://doi.org/10.1109/CDC.2011.6161470

[37] Necmiye Ozay, Ufuk Topcu, Richard M. Murray, and Tichakorn Wongpiromsarn. 2011. Distributed Synthesis of Control Protocols for Smart Camera Networks. In *2011 IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, Chicago, Illinois, USA, 12-14 April, 2011*. IEEE Computer Society, 45–54. https://doi.org/10.1109/ICCPS.2011.22

[38] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *VMCAI (LNCS)*, Vol. 3855. Springer, 364–380. https://doi.org/10.1007/11609773_24

[39] Leonid Ryzhyk and Adam Walker. 2016. Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*. 84–99. https://doi.org/10.4204/EPTCS.229.8

[40] Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik. 2010. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. *STTT* 12, 5 (2010), 319–335. https://doi.org/10.1007/s10009-009-0134-1

[41] Fabio Somenzi. [n.d.]. CUDD: BDD package, University of Colorado, Boulder. http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[42] Spectra [n.d.]. Spectra Website. http://smlab.cs.tau.ac.il/syntech/spectra/.

[43] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M. Murray. 2011. TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control (HSCC '11)*. ACM, New York, NY, USA, 313–314. https://doi.org/10.1145/1967701.1967747

[44] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498