On Well-Separation of GR(1) Specifications

Shahar Maoz
School of Computer Science
Tel Aviv University, Israel

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

ABSTRACT

Specifications for reactive synthesis, an automated procedure to obtain a correct-by-construction reactive system, consist of assumptions and guarantees. One way a controller may satisfy the specification is by preventing the environment from satisfying the assumptions, without satisfying the guarantees. Although valid this solution is usually undesired and specifications that allow it are called non-well-separated.

In this work we investigate non-well-separation in the context of GR(1), an expressive fragment of LTL that enables efficient synthesis. We distinguish different cases of non-well-separation, and compute strategies showing how the environment can be forced to violate its assumptions. Moreover, we show how to find a core, a minimal set of assumptions that lead to non-well-separation, and further extend our work to support past-time LTL and patterns.

We implemented our work and evaluated it on 79 specifications. The evaluation shows that non-well-separation is a common problem in specifications and that our tools can be efficiently applied to identify it and its causes.

CCS Concepts

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

Keywords

reactive synthesis, GR(1), well-separation, assumptions

1. INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [22]. Rather than manually constructing an implementation and using model checking to verify it against a 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 a controller, i.e., an automaton that accepts input from the environment (e.g., from sensors) and produces the system's output (e.g., commands for actuators) to always satisfy the specification.

One challenge for applying reactive synthesis to software engineering practice relates to the process of writing the specification, which typically consists of assumptions and guarantees. Assumptions play an important role as they describe the possible environments a system has to operate in. One way a controller may satisfy the specification is by preventing the environment from satisfying the assumptions, without satisfying the guarantees. Although valid, this vacuous solution to the reactive synthesis problem is usually undesired. Following Klein and Pnueli [13], we call specifications that allow this solution *non-well-separated*.

In this work we investigate non-well-separated specifications in the context of GR(1), a fragment of LTL, which has an efficient polynomial time symbolic synthesis algorithm [6, 21] and whose expressive power covers most of the well-known LTL specification patterns of Dwyer et al. [9, 16]. We distinguish different cases of non-well-separation, and compute strategies showing how the environment can be forced to violate its assumptions. Moreover, we show how to find a core, a minimal set of assumptions that lead to non-well-separation, and further extend our work to support past-time LTL and patterns.

Specifically, first, we present an algorithm for the diagnosis of non-well-separated environment specifications (see Sect. 4.1). The algorithm checks for well-separation and distinguishes several cases: it identifies whether the environment can be forced to violate its assumptions from all initial or only from some reachable states, and whether the safety or the liveness assumptions can be forced to be violated. The distinction between the different cases is important for the following strategy synthesis: in case our algorithm identifies non-well-separation, we synthesize strategies that demonstrate how the environment can be forced to violate its assumptions (see Sect. 4.2).

Second, we define and show how to compute a *non-well-separated core*, a minimal subset of assumptions that already makes the environment specification non-well-separated (see Sect. 5). As shown in our evaluation, see below, the core is indeed typically much smaller than the original set of assumptions, facilitating better focus on the reasons for non-well-separation.

Finally, we show in Sect. 6.1 how the above can correctly handle extensions of the specification language with past-time LTL [6] and LTL specification patterns [16]. We present

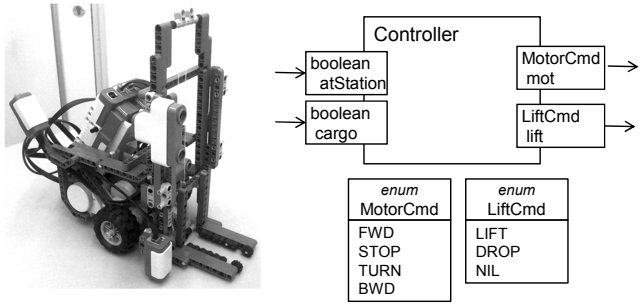

Figure 1: A forklift and its controller `Controller`

Specification

```
1  ASM findStat: -- always possible to find a station
2    G F (atStation);
3  ASM samePos: -- same station position when stopped
4    G (mot=STOP -> next(atStation)=atStation);
5  ASM liftCargo: -- lifting clears sensor
6    G (lift=LIFT -> next(!cargo));
7  ASM dropCargo: -- dropping senses cargo
8    G (lift=DROP -> next(cargo));
9  ASM clearCargo: -- backing up clears cargo
10   G (mot=BWD -> next(!cargo));
```

Listing 1: Excerpt of an environment specification for the forklift controller

further extensions in Sect. 6.

Together, our analyses serve as powerful debugging tools for specifications, which can assist in finding and understanding the reasons for non-well-separation in reactive synthesis specifications.

We have implemented the above in our GR(1) synthesis framework, on top of JTLV [23]. We present evaluation of the diagnosis algorithm and the non-well-separated core computation on 79 specifications created by students who took a six month project class on reactive synthesis. The evaluation shows that non-well-separation appears in many specifications, that our diagnosis algorithm is efficient, and that non-well-separated cores effectively reduce the set of assumptions one has to consider when trying to understand the reasons of non-well-separation. We describe the evaluation in Sect. 7.

Some previous works suggested criteria for environment specifications susceptible to being forced to violate assumptions [5, 8, 13]. Our work is partly inspired by these works. These works, however, did not consider algorithms and automated means for debugging the problematic environment specifications, computing cores etc., and did not evaluate the problem on a corpus of specifications. Unrealizability is another common problem of reactive systems specifications, which can be handled using (counter-)strategies and a notion of core. However, non-well-separation is very different than unrealizability. It requires the construction of strategies not only from all initial states but also from some reachable states. Moreover, its core, as we define it, is made of assumptions, not guarantees. We discuss related work in Sect. 8.

2. EXAMPLE

We start off with a running example, adapted from our specification of a Lego forklift, shown in Fig. 1[1], see [17]. The forklift has two sensors: one sensor to determine whether it is at a station and one sensor to detect cargo. It also has two motors, to drive the forklift and to lift the fork. Values read by the sensors are provided as inputs to component `Controller` and its outputs are commands that control the motors. All inputs and outputs are typed, e.g., the output `mot` has type `MotorCmd`. The datatypes are `boolean` or defined as enumerations in Fig. 1.

A team of engineers is writing a specification of the forklift controller to automatically synthesize an implementation. The main task of the forklift is to traverse an open area and always eventually deliver cargo; it finds cargo at stations, lifts it, and drops it at other stations. A benefit of synthesizing a controller is that it is guaranteed to satisfy its specification. However, without any environment assumptions a forklift controller cannot be synthesized. As an example, the forklift can not ensure that it will find a station to deliver cargo to because the sensors are completely controlled by the environment. To guarantee the completion of its task, the forklift has to assume that it will always find stations. This is expressed in the assumption `G F (atStation)` named `findStat` in Listing 1, ll. 1-2. The temporal operator `G` intuitively stands for always, i.e., at every state, and `F` stands for eventually, i.e., within finitely many steps.

Additional assumptions in Listing 1 describe reactions of the environment to actions of the forklift controller. The assumption `samePos` specifies that the value of the station sensor remains the same if the forklift stops: `G (mot=STOP -> next(atStation)=atStation)` (in other words, stations do not move). The temporal operator `next(v)` interprets `v` in the next time step; here, the next value of `atStation` must be equal to its current value. The next three assumptions follow the same pattern and restrict the expected environment behavior for handling cargo. When the forklift lifts cargo the cargo sensor is cleared (assumption `liftCargo`, l. 5). When it drops cargo the cargo is detected by the sensor (assumption `liftCargo`, l. 7). Finally, the forklift can clear the cargo by moving backward from it (assumption `clearCargo`, l. 9).

The engineers complete the specification consisting of assumptions and guarantees and successfully synthesize a controller that satisfies all guarantees if all assumptions hold. However, once the controller is deployed to the forklift the team observes strange behavior. Sometimes the forklift drops cargo and behaves chaotic. An engineer finds out that this happens when the forklift drives backwards while dropping cargo. Our new well-separation analysis informs her that the environment can be forced to violate safety assumptions from all initial states (diagnosed case (**P-all**, **E-safe**), see Sect. 1). Specifically, one of the assumptions `dropCargo` or `clearCargo` can be forced to be violated. She fixes the problematic assumptions by changing the assumption `dropCargo` from `G (lift=DROP -> next(cargo))` to `G (lift=DROP & mot!=BWD -> next(cargo))`.

After some more runs of the forklift the team observes that the forklift sometimes stops between stations and does not continue delivering cargo. It clearly does not continue to satisfy its guarantees, i.e., some assumption must be violated. Again, our new well-separation analysis informs the team that the environment can be forced to violate a justice assumption from some reachable states (diagnosed case (**P-reach**, **E-just**), see Sect. 1). The reason involves the as-

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

sumption to always eventually find a station `findStat` and the safety assumption `samePos` (which states that the station sensor reading does not change when motors are stopped, as described above). When the forklift is not at a station it stops and thus forces the environment to violate its liveness assumption `findStat`.

This example shows how non-well-separation can lead to unexpected behavior of a synthesized controller and how a small set of relevant assumption can be used to explain the reason for non-well-separation.

3. PRELIMINARIES

3.1 LTL and GR(1)

We repeat some of the standard definitions of linear temporal logic (LTL), e.g., as found in [6], 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 $\mathtt{X}$ (next) and $\mathtt{U}$ (until) and the past-time temporal operators $\mathtt{Y}$ (previous) and $\mathtt{S}$ (since).

Definition 1. *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 $u = u_0u_1.. \in \Sigma^\omega$ 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 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. This strategy can be represented as an automaton 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 [6] handles an assume-guarantee fragment of LTL where specifications contain assertions over initial states, safety constraints relating the current and next state, and justice constraints requiring that an assertion holds infinitely many times during a computation. A GR(1) synthesis problem consists of the following elements[6]:

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

We also write environment and system specifications as tuples $\langle \theta, \rho, J \rangle$. GR(1) synthesis has two different notions of realizability, which are expressed in the following LTL specifications [6]. The first and more intuitive notion is called implication realizability, because all environment assumptions imply all system guarantees:

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

The second kind of realizability is called strict realizability:

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

Realizability of φ^{sr} implies realizability of $\varphi^{\rightarrow}$. 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 for φ^{sr} have been presented in [6, 21]. The algorithm of Piterman et al. [21] computes winning states for the system, i.e., states from which the system can ensure satisfaction of φ^{sr}.

3.2 Well-Separation

Klein and Pnueli [13] defined well-separation as a sufficient property of environment specifications $\langle \theta^e, \rho^e, J^e \rangle$ such that realizability of φ^{sr} is equivalent to realizability of $\varphi^{\rightarrow}$. A well-separated environment can satisfy all assumptions from every reachable state. We repeat the definition of Klein and Pnueli adapted to our syntax in Def. 2.

Definition 2 (Well-Separation [13]). *A GR(1) environment specification $\langle \theta^e, \rho^e, J^e \rangle$, is well-separated iff φ^{sr} has no reachable system winning states for system specification* $\langle \mathtt{true}, \mathtt{true}, \{\mathtt{false}\} \rangle$.

Note that well-separation is defined as a property of the environment part of the GR(1) specification, i.e., the assumptions, without the guarantees. Intuitively, the system specification $\langle \mathtt{true}, \mathtt{true}, \{\mathtt{false}\} \rangle$ means that initially and for every step the system choices are unconstrained ($\theta^s = \mathtt{true} = \rho^s$) but its justice requirements cannot be satisfied $J^s = \{\mathtt{false}\}$. Winning strategies for φ^{sr} thus have to force the environment to violate its assumptions.

Klein and Pnueli [13] showed how to reduce implication realizability to strict realizability. For the system specification in Def. 2 strict realizability φ^{sr} and implication realizability $\varphi^{\rightarrow}$ both reduce to the LTL formula $\neg(\theta^e \wedge \mathtt{G}\rho^e \wedge \bigwedge_{i \in 1..n} \mathtt{GF}J^e_i)$. The set of states where the environment can be forced to violate its assumptions can thus be computed by the standard GR(1) algorithm.

4. DEBUGGING NON-WELL-SEPARATION

We start by arguing why well-separation is desired and why it is necessary to provide tools for debugging non-well-separated environment specifications. We then follow with an analysis of different cases of non-well-separation. These cases distinguish winning positions and environment specification parts that can be forced to violate. The cases on the one hand present an informative summary of well-separation of the specification and on the other require different means to further explain reasons for non-well-separation. We explain in Sect. 4.2 how strategies to demonstrate the different cases of non-well-separation can be computed.

Well-separation is a property of the environment specification, i.e., the assumptions in GR(1) synthesis. In most cases the environment is specified by the same engineer specifying the guarantees of the system[2]. Non-well-separation is a problem in the specifications due to two main reasons. First, controllers that force an environment to violate assumptions are undesired in general because they do not have to satisfy their guarantees. Second, we assume that the true environment (e.g., in the physical world) of the synthesis problem is well-separated, i.e., cannot be forced to violate its assumptions. Thus, it should not be possible to force an environment to a deadlock or prevent it from satisfying its justice assumptions. A non-well-separated environment thus points to a gap or a mismatch between the real environment and the assumptions describing it, i.e., it points to a problem in the specification.

Due to its assume-guarantee nature, the GR(1) synthesis algorithm might exploit non-well-separation and synthesize controllers that fail in a real environment. Note that according to Def. 2 an environment can also be non-well-separated without a malicious system, i.e., no matter what the system does the environment has to violate its assumptions. We consider this again to be an undesirable specification.

We now present approaches to debug non-well-separated environments by further distinguishing different cases of non-well-separation and presenting methods that assist engineers in understanding the reasons for non-well-separation.

4.1 Cases of Non-Well-Separation

We distinguish different cases of non-well-separation based on two different criteria: winning positions and environment specification parts.

4.1.1 Winning positions (all or reachable)

Well-separation in Def. 2 is defined based on winning states for the system specification $\langle \texttt{true}, \texttt{true}, \{\texttt{false}\}\rangle$, i.e., the only way for the system to win is to ensure assumption violations by the environment. Non-well-separation is weaker than realizability of φ^{sr} in Def. 2. Realizability of φ^{sr} implies non-well-separation. If the environment specification is realizable but not non-well-separated there exists a state that can be reached by environment choice from which the environment can be forced to violate its assumptions. To distinguish these two cases we define the positions from which a controller can force the violation of assumption:

P-all force violation from all initial environment choices;
P-reach force violation from some reachable state.

[2]Exceptions that we do not consider here are for example specifications derived from existing components.

The first case **P-all** means that a controller can always force the environment to violate its assumptions, i.e., it does not have to satisfy any guarantee. An example of this case are the two assumption `dropCargo` and `clearCargo` shown in Listing 1: a controller can always force a violation of either one. The second case **P-reach** means that during execution of a controller the environment can reach a state from which the controller can force it to violate its assumptions, i.e., until the state is reached, if ever, all guarantees have to be satisfied. An example for this case is the combination of assumptions `samePos` and `findStat` from a state with `station` $=$ `false`. It is easy to see that **P-all** implies **P-reach** but not the other way around.

4.1.2 Env. parts (initial, safety, or justice)

To express which part of the environment specification can be forced to be violated we distinguish the following cases:

E-ini all initial environment choices are invalid ($\theta^e = \texttt{false}$);
E-safe force violation of safety assumptions ρ^e (deadlock);
E-just force violation of some justice assumption $J_i^e \in J^e$.

Both cases of **E-ini** and **E-safe** imply the case of **E-just**, i.e., all non-well-separated environment specifications are of case **E-just**. An example for **E-safe** are the two assumptions `dropCargo` and `clearCargo` while the two assumptions `samePos` and `findStat` are only an example for case **E-just**.

4.1.3 Summarizing case combinations

When checking all environment parts and the positions from which each can be violated we obtain $2 * 3 * 3$ possible combinations of cases (**E-ini** may appear with **P-all** or be absent, while the other parts may appear with **P-all**, **P-reach**, or be absent). A combinations of cases might contain redundancies because one case implies another one, e.g., case (**P-all**, **E-safe**) implies (**P-all**, **E-just**) and we can summarize their combination as case (**P-all**, **E-safe**).

More formally, we summarize case combinations by defining the total orders **P-all** $\sqsubset_p$ **P-reach** and **E-ini** $\sqsubset_e$ **E-safe** $\sqsubset_e$ **E-just** and their product order on pairs, i.e., the partial order $(p_1, e_1) \sqsubset_{p\times e} (p_2, e_2) \Leftrightarrow p_1 \sqsubset_p p_2 \wedge e_1 \sqsubset_e e_2$. Finally, we represent a combination of cases by their smallest elements. This summarization of case combinations leaves 7 combinations shown in Fig. 2. The only time two cases are reported is for cases (**P-reach**, **E-safe**) and (**P-all**, **E-just**) because these cases are not comparable in the product order.

The representation in Fig. 2 is ordered from the weakest case combination at the top, i.e., well-separation, to the strongest case combination of non-well-separation {(**P-all**, **E-ini**)} at the bottom.

4.1.4 An algorithm to identify the cases

Algorithm 1 presents our algorithm to identify non-well-separation and the cases along the dimensions of winning positions and environment specification parts. Its input is an environment specification $\langle \theta^e, \rho^e, J^e\rangle$ and its output is a set of cases of non-well-separation. A case of non-well-separation is reported as a tuple of winning positions **P-all** or **P-reach** and responsible parts of the specification **E-ini**, **E-safe**, or **E-just**. The algorithm reports summarized strongest case combinations as discussed in Sect. 4.1.3 and shown in Fig. 2. Note that the algorithm returns an empty set iff the environment specification is well-separated.

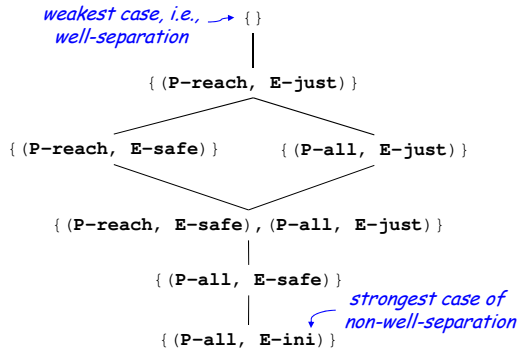

Figure 2: Cases of non-well-separation from weakest case, i.e., well-separation, to strongest case (P-all, E-ini); the cases (P-reach, E-safe) and (P-all, E-just) are incomparable

Algorithm 1 Diagnosing non-well-separation cases

```
1: if θ^e ≡ false then
2:   return {(P-all, E-ini)}
3: end if
4: define res as Set
5: reach ← reachStates(θ^e, ρ^e)
6: win_s ← sysWinSts(⟨θ^e, ρ^e, ∅⟩, ⟨true, true, {false}⟩)
7: if win_s ∩ reach ≠ ∅ then
8:   if sysWinAllIni(win_s, θ^e) then
9:     return {(P-all, E-safe)}
10:  end if
11:  add (P-reach, E-safe) to res
12: end if
13: win ← sysWinSts(⟨θ^e, ρ^e, J^e⟩, ⟨true, true, {false}⟩)
14: if win ∩ reach ≠ ∅ then
15:  if sysWinAllIni(win, θ^e) then
16:    add (P-all, E-just) to res
17:  else if res = ∅ then
18:    add (P-reach, E-just) to res
19:  end if
20: end if
21: return res
```

The algorithm starts by first checking case **E-ini** of non-well-separation by testing the satisfiability of θ^e. For $\theta^e =$ `false` the system wins from all positions. It then checks case **E-safe** by computing the environment reachable states in line 5 and the system winning states according to Def. 2 but for $J^e = \emptyset$ in line 6. For an empty set of justice assumptions a system can only win by violating safety assumptions. The method **sysWinSts**(.,.) returns all system winning states, i.e., the Z fix-point in the GR(1) algorithm of [6]. If any of the winning states is reachable, the environment is non-well-separated (l. 7). In case all initial states are winning for the system (l. 8) the strongest result is (**P-all**, **E-safe**). Otherwise the algorithm also checks the complete environment specification including J^e for possible system winning states. This second **E-just** part in lines 13-20 is analogous to the first part of case **E-safe** but the case (**P-reach**, **E-just**) is only added if *res* is empty, i.e., does not contain the stronger case (**P-reach**, **E-safe**) possibly added in line 11.

The time complexity of Algorithm 1 for $n = |J^e|$ and state space size N is in $O(nN^2)$ because it uses the GR(1) algorithm of [6] to compute `sysWinSts`(.,.) and its other operations `sysWinAllIni`(.,.) and `reachStates`(.,.) are in $O(N)$. Note that a simple algorithm to check well-separation according to Def. 2 without our diagnosis computes the sets *reach* (l. 5) and *win* (l. 13) and checks for an empty intersection (l. 14). The time complexity of this algorithm is also in $O(nN^2)$.

Applied to the example in Listing 1, Algorithm 1 reports {(**P-all**, **E-safe**)}. After replacing assumption `dropCargo` with its modified version, as suggested in the end of Sect. 2, the algorithm reports {(**P-reach**, **E-just**)}.

4.2 Strategies Forcing Assumption Violation

In addition to identifying and distinguishing the different cases of non-well-separation, as a means to further explain the reasons for non-well-separation to the engineer, we compute and present concrete strategies that demonstrate how a system can ensure assumption violations. Different non-well-separation cases require different strategy computations. What is common, is that all strategies are constructed from the game memory stored during realizability checking as implemented in the GR(1) algorithm and described in [6].

4.2.1 *Winning positions (all or reachable)*

In case the system can force an assumption violation from all initial states (case **P-all**) controller construction is the same as for regular GR(1) synthesis. A controller constructed for the system specification ⟨`true`, `true`, {`false`}⟩ shows how to force assumption violations.

The case **P-reach** and not **P-all** is more complicated. If **P-all** does not hold a controller does not exist because for some initial environment choices the environment cannot be forced to violate assumptions. However, from every state that is winning for the system — here those that can force assumption violations — a winning strategy exists and can be computed from the game memory.

The only difference during strategy computation is the treatment of initial states. In controller construction for case **P-all** the initial states are computed one for every initial environment choice. Strategy construction from reachable states in case **P-reach** starts with all reachable states, i.e., possibly multiple assignments to system variables for each assignment to environment variables.

Understanding reasons for non-well-separation for the case **P-reach** might require understanding how the states of the strategy can be reached from initial states. How these states are reached might already exhibit environment behavior that should be restricted by additional assumptions to make the environment well-separated. To show how a state can be reached we can simply compute a trace starting with an initial state and ending with a state in the strategy.

4.2.2 *Env. parts (initial, safety, or justice)*

In addition to the distinction between cases **P-all** and **P-reach** we distinguish the environment parts that can be forced to be violated.

Non-well-separation of case **E-ini** means that the initial assumptions of the environment are contradicting. In this case the system has nothing to do to force assumption violations so there is no need to discuss it further.

To obtain a controller for case **E-safe** that shows safety violations only, we have to use the modified environment specification $\langle \theta^e, \rho^e, \emptyset \rangle$. Otherwise the controller might chose to

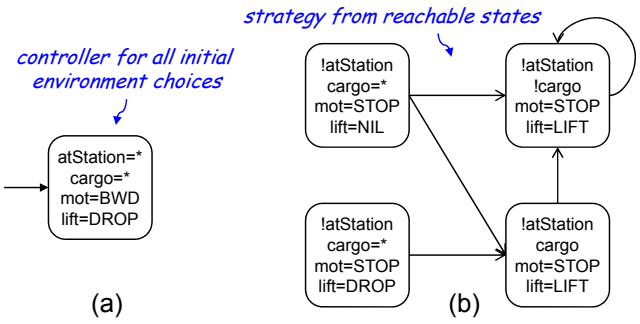

Figure 3: (a) A controller forcing violation of assumptions `clearCargo` or `dropCargo` for all initial environment choices (case (P-all, E-safe)) and (b) a strategy forcing violation of assumption `findStat` from reachable states (case (P-reach, E-just)), for the specification in Listing 1

violate a justice assumption in J^e instead. Strategies for case **E-safe** with environment specification $\langle \theta^e, \rho^e, \emptyset \rangle$ include only finite computations.

To obtain a strategy for case **E-just** we simply use the game memory computed for the check of Def. 2. Strategies for case **E-just** might include infinite computations.

4.2.3 *Examples*

The environment of the example of Sect. 2 shown in Listing 1 is non-well-separated of case (**P-all**, **E-safe**). A controller that forces assumption violations is shown in Fig. 3 (a). The initial choices of the environment are not constrained, i.e., the environment can choose all four possible assignments to the variables `atStation` and `cargo`. The controller has a single initial state labeled with assignments to environment and system variables. The symbol `*` denotes all possible values of a variable, i.e., the symbolic state in Fig. 3 (a) denotes four concrete states. The controller assigns `mot=BWD` forcing `!cargo` in the next state and `lift=DROP` forcing `cargo` in the next state, i.e., a contradiction and thus an assumption violation of `clearCargo` or `dropCargo` from Listing 1.

In the second part of the example of Sect. 2, the assumption `dropCargo` from Listing 1 was changed to `G (lift=DROP & mot!=BWD -> next(cargo))` to resolve the first non-well-separation case (**P-all**, **E-safe**). The modified environment is non-well-separated of case (**P-reach**, **E-just**). A strategy that forces it to violate assumptions from reachable states is shown in Fig. 3 (b). The strategy is only defined for states where the forklift is not at a station and stops. The strategy forces the environment to violate the justice assumption `findStat` by stopping forever and thus forcing the environment to always keep `atStation` set to `false`.

5. CORES OF NON-WELL-SEPARATION

Every environment specification $\langle \theta^e, \rho^e, J^e \rangle$ consists of a set of assumptions similar to the ones shown in Listing 1. The specification elements θ^e, ρ^e, and J^e might result from many assumptions, e.g., for the specification in Listing 1 ρ^e is the conjunction of assumptions `samePos`, `liftCargo`, `dropCargo`, and `clearCargo`. To further assist in debugging non-well-separated environments we compute a minimal subset of the assumptions that demonstrates a reason for non-well-separation. We call these minimal subsets non-well-separated cores.

5.1 The Importance of Monotonicity

The notion of a core has appeared in works that address unrealizability, see e.g. [14, 19]. These works rely on the monotonicity of unrealizability with regard to adding guarantees, to make the core definition meaningful and to allow its efficient computation. A definition of core with regard to non-well-separation is however challenging, because non-well-separation by itself is not monotonic with regard to adding assumptions. Without monotonicity, the reason for non-well-separation exhibited by a subset of assumptions is not necessarily a reason for non-well-separation of the original specification.

THEOREM 1 (NON-WELL-SEPARATION NOT MONOTONIC). *Non-well-separation is not monotonic wrt. adding or removing assumptions.*

PROOF. We show counter examples for both cases. (1) Non-well-separation is not preserved when removing assumptions: the non-well-separated environment specification consisting of justice assumption `findStat` and safety assumption `samePos` from Listing 1 becomes well-separated when removing assumption `samePos`. (2) Non-well-separation is not preserved when adding assumptions: the non-well-separated environment specification consisting of assumptions `findStat`, `samePos` becomes well-separated when adding the assumption `G (atStation)`. □

Def. 2 of well-separation includes both the reachable states and states winning for the system. This provides different possible resolutions of non-well-separation in an environment specification. Intuitively these are (1) weakening assumptions to remove system winning states and (2) strengthening assumptions to remove reachable states. Both cases are demonstrated in the proof of Theorem 1. It is important to note that these cases are not exclusive. The dependence on reachable states and winning states makes the property of non-well-separation not monotonic.

5.2 Non-Well-Separated Core

To address the non-monotonicity challenge, we provide a stronger definition of core, which relates to the reachable states of the original complete specification and ensures that the reason for non-well-separation exhibited by a core is a reason for non-well-separation of the original specification.

Thus, we define a non-well-separated core to be a minimal subset of a specification's assumptions which can be forced to be violated from the reachable states of the original specification. Given a set of assumptions `ASM` we denote the environment specification resulting from it by $\theta^e_{\texttt{ASM}}$, $\rho^e_{\texttt{ASM}}$ and $J^e_{\texttt{ASM}}$ in Def. 3.

DEFINITION 3 (NON-WELL-SEPARATED CORE). *A non-well-separated core for a set of assumptions `ASM` is a minimal set `C` $\subseteq$ `ASM` such that*

$$\mathit{sysWinSts}(\langle \theta^e_C, \rho^e_C, J^e_C \rangle, \langle \texttt{true}, \texttt{true}, \{\texttt{false}\} \rangle) \cap \mathit{reachStates}(\theta^e_{ASM}, \rho^e_{ASM}) \neq \emptyset.$$

Intuitively, we look for a minimal set of assumptions `C` that is non-well-separated within the reachable states for the original set of assumptions `ASM`. On the one hand this restriction is natural when debugging non-well-separation of `ASM` because it relates only to states relevant to the original

specification. On the other hand the restriction makes the check monotonic with respect to adding assumptions, i.e., for all $\texttt{C} \subseteq \texttt{C'} \subseteq \texttt{ASM}$ if C can be forced to violate its assumption in $\texttt{reachStates}(\theta^e_{\texttt{ASM}}, \rho^e_{\texttt{ASM}})$ then the same applies to C' (and ASM). This makes cores as defined in Def. 3 meaningful because the reason for non-well-separation in the core C is a reason for non-well-separation in ASM. We show this monotonicity in Theorem 2.

THEOREM 2 (CORE MONOTONIC). *The check for a non-well-separated core C in Def. 3 is monotonic with respect to adding assumptions from ASM.*

PROOF. A strategy to force violation of $\texttt{C} \subseteq \texttt{ASM}$ in $\texttt{reachStates}(\theta^e_{\texttt{ASM}}, \rho^e_{\texttt{ASM}})$ is also a strategy to force violation of C' with $\texttt{C} \subseteq \texttt{C'} \subseteq \texttt{ASM}$ in $\texttt{reachStates}(\theta^e_{\texttt{ASM}}, \rho^e_{\texttt{ASM}})$. Adding assumptions can only make the environment specification stronger, i.e., $\theta^e_{\texttt{C}'} \Rightarrow \theta^e_{\texttt{C}}$, $\rho^e_{\texttt{C}'} \Rightarrow \rho^e_{\texttt{C}}$, and $J^e_{\texttt{C}} \subseteq J^e_{\texttt{C}'}$. All C winning states in $\texttt{reachStates}(\theta^e_{\texttt{ASM}}, \rho^e_{\texttt{ASM}})$ are also C' winning states because their successor states in C are deadlocks or exist in C', i.e., if a strategy can force a deadlock in C it can force a deadlock in C' because $\rho^e_{\texttt{C}'} \Rightarrow \rho^e_{\texttt{C}}$. Otherwise there exists at least one $j \in J^e_{\texttt{C}} \subseteq J^e_{\texttt{C}'}$ that can be prevented by a strategy to violate assumptions C. A winning strategy for C also prevents j in C' because $\rho^e_{\texttt{C}'} \Rightarrow \rho^e_{\texttt{C}}$ and thus states in C' have the same ore less successor states not satisfying j. □

The proof of Theorem 2 also shows how to construct a strategy to force violation of $\langle \theta^e_{\texttt{ASM}}, \rho^e_{\texttt{ASM}}, J^e_{\texttt{ASM}} \rangle$ from the result of checking the GR(1) realizability of C.

5.3 Implementation and Example

We implemented the computation of non-well-separated cores in our synthesis framework based on JTLV [23] using the delta-debugging algorithm of Zeller [25] to check subsets of assumptions and compute minimal cores according to Def. 3. After core computation our tools can construct a strategy as described in Sect. 4.2 to demonstrate non-well-separation and present a minimal set of assumptions to the engineer.

For the example of Sect. 2 shown in Listing 1 our core calculation algorithm computes the core {liftCargo, clearCargo}. After the modification of assumption dropCargo as described in Sect. 2 the core calculation algorithm computes the core {samePos, findStat}.[3]

6. EXTENSIONS

6.1 Support for Patterns and Past-Time LTL

The LTL fragment of GR(1) is limited to initial constraints, safety constraints over the current and next state, and justice constraints over states to visit infinitely often. This is a very restricted subset of LTL. However, GR(1) synthesis is quite expressive because deterministic Büchi automata can be used as assumptions and guarantees [6]. This additional expressiveness is achieved by adding auxiliary variables to the GR(1) synthesis problem. In practice, this allows GR(1) specifications to include most LTL specification patterns of Dwyer et al. [9] as shown in [16] and past-time LTL as shown in [6].

[3]Actually, both checks returned the second core, which is also a core for the first variant. To obtain the core {liftCargo, clearCargo} we had to restrict the analysis to safety parts of the specification as in Algorithm 1, l. 6.

Technically the support of LTL specification patterns and past-time LTL works by encoding deterministic Büchi automata or observer automata as additional safety constraints θ^a, ρ^a with auxiliary variables $\mathcal{A}$ encoding the statespace of the automata. The acceptance of Büchi automata is encoded as justice assumptions in J^e or justice guarantees in J^s. Finally, a system specification for synthesis with patterns and past-time LTL is updated to $\langle \theta^s \wedge \theta^a, \rho^s \wedge \rho^a, J^s \rangle$ over the new system variables $\mathcal{Y} \cup \mathcal{A}$.

As an example consider the following assumption using the response pattern to express that the forklift can find a station by going forward:

```
ASM res:  G(!atStation -> F(atStation | mot!=FWD))
```

This response formula is not in the GR(1) fragment. It can however be used in GR(1) synthesis by adding a new Boolean variable aux to $\mathcal{A}$, adding aux=true to θ^a, adding next(aux) <-> ((atStation | mot!=FWD) | aux & atStation) to ρ^a, and adding aux=true to J^e. Similar translations exist for past-time LTL [6]. The translations have in common that the auxiliary specification parts θ^a and ρ^a and the new variables $\mathcal{A}$ are added on the system side of the specification for the GR(1) algorithm.

The definition of well-separation in Def. 2 uses the system specification $\langle \texttt{true}, \texttt{true}, \{\texttt{false}\} \rangle$ and thus does not support patterns and past-time LTL. When naively applying Def. 2 to specifications that use patterns or past-time LTL in assumptions an analysis automatically yields (potentially false) negatives. For the example of assumption res the system $\langle \texttt{true}, \texttt{true}, \{\texttt{false}\} \rangle$ fully controls the new variable aux and can thus always prevent the environment justice $\texttt{aux=true} \in J^e$.

We fix this inconsistency between the support for patterns and past-time LTL for GR(1) synthesis and the definition of well-separation by replacing the system specification $\langle \texttt{true}, \texttt{true}, \{\texttt{false}\} \rangle$ in Def. 2 with $\langle \theta^a_{\texttt{ASM}}, \rho^a_{\texttt{ASM}}, \{\texttt{false}\} \rangle$ where $\theta^a_{\texttt{ASM}}$ and $\rho^a_{\texttt{ASM}}$ are the auxiliary initial and safety constraints from the translation of patterns and past-time LTL that appear as part of assumptions in ASM. The reachable states to consider for well-separation are accordingly $\texttt{reachStates}(\theta^e \wedge \theta^a_{\texttt{ASM}}, \rho^e \wedge \rho^a_{\texttt{ASM}})$. We update Algorithm 1 with the restriction of reachable states in line 5 and the system specification $\langle \theta^a_{\texttt{ASM}}, \rho^a_{\texttt{ASM}}, \{\texttt{false}\} \rangle$ in line 6 and line 13.

In the updated definition of well-separation the assumption res alone will not be identified as a reason for non-well-separation. The environment can always force aux=true by either keeping atStation=true or setting atStation=true after atStation=false.

6.2 Preventing Forced Assumption Violations via Safety Guarantees

A non-well-separated environment is undesirable because it might allow synthesis of controllers that force assumption violations instead of satisfying their guarantees. We have presented ways to diagnose and debug non-well-separated environments. In this subsection we consider a different approach: considering the guarantees of a specification to prevent the system from forcing assumption violations.

From a methodological point of view fixing a non-well-separated environment specification should be preferred to enable reuse and support evolution of the environment specification. From a pragmatic point of view it might be enough that a system guarantees to not force the environment to

violate assumptions. Recall that the satisfaction of safety guarantees until assumptions are violated is part of strict realizability semantics φ^{sr} but not implication semantics $\varphi^{\rightarrow}$.

As an example, consider a non-well-separated environment specification consisting of the assumptions `dropCargo` and `clearCargo` from Listing 1. A strategy to force assumption violation is dropping cargo to require `next(cargo)` and driving backwards to require `next(!cargo)`. The following new guarantee `dropStop` expresses that the system has to stop when dropping cargo:

```
GAR dropStop:  G(lift=DROP -> mot=STOP)
```

A system satisfying `dropStop` cannot force a violation of an environment specification consisting the two assumptions `dropCargo` and `clearCargo`.

More formally, we define a weaker version of well-separation over the complete specification, i.e., all assumptions and guarantees.

Definition 4 (Well-Separation wrt. Guarantees). *A GR(1) specification with environment $\langle \theta^e, \rho^e, J^e \rangle$ and system $\langle \theta^s, \rho^s, J^s \rangle$, is well-separated iff*

$$sysWinSts(\langle \theta^e, \rho^e, J^e \rangle, \langle \theta^s, \rho^s, \{\texttt{false}\} \rangle) \cap reachStates(\theta^e \wedge \theta^s, \rho^e \wedge \rho^s) = \emptyset.$$

Note that well-separation implies well-separation with respect to guarantees, but not the other way around. Also note that a trivial case of well-separation wrt. guarantees is unrealizability of the safety part of the system, i.e., when the environment can force the system to a deadlock before the system can force an assumption violation.

6.3 Annotating Synthesized Controllers

Finally, we present a complementary approach for helping to understand non-well-separation by annotating a synthesized controller with traceability information (see [20]) of the assumptions it tries to violate. Note that we now talk about controllers synthesized for realizing a GR(1) specification (including system guarantees) and not the ones of Sect. 4.2 synthesized to show non-well-separation.

Given a controller realizing a specification we annotate every state in the controller with a reason why it was added to the controller. The annotations link states to guarantees the controller tries to satisfy or assumptions it tries to prevent from being satisfied. An excerpt of a controller for the forklift is shown in Fig. 4. The system specification for this example has two justice guarantees to always eventually pick up cargo, formally, `GF (lift=LIFT)`, and to always eventually deliver cargo, formally, `GF (lift=DROP)`. The upper left state in Fig. 4 satisfies the justice guarantee `GF (lift=LIFT)` and its successor to the right works towards satisfying the next justice guarantee `GF (lift=DROP)`. Note that the state on the bottom right is annotated as trying to prevent the justice assumption `GF (atStation)`. This annotation helps to identify undesired behavior caused by non-well-separation and links it to a responsible assumption.

We have implemented this approach in our synthesis environment as an extension of [20] for assumption traceability.

7. EVALUATION

We have implemented Algorithm 1, the non-well-separated core computation of Sect. 5, and the three extensions discussed in Sect. 6 in our synthesis framework using JTLV [23].

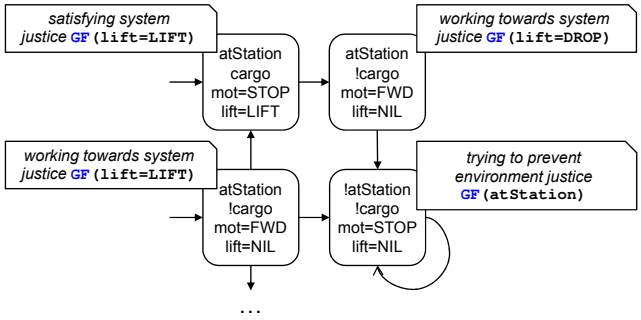

Figure 4: An excerpt of a synthesized controller annotated with reasons why states were added during controller construction

To evaluate the mechanism we suggest for debugging non-well-separation we consider the following research questions:

R1 Does non-well-separation appear in specifications?
R2 Can diagnosis be run efficiently during development?
R3 Does core computation effectively reduce the number of assumptions to inspect?

7.1 Material and Execution

Only few GR(1) specifications are available and these were usually created by authors of synthesis algorithms or extensions thereof. The most popular GR(1) specifications are AMBA and GenBuf published and used in many works [6, 14, 1]. We consider these specifications not well suited for our evaluation due to their origin and purpose.

To collect more realistic specifications we have conducted a project class on reactive synthesis for undergraduate students at Tel Aviv University. The task of the students was to write GR(1) specifications for robotic systems similar to the case study described in [17] with automation for synthesis and code generation to Lego NXT robots. Students worked in six teams of two or three students each for the full duration of a semester. They developed specifications using an extended version of the AspectLTL tools [18] and stored these in a version control system. The time between the first committed specification and the last committed specification was six month. The typical development cycle of the students was updating their specification, synthesizing a controller, and deploying generated code directly on their robot for validation. The different robots were a color sorter, an elevator, a humanoid, a self-balancing and remotely controlled robot, and two self-parking cars.

The students had tools for synthesis and code generation but not for detecting or debugging non-well-separation except for the possibility to synthesize controllers annotated with the controller's objective as described in Sect. 6.3.

We have collected a total of 86 specifications for six robots with 2 to 26 revisions per robot.[4] Seven committed specifications were syntactically invalid. The 79 valid specifications have on average 6 assumptions per specification from a single case of 0 up to a maximum of 10 assumptions. The state space (input, output, and auxiliary variables) ranges from 2^4 to 2^{47} (median 2^{27}, third quartile 2^{35}).

[4]The specifications are available from http://smlab.cs.tau.ac.il/syntech/separation/.

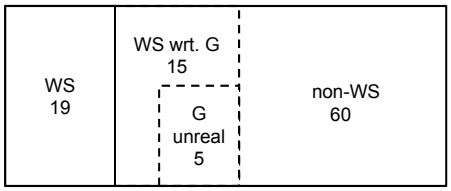

Figure 5: An overview of the evaluated specifications: 19 are well-separated, 60 are non-well-separated of which 15 are well-separated wrt. guarantees of which 5 have unrealizable safety guarantees

7.2 R1: Occurrence of Non-Well-Separation

We have analyzed all 79 syntactically valid specifications for well-separation with Algorithm 1 extended to support patterns and past-time LTL as described in Sect. 6.1.

As a result, only 19 out of 79 environment specifications were found to be well-separated. The algorithm detected non-well-separation for 60 environment specifications.

Algorithm 1 diagnosed the following cases of non-well-separation: 4 times case (**P-all**, **E-safe**), 11 times case {(**P-reach**, **E-safe**), (**P-all**, **E-just**)}, 12 times case {(**P-reach**, **E-safe**), (**P-reach**, **E-just**)}, 26 times case (**P-all**, **E-just**), and 7 times case (**P-reach**, **E-just**). These cases do not only appear in intermediate versions but also in the final submissions of the students.

The tool used by the students implements the more robust strict realizability semantics φ^{sr}. We have thus also checked whether including the guarantees in the well-separation check (described as an extension in Sect. 6.2) prevents the system from exploiting non-well-separation. Indeed, 15 out of 60 non-well-separated environment specifications are well-separated wrt. guarantees, i.e., the safety guarantees ensure that the environment cannot be forced to violate assumptions. A further check revealed that in 5 out of these 15 cases the well-separation wrt. guarantees is trivially satisfied because the safety guarantees are unrealizable. We have summarized the numbers in an overview in Fig. 5. The left-most box represents the desired 19 well-separated specifications while the right boxes represent 60 non-well-separated specifications. With strict realizability semantics 20 specifications in the dotted middle box prevent forced assumption violations (5 trivially by unrealizable safety guarantees).

To answer research question R1: **non-well-separation frequently occurs in environment specifications** (60 out of 79 specifications) and in many specifications violation of an assumption can be **forced from all initial positions** (**P-all**) (41 out of 79 specifications).

7.3 R2: Efficiency of Diagnosis

It is important that the diagnosis of non-well-separation is fast and can be executed frequently by the engineer. We have thus measured running times of Algorithm 1 for a diagnosis of cases of non-well-separation for the 79 specifications in our evaluation. Again the algorithm supports past-time LTL and patterns as described in Sect. 6.1.

We have run the experiments for measuring diagnosis times on an ordinary laptop computer with an i7 2.3 GHz CPU running Windows 10 64Bit, Java 1.7 32Bit, and CUDD 2.4.2 32Bit with automatic variable reordering. Neither our Java implementation nor CUDD made use of more than one core of the CPU. The times we report are wall-clock-times measured by the Java API. We repeated each experiment 12 times and report running times in milliseconds for all 12*79 runs on syntactically valid specifications.

Due to the small size of some specifications many runs of Algorithm 1 were reported to complete after 0ms. The minimum time and the first quartile of all runs are both 0ms. The median of measured running times is 15ms, the third quartile is 78ms and the maximum time to run Algorithm 1 on the specifications was 250ms. As a reference time for each specification we have also measured synthesis times, i.e., time of the realizability check and construction of a symbolic controller[5]. We report on the factors that non-well-separation diagnosis is faster than symbolic controller synthesis (we removed 452 out of 948 cases where the time of Algorithm 1 was measured as 0ms). The first quartile is at factor 5 (i.e., non-well-separation diagnosis is 5 times faster than checking realizability and constructing the controller), the median at factor 9, and the third quartile is at factor 23. Only for five very small specifications synthesis was faster than diagnosis and both times measured were below 16ms.

To answer research question R2: **diagnosing non-well-separation using Algorithm 1 is conveniently efficient**, indeed it is **in 75% of the cases more than 5 times faster than the synthesis step**, which is executed frequently during specification development.

7.4 R3: Reduction by Core Computation

We are interested whether the computation of a non-well-separated core effectively reduces the number of assumptions to consider for understanding a reason for non-well-separation of the environment specification.

We have run our core computation algorithm based on delta-debugging and Def. 3 as described in Sect. 5 on all 60 non-well-separated specifications. The results of the computation were minimal sets of assumptions consisting of a single assumption for 45 specifications and of two assumption for 15 specifications. The reduction factor first quartile is 3, the median is 5, and the third quartile is 6.25.

A non-well-separated core of a single assumption is very surprising. We expected a set of assumptions that together presents a non-well-separation problem. We have thus inspected some of the specifications with a singleton core and found out that indeed the assumptions in the core refer to system variables the environment cannot control. We have rerun the experiments with all system guarantees according to Sect. 6.2 to ensure that the analysis did not miss implicit patterns implemented by the students. Except for the 15 specifications where the guarantees prevent forced assumption violation the cores remained at sizes of one or two assumptions.

To answer research question R3: Core computation **effectively reduces the number of assumptions** to consider (by a factor larger than 3 in 75% and larger than 5 in 50% of the specifications).

[5]For most of the specifications concrete controller construction was not feasible and students executed symbolic controllers on their robots.

7.5 Discussion and Threats to Validity

It is important to note the limitations of our findings and threats to their validity. We have based our evaluation on specifications created by students with no prior experience in LTL and specification writing.

From diagnosing specifications we found out that 76% of the specifications are non-well-separated and most non-well-separated cores consisted of a single assumption. These rather simple cases of non-well-separation may be due to students' unawareness of the problem of non-well-separation, which was not part of the teaching during the project. Moreover, non-well-separation did not always lead to a visible problem because our GR(1) implementation uses the more robust strict realizability semantics φ^{sr} and favors progress over assumption violation during strategy extraction and because validation by execution on the robot typically did not exercise many scenarios. Nevertheless, many times students consulted with us about observed undesired behavior in controllers resulting from non-well-separation, which indicates that the problem was indeed there, implicitly.

The analysis of multiple revisions of the same specification (from the version control system used by the students) bares the risk of analyzing intermediate specifications that are more likely to contain problems. Our observations of development behaviors and the low number of committed revisions however indicate that most versions were considered stable by the students before committing them. Nevertheless, our debugging techniques are intended to support developers during all stages of development, including intermediate versions.

Finally, students might have implemented assumption patterns in the specification that were not identified by our analysis. The analysis is limited to patterns described in Sect. 6.1. To mitigate this threat we have overapproximated the pattern implementations by considering all guarantees as described in Sect. 7.2. We still found that 45 out of 60 non-well-separated specifications remain non-well-separated.

8. RELATED WORK

We give a brief overview of works on the relation between environment specifications and controllers, debugging mechanisms for unrealizable specifications, and vacuity.

Assumptions: Bloem et al. [4] provide an overview and discussion how assumptions are treated in reactive synthesis. They argue that most approaches, e.g., GR(1) synthesis [6, 21], insufficiently handle assumption violations. They suggest and later define [5] cooperative synthesis levels where the highest level ensures that assumptions and guarantees are always satisfied. Ehlers et al. [10] present cooperative GR(1) synthesis for one of the levels that ensures that from every state the environment can satisfy its guarantees. Cooperative synthesis may fail when regular GR(1) synthesis does not. Our work on well-separation can help in the context of cooperative synthesis because every controller for a well-separated specification is cooperative.

Well-separation for GR(1) environments has been introduced by Klein and Pnueli [13]. They defined this property to show when the GR(1) semantics of strict realizability and implication realizability agree. We have argued for the importance of well-separation in general and extended Klein and Pnueli's work with more fine-grained analysis of cases, strategies, a core, implementation and evaluation.

D'Ippolito et al. [8] present GR(1) synthesis for event-based controllers and define anomalous controllers that satisfy their specification by forcing assumption violations. The notion of assumption compatibility in the event-based case is a dual of well-separation. We believe that the techniques we introduced can be transferred to debug environments in the event-based case.

Debugging Unrealizability: One may view our work as using and extending ideas that were recently introduced to deal with unrealizability (counter strategies, cores), in order to create new means to deal with the challenge of well-separation. Counter-strategy synthesis for GR(1) and interactive execution were presented, e.g., by Cimatti et al. [7], Könighofer et al. [14], Maoz and Sa'ar for AspectLTL [20] and for scenarios [19], and Raman and Kress-Gazit [24] in a robotics domain. Non-well-separation is very different than unrealizability. Compared to our work on non-well-separation, counter-strategies for unrealizability focus on the system part and deal only with the case **P-all**.

Alur et al. [1] suggest a heuristics for fixing unrealizability by adding assumptions obtained from enumerated candidates checked against a counter strategy. The case of well-separation is different because it is not monotonic and deals with assumptions only. A dual approach might be possible for the extension with guarantees defined in Sect. 6.2.

Vacuity: Finally, the problem of vacuity as defined by Beer et al. [3] and studied in many works (e.g., [2, 12, 15]), in the context of model-checking, can be viewed as related to well-separation and synthesis. Fisman et al. [11] have extended vacuity to specifications where satisfaction is replaced by realizability and witnesses are subformulas. All these works are however very different than our work in their approach and motivation. First, a vacuous controller could be synthesized for a non-well-separated specification of case **P-all**[6] and second, non-well-separation of case **P-reach** cannot be detected by checking a controller for vacuous satisfaction.

9. CONCLUSION

In this work we investigated non-well-separation, where an environment can be forced to violate the assumptions, in the context of GR(1), an expressive fragment of LTL. We distinguished different cases of non-well-separation, and computed strategies showing how the environment can be forced to violate its assumptions. We further showed how to find a core, a minimal set of assumptions that lead to non-well-separation, and extended our work to support past-time LTL and patterns.

We implemented our work and evaluated it on 79 specifications. The evaluation shows that non-well-separation is a common problem in specifications and that our tools can be efficiently applied to identify it and its causes. Our work is the first to investigate and implement means to identify and address non-well-separation. It is also the first to evaluate non-well-separation on a corpus of specifications.

The 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. As part of this project we are building engineer-friendly tools for writing and understanding temporal specifications for reactive synthesis.

[6]Non-well-separation is necessary but not sufficient for a controller that is vacuous in the system guarantees.

10. ACKNOWLEDGMENTS

JOR acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research. Part of this work was done while SM was on sabbatical as visiting scientist at MIT CSAIL. 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] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*, pages 26–33. IEEE, 2013.

[2] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *CAV*, volume 2725 of *LNCS*, pages 368–380. Springer, 2003.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulaas. In *CAV*, volume 1254 of *LNCS*, pages 279–290. Springer, 1997.

[4] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer. How to handle assumptions in synthesis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, volume 157 of *EPTCS*, pages 34–50, 2014.

[5] R. Bloem, R. Ehlers, and R. Könighofer. Cooperative reactive synthesis. In *ATVA*, volume 9364 of *LNCS*, pages 394–410. Springer, 2015.

[6] 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.

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

[8] 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.

[9] 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.

[10] R. Ehlers, R. Könighofer, and R. Bloem. Synthesizing cooperative reactive mission plans. In *IROS*, pages 3478–3485. IEEE, 2015.

[11] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In *Haifa Verification Conference (HVC)*, volume 5394 of *LNCS*, pages 7–22. Springer, 2008.

[12] A. Gurfinkel and M. Chechik. How vacuous is vacuous? In *TACAS*, volume 2988 of *LNCS*, pages 451–466. Springer, 2004.

[13] U. Klein and A. Pnueli. Revisiting synthesis of GR(1) specifications. In *Haifa Verification Conference (HVC)*, volume 6504 of *LNCS*, pages 161–181. Springer, 2010.

[14] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

[15] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

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

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

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

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

[20] S. Maoz and Y. Sa'ar. Two-way traceability and conflict debugging for aspectltl programs. *T. Aspect-Oriented Software Development*, 10:39–72, 2013.

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

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

[23] 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.

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

[25] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, volume 1687 of *LNCS*, pages 253–267. Springer, 1999.