Kind Controllers and Fast Heuristics for Non-Well-Separated GR(1) Specifications

Ariel Gorenstein
Tel Aviv University
Israel

Shahar Maoz
Tel Aviv University
Israel

Jan Oliver Ringert
Bauhaus University Weimar
Germany

ABSTRACT

Non-well-separation (NWS) is a known quality issue in specifications for reactive synthesis. The problem of NWS occurs when the synthesized system can avoid satisfying its guarantees by preventing the environment from being able to satisfy its assumptions.

In this work we present two contributions to better deal with NWS. First, we show how to synthesize systems that avoid taking advantage of NWS, i.e., do not prevent the satisfaction of any environment assumption, even if possible. Second, we propose a set of heuristics for fast detection of NWS. Evaluation over benchmarks from the literature shows the effectiveness and significance of our work.

ACM Reference Format:
Ariel Gorenstein, Shahar Maoz, and Jan Oliver Ringert. 2024. Kind Controllers and Fast Heuristics for Non-Well-Separated GR(1) Specifications. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3608131

1 INTRODUCTION

Given a temporal specification, reactive synthesis is the problem of synthesizing a correct-by-construction reactive system, i.e., one whose all runs satisfy the specification. We focus on GR(1) [6], a fragment of LTL where specifications consist of initial, safety, and justice assumptions and guarantees. Initial assumptions and guarantees specify what is expected to hold in all initial states. Safety assumptions and guarantees specify what is expected to hold in all states and transitions. Justice assumptions and guarantees specify assertions that are expected to hold infinitely often in every infinite run.

The GR(1) fragment is relatively popular thanks to its relative lower complexity compared to LTL on the one hand and to its expressiveness on the other hand. For example, it allows to express almost all the well-known specification patterns of Dwyer et al. [8, 18]. GR(1) specifications have been used in several application domains, e.g., to specify and implement autonomous robots [15, 19], control protocols for smart camera networks [24], distributed control protocols for aircraft vehicle management systems [23], and device drivers [26]. It is supported by several tools [1, 10, 21, 27].

Non-well-separation (NWS) is a known quality issue in specifications for reactive synthesis. The problem of NWS occurs when the synthesized system is able to avoid satisfying its guarantees by preventing the environment from being able to satisfy its assumptions. Although this solution to the synthesis problem correctly satisfies the specification, it is undesired, as it misses the purpose of the system and the intention of the engineer.

NWS appears in many specifications in several benchmarks, in specifications written by students and by formal methods experts alike [20]. Moreover, it was recently reported as one of the major challenges faced by software engineers who start using a reactive synthesizer [16].

In this work we present two independent contributions to better deal with NWS. The first contribution relates to the usefulness of the controller one may synthesize from an NWS specification. The second contribution relates to the performance of NWS detection and problem localization.

First, when a specification is NWS, it may still allow for the synthesis of a *kind controller*, i.e., one that realizes it without forcing the environment into assumption violation. Indeed, in [17], Majumdar et al. have shown an algorithm for generation of a system controller that not only allows the environment to fulfill its justice assumptions, but also complies with its guarantees if the environment cooperates and satisfies its assumptions. However, their algorithm handles only one case of NWS, the case where the system prevents the environment from satisfying a justice assumption. We extend this algorithm with a novel reduction such that the extended algorithm handles not only justice assumption violations but also safety assumption violations, if any.

We show that our reduction has small performance overhead and that the extended algorithm is sound and complete; it results in a kind controller, one that does not prevent the environment from satisfying any of its assumptions, if and only if a kind controller for the specification exists.

Second, we present *novel heuristics for NWS detection.* Rather than detecting NWS by reduction to realizability checking, as was done in [20], we define a set of heuristics, each able to detect a single commonly-reoccurring problem that makes a specification NWS. As our experiments show, our heuristics are much faster than the technique in [20]. Moreover, in many cases, they provide faster problem localization information, as they directly point to the root cause of the problem and thus make the NWS core computation, with expensive realizability checks, unnecessary.

Importantly, the improved performance in detection does not come at the expense of soundness, but only at the expense of completeness. That is, our heuristics may miss some NWS specifications, but as we prove, when they detect a case of NWS, the specification is indeed NWS.

ICSE 2024, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3608131

```
1  env boolean atStation;
2  env boolean cargo;
3
4  sys {FWD, BWD, STOP} mot;
5  sys {LIFT, DROP} lift;
6
7  asm findStat: //always possible to find a station
8    alwEv (atStation);
9  asm samePos: //same station position when stopped
10   alw (mot=STOP -> next(atStation)=atStation);
11 asm liftCargo: //lifting clears sensor
12   alw (lift=LIFT -> next(!cargo));
13 asm dropCargo: //dropping senses cargo
14   alw (lift=DROP -> next(cargo));
15 asm clearCargo: //backing up clears cargo
16   alw (mot=BWD -> next(!cargo));
17 gar move: //always eventually advance
18   alwEv (mot=FWD);
```

Listing 1: Excerpt of specification for the forklift controller

We have implemented and evaluated our ideas in the Spectra synthesizer and IDE [21]. Our evaluation examines the proposed approaches on known benchmarks from the literature, the SYNTECH well-known specification benchmarks [21] and the parametric specification benchmarks AMBA[4] and GenBuf [5], which are both NWS. The evaluation shows that in many cases, the algorithm of [17] misses the safety assumptions and generates controllers that falsify them, while the reduction that we propose ensures that controllers that are unkind are never synthesized, with small increase in running time. It also shows that our heuristics can detect quite a lot of specifications that are NWS, and that they execute considerably faster than previous methods for NWS detection and localization.

2 EXAMPLE

We use a running example, adapted from a specification of a Lego forklift [19]. The example demonstrates NWS, shows how previous work has proposed to detect and address it, and how our new work improves upon these, to help engineers dealing with it.

The forklift has two sensors: one to determine whether it is at a station and another to detect cargo. It has two motors, to drive the forklift and to lift the fork. Values read by the sensors are provided as inputs, and outputs are commands that control the motors.

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 cannot 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 **alwEv** (`atStation`) named `findStat` in List. 1, ll. 7-8. The temporal operator **alw** intuitively stands for always, i.e., at every state, and **Ev** stands for eventually, i.e., within finitely many steps (we use the keywords `alw` and `alwEv` in Spectra syntax for LTL `G` and `GF`).

Additional assumptions in List. 1 describe environment reactions to actions of the forklift. The assumption `samePos` specifies that the value of the station sensor remains the same if the forklift stops. 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 environment behavior for handling cargo. When the forklift lifts cargo the cargo sensor is cleared (assumption `liftCargo`, l. 11). When it drops cargo the cargo is detected by the sensor (assumption `dropCargo`, l. 13). Finally, the forklift can clear the cargo by moving backward from it (assumption `clearCargo`, l. 15).

The engineers complete the specification consisting of assumptions and guarantees and successfully synthesize a controller that satisfies all guarantees if all assumptions hold. Yet, once the controller is deployed the team observes strange behavior. They see this happens when the forklift drives backwards while dropping cargo. They launch our detection tool, which informs them that the environment can be forced to violate some safety assumptions, namely `dropCargo` and `clearCargo`, which can cause a contradiction in the value of the variable `cargo`. Having discovered the cause of the problem, it is a much easier for the engineers to find a solution.

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., again, some assumption must be violated. This time, our tool informs the team that some safety assumptions can contradict a justice assumption: in this case `samePos` (which states that the station sensor reading does not change when motors are stopped) contradicts `findStat` (which states a station must be reached). When the forklift is not at a station it stops and thus forces the environment to violate its justice assumption `findStat`.

The above are examples of a NWS problem, as detected by our tool, which also points to the assumptions involved in the problem. **While detection and localization of NWS were presented before [20], our heuristics for it are much faster**.

As an alternative approach to fixing NWS, the team can use an algorithm for synthesizing a fair system strategy, one that does not exploit the fact that the environment assumptions can be forced to be violated. Indeed, when using the algorithm of [17], the generated controller ensures not to take advantage of the fact that the assumptions `findStat` and `samePos` can be forced to contradict each other. However, this algorithm still may synthesize a controller that exploits the combination of the assumptions `dropCargo` and `clearCargo`, both of which are safety assumptions. **Our extended algorithm, on the other hand, guarantees that *all* NWS flaws are not exploited, if possible**.

3 PRELIMINARIES

3.1 Linear Temporal Logic and GR(1) Games

We use standard definitions of linear temporal logic (LTL), e.g., as found in [6]. LTL formulas can be used as specifications of reactive systems, which are represented as a 2-player game between an environment player and a system player. Atomic propositions in such games are interpreted as environment (input) variables $\mathcal{X}$ and system (output) variables $\mathcal{Y}$. An assignment to all variables is called a state, which can be represented as a tuple $\langle X, Y \rangle$, where $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$ are the sets of environment and system variables resp. that hold at the state. A transition between states occurs when the

environment player assigns values to the input variables, which is then answered by the system player assigning values to the output variables. A play is a sequence of states, which can be infinite or finite, if it reaches a deadlock for one of the players, i.e. a state from which the player has no legal assignment to its variables.

An environment initialization E_I is a subset of $\mathcal{X}$ that indicates which input variables hold initially. A system initialization function $S_I\colon 2^{\mathcal{X}} \to 2^{\mathcal{Y}}$ indicates which output variables hold initially, given the initial input variables. The initial state in the game is $\langle E_I, S_I(E_I)\rangle$. An environment transition function $E_T\colon (2^{\mathcal{X}\cup\mathcal{Y}})^+ \to 2^{\mathcal{X}}$ prescribes the next input variable assignments given the previous states in the game. A system transition function $S_T\colon (2^{\mathcal{X}\cup\mathcal{Y}})^+ 2^{\mathcal{X}} \to 2^{\mathcal{Y}}$ prescribes output variable assignments, given the previous states and the current input variable assignment. An environment strategy E is the tuple $\langle E_I, E_T\rangle$, and a system strategy S is the tuple $\langle S_I, S_T\rangle$. An environment strategy E and a system strategy S together induce a single play which follows both strategies. Such strategies can be represented as an automaton called a controller.

Given a system strategy S and an LTL specification ψ, the winning states of S in ψ are the states from which the play induced by S and E satisfies ψ, for any environment strategy E. A specification ψ is called realizable if a system strategy $\langle S_I, S_T\rangle$ exists such that for any environment strategy $\langle E_I, E_T\rangle$, the initial state $\langle E_I, S_I(E_I)\rangle$ is a winning state. Such a strategy is called a winning strategy. The goal of LTL synthesis is, given an LTL specification, find a winning strategy that realizes it, if such a strategy 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 often. 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
- $\theta^e = \bigwedge_{n=1\ldots i_e} \theta^e_n$ a conjunction of i_e boolean assertions over $\mathcal{X}$, the initial input variable assignments.
- $\theta^s = \bigwedge_{n=1\ldots i_s} \theta^s_n$ a conjunction of i_s boolean assertions over $\mathcal{X}\cup\mathcal{Y}$, the initial input and output variables.
- $\rho^e = \bigwedge_{n=1\ldots t_e} \rho^e_n$ a conjunction of t_e assertions over $(\mathcal{X}\cup\mathcal{Y}, \mathcal{X})$, characterizing the constraints over the input variables in the next state given the input and output variables in the current state.
- $\rho^s = \bigwedge_{n=1\ldots t_s} \rho^s_n$ a conjunction of t_s assertions over $(\mathcal{X}\cup\mathcal{Y}, \mathcal{X}\cup\mathcal{Y})$, describing constraints over the output variables in the next state given the input and output variables in the current state and the input variables in the next state.
- $J^e = \{J^e_n\}_{n=1}^{j_e}$ a set of j_e justice requirements of the environment.
- $J^s = \{J^s_n\}_{n=1}^{j_s}$ a set of j_s justice requirements of the system.

Specifications for GR(1) synthesis have to be expressible in the above structure and thus do not cover the complete LTL. The environment and system requirements are the tuples $\langle \theta^e, \rho^e, J^e\rangle$ and $\langle \theta^s, \rho^s, J^s\rangle$, resp. GR(1) synthesis has two different notions of realizability [6]. The first and more intuitive notion is called implication realizability, because all environment assumptions imply all system guarantees. It is expressed in the following LTL formula:

$$\psi^{\rightarrow} = (\theta^e \wedge \mathbf{G}\rho^e \wedge \bigwedge_{n\in 1..j_e} \mathbf{GF} J^e_n) \rightarrow (\theta^s \wedge \mathbf{G}\rho^s \wedge \bigwedge_{n\in 1..j_s} \mathbf{GF} J^s_n)$$

The second kind of realizability, which is the one mostly used in the literature, is called strict realizability, and is given in Def. 1.

Definition 1 (Strict Realizability). *A GR1 specification with environment and system requirements $\langle \theta^e, \rho^e, J^e\rangle$ and $\langle \theta^s, \rho^s, J^e\rangle$ resp. is strictly realizable if the LTL formula ψ^{sr} is realizable, where*

$$\psi^{sr} = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathbf{G}((\mathbf{H}\rho^e) \rightarrow \rho^s)) \wedge (\theta^e \wedge \mathbf{G}\rho^e \rightarrow (\bigwedge_{n\in 1..j_e} \mathbf{GF} J^e_n \rightarrow \bigwedge_{n\in 1..j_s} \mathbf{GF} J^s_n))$$

Realizability of ψ^{sr} implies realizability of $\psi^{\rightarrow}$. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis for ψ^{sr} have been presented in [6, 25]. The synthesis algorithm utilizes μ calculus theory [14], by evaluating a formula with three nested fixed-point computations.

3.2 Well-Separation

Klein and Pnueli [13] defined well-separation (WS) as a sufficient property of environment specifications such that realizability of ψ^{sr} is equivalent to realizability of $\psi^{\rightarrow}$. A WS environment can satisfy all assumptions from every reachable state. In specifications where the environment is non-well-separated (NWS), the controller can satisfy ψ^{sr} by actively preventing the environment from satisfying its assumptions. Def. 2 repeats Klein and Pnueli's definition of WS.

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

Note that WS is defined as a property of the environment part of the specification, i.e., the assumptions, without the guarantees. Intuitively, the system specification $\langle$`true`, `true`, {`false`}$\rangle$ means that initially and for every step the system choices are unconstrained (θ^s = `true` = ρ^s) but its justice requirements are unsatisfiable J^s = {`false`}. System strategies for ψ^{sr} thus cannot win, except by ensuring that the left-hand-size of one of the three implications of ψ^{sr} does not hold. So, existence of a system winning strategy means the system can force the environment to violate its assumptions.

Maoz and Ringert [20] identify cases of NWS. They distinguish whether environment assumptions can be violated in runs starting from all initial states (P-all), or only from some initial environment choices (P-reach), and they distinguish the type of assumption that can be violated (E-ini, E-safe, or E-just).

Majumdar et al. [17] propose an alternative to the standard GR(1) synthesis algorithm, which synthesizes a controller that satisfies the system guarantees while allowing the environment to satisfy its justice assumptions, if such a controller exists. That is, in addition to the standard requirement of strict realizability, every finite play compliant with the controller's strategy must be able to be extended by the environment to an infinite play that satisfies the environment's justice assumptions. This ensures that no finite play leads to a state from which these justices can no longer be satisfied. It is achieved by replacing the original GR(1) three nested fixed-point (3FP) computation by four nested fixed-points. The BDDs

computed by this algorithm can be used to construct a controller as usual.

In spite of the excess computation time needed because of the additional fixed-point, the benefit of the algorithm is that even in a specification with a NWS environment, it can be used to synthesize a controller that does not exploit this NWS, and allows the environment to satisfy its justices, if such a controller exists. Importantly however, if a *safety* assumption of the environment can be forced to be violated, then this algorithm may still lead the environment into a deadlock. We overcome this limitation in Sect. 4.

4 SYNTHESIZING KIND CONTROLLERS

Our first contribution is extending the algorithm of Majumdar et al. [17] in a way that ensures it synthesizes controllers that allow the environment to satisfy *all* of its assumptions, not only its justice assumptions, if such a controller exists. That is, even though the controllers generated by the 4 fixed-point algorithm of [17] are guaranteed to allow the environment to fulfill its *justice* assumptions, they still may actively prevent it from satisfying its *safety* assumptions by taking advantage of NWS. Therefore, their algorithm is useful in specifications where the NWS is of type E-just, but is not useful, and in a way, misleading, for those of type E-safe.

In order to deal with this problem, we define a stronger notion of strict realizability, called *kind realizability*, which requires not only the satisfaction of ψ^{sr}, but also that any finite prefix of a play between the system and the environment can be extended into an infinite play that satisfies the environment assumptions. We extend the 4FP algorithm such that it checks for kind realizability of specifications, rather than strict realizability. That is, the controllers generated by the extended algorithm do not prevent the environment from satisfying any of its assumptions, regardless of their type. If there does not exist a controller that ensures this while also satisfying the guarantees of the system, then our algorithm reports that the specification is not kindly realizable.

Example 1. *Consider the example from List. 1. When an engineer uses the standard 3FP GR(1) algorithm to check the realizability and synthesize a controller for this specification, the transition function of this controller always sets the system variable* `mot` *to* `STOP`*. If the robot was initially not at a station, then due to the assumption* `samePos` *the robot will never be at a station. This violates the justice assumption* `findStat`*, which requires that a station is visited infinitely often. Therefore, even though the runs of this controller do not satisfy the justice assumption, it technically correctly realizes the specification.*

Frustrated that the controller actively prevents the justice assumption of the environment to be satisfied, the engineer may try to use the 4FP algorithm of [17], which guarantees that the controller allows justice assumptions to be satisfied. Unfortunately, when this algorithm constructs a controller for this specification, the transition function of this new controller always sets the variable `lift` *to* `DROP` *and the variable* `mot` *to* `BWD`*. This forces the environment into violating either the safety assumption* `dropCargo` *or the safety assumption* `clearCargo`*.*

Instead, the engineer can use the extended 4FP algorithm we suggest in order to generate a new controller. The transition function of this controller sets the variables `lift` *and* `mot` *to* `DROP` *and* `FWD` *respectively. It thus allows both the environment and the system to satisfy all their assumptions and guarantees, respectively.*

Definition 3 (kind strategy). *A winning system strategy S is a kind strategy if for any environment strategy E, the play compliant with S and E satisfies ψ^{sr}. In addition, at any position i in the play, E can be replaced by an alternative environment strategy E'_i, such that the play starting at state σ_i and following S and E'_i satisfies $\mathbf{G}\rho^e \wedge \bigwedge_{n=1\ldots j_e} \mathbf{GF} J^e_n$.*

Definition 4 (kind realizability). *A specification is kindly realizable if there exists a kind strategy that realizes it.*

A controller that follows a kind strategy is called a kind controller. Although they do not use the term kind realizability, Majumdar et al. define an equivalent synthesis problem and prove that their algorithm correctly solves it. However, they implicitly assume that deadlocks are not possible in the GR(1) game (i.e., that every reachable state has at least one successor state). Hence, their proof does not apply to specifications where this assumption does not hold. In contrast, we prove that our extended algorithm correctly synthesizes kind controllers even when deadlocks are reachable.

4.1 Algorithm

Roughly, our algorithm begins by applying a reduction to the specification, transforming it into a modified specification. We then give this modified specification to the 4FP algorithm of [17] as input. The reduction makes sure that if a *safety* assumption can be violated in the original specification, then a *justice* assumption can be violated in the modified specification. Since the 4FP algorithm generates controllers that do not violate justice assumptions, these controllers will not violate safety assumptions in the original specification.

More technically, the reduction adds a new environment variable called `envHelp` and three assumptions that reference it:

- **`ini`** `(!envHelp)`
- **`alw`** `(envHelp -> next(envHelp))`
- **`alwEv`** `(!envHelp)`

That is, `envHelp` is required to initially be false, to remain true forever once it is set to true, and to infinitely often be set to false. In addition to these three new assumptions, the reduction converts each safety assumption ρ^e_n in the original specification to an assumption $\rho^e_n \vee \mathbf{next}(envHelp)$ in the new specification. Finally, it converts each safety guarantee ρ^s_n in the original specification to a guarantee $\rho^s_n \vee envHelp$ in the new specification.

Example 2. *List. 2 demonstrates the output of the reduction on the example specification from List. 1. Note that in addition to the three new assumptions at the top, each of the original safety assumptions was disjuncted with* **`next`**`(envHelp)`*. If there were safety guarantees, each of them would have been disjuncted with* `envHelp`*.*

Intuitively, at any time during the interaction between the system and the environment, the environment may choose to ignore its safety assumptions by setting `envHelp` to `true`. Once it does so, the new assumption **`alw`** `(envHelp` $\implies$ **`next`**`(envHelp))` enforces that this variable stays `true` forever, and both the system and the environment may ignore their original safeties from that point on. However, if this happens, the new justice assumption ensures

```
env boolean envHelp;
env boolean atStation;
env boolean cargo;

sys {FWD, BWD, STOP} mot;
sys {LIFT, DROP} lift;

asm ini (!envHelp);
asm alw (envHelp -> next(envHelp));
asm alwEv (!envHelp);

asm findStat: //always possible to find a station
  alwEv (atStation);
asm samePos: //same station position when stopped
  alw ((mot=STOP -> next(atStation)=atStation) or next(
      envHelp));
asm liftCargo: //lifting clears sensor
  alw ((lift=LIFT -> next(!cargo)) or next(envHelp));
asm dropCargo: //dropping senses cargo
  alw ((lift=DROP -> next(cargo)) or next(envHelp));
asm clearCargo: //backing up clears cargo
  alw ((mot=BWD -> next(!cargo)) or next(envHelp));
gar move: //always eventually advance
  alwEv (mot=FWD);
```

Listing 2: The specification from List. 1 after the reduction

that the environment loses the game. Therefore, if in the original specification the system can lead the environment into a deadlock, then in the modified specification the environment will be forced to avoid the deadlock by setting `envHelp` to `true`, and will therefore eventually violate the new justice assumption.

We denote by $\theta^{e'}$, $\rho^{e'}$, and $J^{e'}$ the initial, safety, and justice environment assumptions in the modified specification, and we denote by $\theta^{s'}$, $\rho^{s'}$, and $J^{s'}$ the initial, safety, and justice system guarantees in the modified specification (where $\theta^{s'} = \theta^{s}$, and $J^{s'} = J^{s}$, because the initial and justice guarantees of the system are unmodified by the reduction). We denote the new environment justice assumption by $J^{e'}_{j_e+1}$. That is, $J^{e'} = J^{e} \cup \{J^{e'}_{j_e+1}\}$. Finally, we denote by $\psi^{sr\prime}$ the strict realizability formula for the system specification $\langle \theta^{s'}, \rho^{s'}, J^{s'} \rangle$ and the environment specification $\langle \theta^{e'}, \rho^{e'}, J^{e'} \rangle$.

4.2 Correctness

We now state and prove the correctness of the reduction. In Theorems 1 and 2 we state that the reduction preserves both the realizability and the well-separation of the original specification, thereby preserving its semantics. Formally:

Theorem 1. *Let M be a specification, and let M' be the specification generated from M by the reduction. Then M is realizable if and only if M' is realizable.*

Theorem 2. *Let M be a specification and let M' be the specification generated from M by the reduction. Then M is well-separated if and only if M' is well-separated.*

Finally, we state that the 4FP algorithm combined with the reduction correctly answers the kind realizability synthesis problem.

Theorem 3. *A controller S synthesized by the algorithm from M' is a kind controller for M.*

Proofs of these theorems are in the supporting materials [12].

To conclude our first contribution, our algorithm overcomes NWS and synthesizes only a kind controller, if one exists. This is in contrast to the original 3FP GR(1) algorithm which may force the violation of any environment assumptions, or the 4FP algorithm of [17] which may violate environment safety assumptions.

5 FAST HEURISTICS

Our second contribution is a set of heuristics for fast detection, localization, and strategy construction for NWS caused by commonly occurring mistakes. The specification is tested against three algorithmic checks one by one; when a check is positive, the specification is detected as NWS. In addition, the heuristics point to the root cause of the NWS and allow the generation of a symbolic controller that demonstrates how the system can exploit the NWS.

It is important to note that our heuristics and their implementation in our tool are semantic, not syntactic. For example, to check whether an assumption references an environment variable we do not only look at the concrete syntax of the assumption as written by the engineer, which may indirectly reference such variables through Spectra predicates and defines. Instead, we look at the support of the BDD that represents this assumption in Spectra. This semantic analysis reduces the NWS cases we may miss.

Since our heuristics operate by semantic analyses of specifications with symbolic operations that are linear in the number of assumptions, rather than by an iterative, fixed-point computation, they provide a very quick method for NWS detection. The analysis is sound, in the sense that if a specification is detected as NWS, it is guaranteed to indeed be NWS. However, the analysis is incomplete; some NWS specifications may not be detected as such. Nevertheless, as each of the heuristics was inspired by a commonly reoccurring mistake that we have observed in many specifications, oftentimes, as we show in the evaluation section, the analysis successfully detects the NWS.

The heuristics operate by sequentially executing the checks detailed in Subsections 5.1-3. The following subsections detail the motivation and operation of each of the three heuristics. Localization is discussed in Sect. 6.3, and controller construction and proof of soundness are given in the supporting materials [12]. We prove the soundness of each heuristic using the following theorem.

Theorem 4. *If the heuristic reports NWS, then given the system specification $\langle$`True`, `True`, {`False`}$\rangle$, there exists a reachable state which is winning for the system.*

5.1 T1: Assumption with no Environment Variables

The first check looks for assumptions that reference only system variables. Such an assumption makes the specification NWS because the environment has no control over its satisfaction.

Example 3. *If List. 1 from Sect. 2 contained the justice assumption* `alwEv(mot=STOP)`*, this assumption would be a cause for NWS, because the variable* `mot` *belongs to the system and the environment has no means for ensuring that it is set to* `STOP` *infinitely often.*

Detection Let the assumptions in the specification be $s_1, \ldots, s_m$. The algorithm iterates the assumptions s_i one by one. For each, if the support of the BDD representing s_i contains only system variables, the algorithm reports an occurrence of NWS.

Algorithm 1 Heuristic T2

```
1:  C ← True
2:  for ρ_n^e ∈ ρ_1^e, ..., ρ_{l_e}^e do
3:      if StateInv(ρ_n^e) then
4:          C ← C ∧ ρ_n^e
5:      end if
6:  end for
7:  for ρ_n^e ∈ ρ_1^e, ..., ρ_{l_e}^e do
8:      if VAR(ρ_n^e) ∩ X ≠ ∅ and VAR(ρ_n^e) ∩ Y ≠ ∅ and VAR(ρ_n^e) ∩
        X' = ∅ then
9:          env_part ← [¬ρ_n^e]_{∃Y}
10:         if C ∧ env_part ≠ False then
11:             return True
12:         end if
13:     end if
14: end for
15: return False
```

5.2 T2: Safety Assumption with Only Current State Environment Variables

This heuristics targets safety assumptions where environment variables are referenced in their current state rather than next state.

Example 4. *Consider the assumption* `dropCargo` *in List. 1. A common mistake developers make is writing* **`alw`** `(lift=DROP ->` `cargo)` *instead of* **`alw`** `(lift=DROP ->` **`next`**`(cargo))`. *As the environment makes its move before the system, if it sets the variable* `cargo` *to* **`false`**, *the system can falsify this assumption by setting* `lift` *to* `DROP`. *This makes the specification NWS.*

Detection The execution of this heuristic is shown in Alg. 1. Lines 1-6 computes the conjunction of all safety assumptions that are state invariants and store it in the variable C. State invariants are assumptions that reference only environment variables and only in the current state (see Sect. 4.3 in [21]). Next, the algorithm iterates the safety assumptions. For each assumption, it first checks in line 8 whether it contains at least one environment variable in the current state, at least one system variable in the current state, and no environment variables in the next state. For this, it uses the function VAR, which returns the support of the BDD, i.e., the set of variables it references. If this check passes, the assumption is suspected as a source of NWS, and the heuristic continues to analyze it further in lines 9-12. Line 9 computes *env_part*, the constraint over the environment variables for which there exists an assignment to system variables that negates the assumption. This is done using existential quantification over the system variables. If the constraint expressed in *env_part* is satisfiable, then NWS should be reported, because once the environment satisfies this constraint, the system would be able to set a variable assignment that violates the assumption. Line 10 checks whether this constraint is indeed satisfiable.

5.3 T3: Contradicting Assumptions

This heuristics detects specifications where the system player can ensure that a set of assumptions cannot be mutually satisfied.

Example 5. *In List. 1, the system player can set the variable* `lift` *to* `DROP` *and the variable* `mot` *to* `BWD` *in order to ensure that either the assumption* `dropCargo` *or the assumption* `clearCargo` *would not be satisfied, because these assumptions require the environment to assign contradicting values to the variable* `cargo`. *As another example, if the specification contained the justice assumption* **`alwEv`** `cargo`, *it would conflict with the assumption* `clearCargo`. *This is because the system player could persistently set* `mot` *to* `BWD`, *making it impossible for the environment to satisfy both assumptions.*

The example describes two cases where the environment has a subset of assumptions that cannot be mutually satisfied. The first case is a set of safety assumptions contradicting each other, which occurs when these assumptions reference the same environment variables in the next state, and there exists an assignment to system and environment variables in the current state for which the conjunction of requirements on the variables in the next state is false. These are the assumptions `dropCargo` and `clearCargo` in the above example. The second case occurs when some safety assumptions can ensure that a condition specified by a justice assumption is never met, thereby making it infeasible for the justice assumption to hold infinitely often, as demonstrated by the second example. The goal of this heuristic is to detect such contradicting specifications.

Detection The details of the check are presented in Alg. 2. In addition to the function VAR, presented with the previous heuristic, which returns the set of variables referenced by a BDD, this algorithm utilizes a few other functions. First, $prime\colon \mathcal{B}(\mathcal{X} \cup \mathcal{Y}) \rightarrow \mathcal{B}(\mathcal{X}' \cup \mathcal{Y}')$ receives a boolean assertion over $\mathcal{X} \cup \mathcal{Y}$ and returns a boolean assertion over $\mathcal{X}' \cup \mathcal{Y}'$, by changing all variable references to references of the same variables in the next state. Similarly, $Unprime\colon \mathcal{B}(\mathcal{X}' \cup \mathcal{Y}') \rightarrow \mathcal{B}(\mathcal{X} \cup \mathcal{Y})$ receives an assertion over the variables in the next state, and returns the same assertion with all variables references being in the current state.

We now discuss the basic version of the algorithm, the one presented in black. The parts in blue are an extension to the algorithm, which we will discuss in Sect. 6.2.

The algorithm begins by initializing three BDD variables to True. The purpose of the variables, *next* and *next_persist*, is to describe constraints that the system can enforce over environment variables in the next state. *next* holds the constraints that the system can enforce once, and is used to detect the case of mutually contradicting safety assumptions. *next_persist* holds the constraints that can be enforced persistently, and is used to detect the case of safety assumptions that contradict a justice assumption. Finally, the variable *present* should express the constraints over the current state that need to hold in order for the system to be able to enforce the constraints *next* and *next_persist*.

Lines 5-30 detect the first described case, where safety assumptions cannot be mutually satisfied. In line 5 the algorithm iterates the safety assumptions. For each assumption, it calculates in line 6 a requirement over variables in the present state, for which there exists an assignment to variables in the next state that would negate the assumption. This is done by universal quantification over X' to obtain the requirements over X and Y that must hold for all values of X', and negating the result. This requirement is assigned to c. Line 7 verifies that the conjunction of c with *present* is not false, and if so, this conjunction is assigned to *present* in line 20. Lines 8 and

Algorithm 2 Heuristic T3

```
1: next ← True
2: next_persist ← True
3: present ← True
4: next_range ← {⟨True, {True}⟩}
5: for ρ_n^e ∈ ρ_1^e, ..., ρ_{t_e}^e do
6:   c ← ¬ [ρ_n^e|_{∀X'}] ∧ Unprime(next_persist)
7:   if present ∧ c ≠ False then
8:     f ← [ρ_n^e ∧ c]_{∃Y}
9:     if VAR(f) ∩ X ≠ ∅ and VAR(f) ∩ X' ≠ ∅ and
       length(VAR(f)) = 2 and Prime(f|_X) ⟺ f|_{X'} then
10:      v ← (VAR(f) ∩ X')[0]
11:      S ← ∅
12:      for val ∈ dom(v) do
13:        if (θ^e ∧ [v = val]) ≠ False then
14:          S ← S ∪ {[v = val]}
15:        end if
16:      end for
17:      next_range ← next_range ∪ {⟨c, S⟩}
18:    else
19:      f ← f|_{∃X}
20:      present ← present ∧ c
21:      next ← next ∧ f
22:      if next = False then
23:        return True
24:      end if
25:      if VAR(c) ∩ Y ≠ ∅ and Unprime(f) ∧ c ≠ False then
26:        next_persist ← next_persist ∧ f
27:      end if
28:    end if
29:  end if
30: end for
31: for ⟨c, S⟩ ∈ next_range do
32:   if present ∧ c ≠ False then
33:     for f ∈ S do
34:       if f ∧ next = False then
35:         return True
36:       end if
37:     end for
38:   end if
39: end for
40: for J_n^e ∈ J_1^e ... J_{j_e}^e do
41:   if Prime([J_n^e]_{∀Y}) ∧ next_persist = False then
42:     return True
43:   end if
44:   for ⟨c, S⟩ ∈ next_range do
45:     if present ∧ c ≠ False then
46:       for f ∈ S do
47:         if f ∧ Prime([J_n^e]_{∀Y}) = False then
48:           return True
49:         end if
50:       end for
51:     end if
52:   end for
53: end for
54: return False
```

19 calculate the requirement over X' resulting from ρ_n^e and c and assign the result to the variable f. This requirement is calculated by computing the conjunction of ρ_n^e with c, and the quantifying X and Y out. Line 21 conjuncts *next* with f, and if this conjunction is false, the algorithm reports in line 23 that it detected NWS. This is because the assertion over X' resulting from the conjunction of the variable f in the preceding iterations is unsatisfiable. In lines 25-27, the BDD *next_persist* is conjuncted with f, given that c contains at least one system variable and that the assertion f does not prevent the assertion c from holding next time. The purpose of this check is to ensure that *next_persist* only holds constraints that can be enforced by the system in every single step of the game.

Lines 40-53 detect the second case, where the system can violate a justice assumption by taking advantage of safety assumptions. The algorithm relies on the BDD *next_persist* already being initialized in the previous part. Line 40 iterates the justice assumptions of the environment. Given a justice assumption, line 41 checks whether there exists an assignment to system variables for which this justice assumption contradicts *next_persist*, and if so, NWS is reported in line 42. Since justice assumptions only use variables in the present state, while *next_persist* expresses a condition over the next state, the function *Prime* is used to make the two conditions comparable.

Finally, in the case that line 54 is reached, the heuristic reports that it did not detect NWS.

6 EXTENSIONS

In this section we present several independent extensions to the heuristics. In Subsection 6.1 we show how we adapt the heuristics to a different definition of WS that takes the system guarantees into account. Next, in Subsection 6.2 we extend T3 to detect more NWS instances, using the blue parts of Alg. 2. Subsection 6.3 shows how the heuristics can be used to construct a NWS core, and Subsection 6.4 explains how the heuristics deal with specifications that contain auxiliary variables.

6.1 Taking Guarantees into Account

An alternative definition of NWS that takes into account the constraints over the system's initialization and transitions is defined in Sect. 6.2 of [20]. This definition states that a specification is NWS if there exist reachable winning states for the system specification $\langle \theta^s, \rho^s, \{\texttt{false}\}\rangle$, i.e., where the system has to adhere to initialization and transition constraints, but its justice requirements are unsatisfiable. For example, if the specification from List. 1 contained the safety guarantee `alw !(lift=DROP and mot=BWD)`, then according to this alternative definition, the combination of the safety assumptions `dropCargo` with `clearCargo` would not be a cause of NWS, as the system would not be able to create a contradiction on the constraints over the variable `cargo`. In many cases, this relaxed definition of NWS is considered more useful than the original one, since it is easier and good enough to follow in practice.

In order to adapt our heuristics to this definition of WS, whenever the heuristics for the general case would report NWS and construct a controller $\langle S_I, S_T\rangle$ (controller construction by the heuristic is explained in their proofs of soundness, found in the supporting materials [12]), we report NWS only if $S_I \wedge \theta^s \neq false$ and $S_T \wedge \rho^s \neq false$, and we instead generate the controller $\langle S_I \wedge \theta_s, S_T \wedge \rho^s\rangle$.

6.2 Extending T3 to Detect More Cases

The combination of assumptions `findStat` and `samePos` in the example in List. 1 is a cause of NWS which is not detected by T3 as presented in Sect. 5.3. The blue parts in Alg. 2 address this.

Intuitively, the assumption `samePos` is special in the sense that the system can exploit it to persistently set the variable `atStation` to any value in its domain (in contrast, for example, the assumption `dropCargo`, can persistently set the variable `cargo` only to the value true). To account for this, the heuristic uses the variable *next_range* to hold a mapping between constraints over Y and the resulting set of constraints over X'. In this example, it would map the constraint `mot=STOP` to the set {`atStation'=true, atStation'=false`}.

Lines 9-17 of Alg. 2 construct this mapping. After the system variables are quantified out of f (line 8), the heuristic checks whether the resulting BDD has only two variables remaining, one in X and one in X', and whether the constraints over X and X' are biconditional. If so, a new entry is added to *next_range*, where the constraint over Y is c and the set of constraints over X' consists of a BDD for every value val in the domain of v (where v is the single variable in X'), stating that v equals val (except those values that are not allowed by the initial constraints of the environment). Later, lines 31-39 check whether there exists an entry in the mapping, $\langle c, S\rangle$, such that the conjunction of c with *present* is not false and whether there exists a BDD f in S that contradicts *next* (recall that *next* is also an assertion over X'). If so, NWS is reported. Lines 44-52 perform the same check against each of the justice assumptions.

6.3 NWS Core

In order to debug NWS specifications, it is often useful to find an NWS core, i.e., a (locally) minimal subset of assumptions that makes the specification NWS. Our heuristics can be used for this purpose.

When either of the first two heuristics detects NWS, finding the core is easy, as it consists of the single assumption that causes the NWS that was detected. When T3 detects NWS, we utilize Delta Debugging (DDMin) [28] to compute a core. Roughly, given a set that satisfies a monotonic criterion, DDMin can detect a locally minimal subset that satisfies it. If the heuristic detects that some safety assumptions cannot be mutually satisfied (i.e., it returns at line 23 or 35), we use the safety assumptions as the set, and the first part of the algorithm (lines 1-39) as the monotonic criterion (the criterion is monotonic as it checks for the satisfaction of a conjunction of assertions). Otherwise, if the algorithm detects that a justice cannot be satisfied (it returns at line 42 or 47), then we use the justice assumptions as the set, and the second part of the algorithm (lines 40-53) as the monotonic criterion (after *next_persist* and *next_range* have already been initialized in the first part).

NWS core computation is often very effective in localizing the cause for NWS. For the example of List. 1, there are several possible cores that the computation can return. One example would be the assumptions `findStat` and `samePos`. Another possible output consists of the two assumptions `dropCargo` and `clearCargo`. Thanks to our heuristics, as we show in the evaluation, it is typically much faster than the core computation from [20].

6.4 Dealing with Auxiliary Variables

Advanced Spectra structures such as patterns [18] add auxiliary variables to the GR(1) game. To allow T1 and T3 to detect NWS cases that are caused by these structures we adapt them as follows.

Whenever T1 detects an assumption that contains an auxiliary variable *aux* but no environment variables, it checks whether there is another assumption or auxiliary guarantee that references both *aux* and some environment variable. If it is not the case, it should report NWS, since the environment has to control over the values of *aux*. This is accomplished by adding *aux* to a list *aux_vars* whenever such an assumption is encountered. After T1 finishes iterating through the assumptions, it makes a second iteration through all assumptions and auxiliary guarantees. During the second iteration, whenever an assumption or guarantee is encountered that references both an auxiliary variable and an environment variable, that auxiliary variable is removed from *aux_vars*. After the second iteration is completed, if *aux_vars* is not empty, NWS is reported.

We adapt T3 to use *next_range* to store constraints over auxiliary variables in the next state. During the iteration on the safety assumptions, if an assumption ρ_n^e contains auxiliary variables in the next state, the heuristic iterates *next_range*. For each entry $\langle c, S\rangle$, and for each $f \in S$, if $present \wedge c \neq False$, the constraint over auxiliary variables in the next state induced by $\rho_n^e \wedge c \wedge Unprime(f)$ is computed and added to S (unless it is false). The rest of the algorithm continues as usual (this addition is not shown in Alg. 2).

7 EVALUATION

We have implemented the four fixed-point synthesis algorithm of [17] (4FP), our extension with the reduction (R+4FP), and our heuristics on top of the Spectra specification language and IDE [21].

Means to run our implementation, all specifications used in our evaluation, and all data we report, are available in supporting materials for inspection and reproduction [12]. We encourage the interested reader to try them out.

The following research questions guide our evaluation.

RQ1 How does R+4FP compare to 4FP in terms of synthesizing kind controllers?

RQ2 How does R+4FP compare to 4FP and to 3FP in performance?

RQ3 How many NWS specifications do the heuristics detect?

RQ4 What is the distribution between our three heuristics?

RQ5 How does our heuristics for NWS detection compare to previous detection approach in terms of performance?

RQ6 How does our heuristics for NWS core localization compare to previous detection approach in terms of performance?

We now describe the experiments we have conducted in pursuit of answering these questions.

7.1 Specification Corpus

We use the SYNTECH well-known specification benchmarks [21], which contain hundreds of Spectra specifications written by senior undergraduate students in semester-long project classes at Tel Aviv University. The specification sets we use, SYN15, SYN17, and SYN20 contain 61, 122, and 62 specifications resp., a total of 245 specifications, and statistics about their size and complexity appear in [21]. We also use SYN23' (a subset of SYN23), a set of

	Kind Realizability		Time Ratio	
	Kind by R+4FP	Add. unkind by 4FP	4FP	R+4FP
SYN15	29/48	≥ 16	3.34	4.14
SYN17	61/73	≥ 10	3.67	4.81
SYN20	10/11	≥ 1	3.42	3.56
SYN23'	93/186	≥ 76	8.19	4.09

Table 1: Results for RQ1 and RQ2: R+4FP, 4FP, and 3FP

	Heuristics Hit Ratio		Heuristics Distribution			
	Pure	Guars.	T1	T2	T3 (basic)	T3 (full)
SYN15	40/48	26/39	11	21	14	19
SYN17	47/73	12/27	5	3	29	42
SYN20	9/11	6/7	5	0	9	9
SYN23'	84/186	47/186	13	47	17	26

Table 2: Results for RQ3 and RQ4

633 NWS specifications from a recent instance of the same class. We did not use SYN19 in our evaluation, since it contains very few specifications that are NWS.

Note that for experiments that measure time, we use all specifications from these benchmarks, but for experiments that measure hit ratio, we use only the NWS specifications. There are 48, 73, 11, and 186 NWS specifications in each benchmark resp.

Finally, we use several instances of the well-known parametric specifications, AMBA [4] and GenBuf [5], written by expert researchers. We report about them separately in Sect. 7.5.

7.2 Validation and Experiment Setup

We have implemented several tests to validate the correctness of our implementations. First, we created two implementations of 4FP, one in Java and one in C, and wrote a test that checks that they return the same result. Second, we wrote a test that checks that whenever our heuristics for NWS detection reports that a given specification is NWS, it is indeed NWS according to the existing NWS detection implemented in Spectra. We ran these tests over all the specifications in our corpus.

We ran all experiments on an AWS t3.2xlarge instance (representing an ordinary laptop with up to 3.1GHz CPU and 32GB RAM) with Windows 10 64-bit OS, Java 11 64Bit, and CUDD 3 compiled for 64Bit, using one core of the CPU.

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

7.3 Results: R+4FP

Table 1 compares 4FP and R+4FP in terms of hit ratio and running time. It has a row for each benchmark, and its columns are divided into two sections: one to deal with RQ1 and one with RQ2.

In the first section of the table, the first column displays the number of specifications that were found realizable by R+4FP out of the total number of realizable and NWS specifications in the given benchmark. So, for example, in SYN15 there were 48 realizable and NWS specifications, and 29 of them were found realizable by R+4FP. Hence for the remaining 19 specifications, no kind controller exists.

The third column shows the number of specifications that were found realizable by 4FP, but not by R+4FP. As they are unrealizable by R+4FP, no kind controller exists for them. Therefore, the fact they are found realizable by 4FP means that the controller 4FP generates for them necessarily breaks environment safety assumptions, and is therefore unkind. Note that there may be additional unkind controllers generated by 4FP for specifications that R+4FP found realizable. For this reason we title this column "Additional unkind controllers by 4FP". For example, in SYN15, there are at least 16 specifications where 4FP generates such unkind controllers. There are additional 3 specifications (not shown in the table) where both 4FP and R+4FP return that the specification is unrealizable.

In total, R+4FP shows us that from the 296 specifications that 4FP finds realizable, only 193 are kindly realizable, and the remaining 103 specifications cannot be synthesized into a kind controller. Hence, the controllers that are synthesized by 4FP for these 103 specifications necessarily lead the environment into a deadlock.

To answer RQ1, 4FP usually (in 93% of the cases) finds a controller to NWS specifications. However, although these controllers are guaranteed not to violate justice assumptions, they oftentimes violate safety assumptions (in at least 35% of the cases). Our extension, R+4FP, avoids generating such problematic controllers.

The second section of the table deals with the running time of 4FP and R+4FP in comparison to the original 3FP algorithm. We used each of the algorithms to check the realizability of each specification 10 times, and computed the average running time among them. For each specification, we computed the ratio between 4FP and 3FP running times, as well as the ratio between R+4FP and 3FP running times. We report the geometric means of these ratios. So, for example, for the specifications in SYN15, checking realizability by 4FP took on average 3.34 times longer than by 3FP.

We observe that 4FP has a relatively high running time overhead over 3FP - it is more than 3 times slower. That said, we see that the additional overhead caused by our reduction is small. This may be expected, due to the addition of the new variable and the new justice. This answers RQ2.

We conclude that it is beneficial to use R+4FP, as it adds little running time over 4FP (RQ2) and often improves the synthesized controllers and prevents the generation of problematic ones (RQ1).

7.4 Results: Heuristics

Table 2 displays the coverage results for the performance of the heuristics. As before, it contains a row for each of the specification benchmarks. Its columns are divided into two sections, one for RQ3 and one for RQ4.

The first section summarizes the results of the experiments that aim to answer RQ3 on the hit ratio of the heuristics. The two columns show the success ratios for the specifications that are NWS according to the original definition and for the definition of NWS that accounts for guarantees (see Sect. 6.1). For example, in SYN15, out of 48 specifications that are NWS, 40 are detected as such by the heuristics, and out of 39 specifications that are NWS with respect to guarantees, 26 are detected as such by the heuristics. We used the algorithm of [20] as the ground truth for WS of specifications, as it is both sound and complete.

	WS pure definition							WS considering guarantees						
	Heuristics			Via Realizability [20]			Ratios	Heuristics			Via Realizability [20]			Ratios
	min	max	avg	min	max	avg		min	max	avg	min	max	avg	
SYN15	0.09	1.17	0.45	1.81	15.69	5.66	12.84	0.11	1.67	0.52	1.63	581.66	44.83	40.52
SYN17	0.13	156.05	5.24	0.05	1922.23	206.01	29.17	0.13	163.82	3.81	0.03	379700.00	6513.50	127.76
SYN20	0.13	46.91	2.33	1.55	287.97	11.26	5.41	0.17	5.79	0.74	1.61	19729.17	854.10	63.25
SYN23'	0.07	3831.30	68.30	1.47	22062.13	479.54	7.78	0.09	3871.36	70.43	1.41	6368070.00	122429.58	333.58

Table 3: Results for RQ5: NWS detection times; all absolute times are in ms

	WS pure definition							WS considering guarantees						
	Heuristics			Via Realizability [20]			Ratios	Heuristics			Via Realizability [20]			Ratios
	min	max	avg	min	max	avg		min	max	avg	min	max	avg	
SYN15	0.09	2.16	0.80	1.87	24.43	7.49	10.87	0.14	2.61	1.11	1.66	398.02	54.19	30.35
SYN17	0.19	42.04	4.28	2.28	1749.37	323.05	29.66	0.30	41.91	4.56	2.30	361900.00	13290.22	255.29
SYN20	0.14	10.18	3.74	4.80	26.54	13.80	12.37	0.20	13.02	4.21	9.56	107859.55	15833.44	402.76
SYN23'	0.10	1189.67	42.26	2.42	6800.70	288.24	9.62	0.11	1144.07	56.48	1.70	4851000.00	75409.30	118.71

Table 4: Results for RQ6: NWS core computation times; all absolute times are in ms

The second section of the table answers RQ4. It consists of a column for each of the heuristics T1, T2, T3 (basic), and T3 (full). T3 (basic) is the version of T3 explained in Sect. 5.3, and T3 (full) refers to the extended version described in Sects. 6.2 and 6.4. The data in each column shows the number of specifications in each benchmark that are identified as NWS by the respective heuristic. The experiments use the original definition of NWS (we include in the supporting materials the results for the experiments where guarantees are considered). Note that a specification can be detected as NWS by more than one heuristic. For example, in SYN15, T1 detects 11 NWS specifications, T2 detects 21, the basic version of T3 detects 14, and the full version detects 19 NWS specifications. We see that the full version of T3 often detects NWS cases that its basic version does not. Lastly, all of the heuristics are significant, as each detects a non-negligible number of specifications as NWS.

Tables 3 and 4 summarize the experiments that answer RQ5 and RQ6, respectively. Table 3 compares the NWS detection times of the heuristics to the approach of [20] (which does so by reduction to realizability), while Table 4 compares the NWS core computation times. Each of the tables consists of two sections: one for the pure definition of WS and one for the definition that accounts for guarantees. For each of the two settings, we used both methods (heuristics and reduction to realazability [20]) to analyze each specification 10 times, and measured the average time for each. Then, for each of the four specification benchmarks, we report the minimal, maximal, and average running times amongst the averages in that benchmark, in milliseconds. For example, amongst the specifications of SYN15, the minimal average time for detecting NWS with the pure definition using the heuristics was 0.09 milliseconds, while the minimal average time for doing so by reduction to realizability was 1.81 milliseconds. When using the definition of WS that accounts for guarantees, the minimal average time of the heuristics is 0.11 milliseconds, while that of the realizability-based approach is 1.63 milliseconds. Similarly, the minimal average time of computing a NWS core for the specifications of SYN15 was 0.09 milliseconds with the heuristics and 1.87 milliseconds with the realizability-based approach when the pure definition is used, but when the definition that accounts for guarantees is used, the times are 0.14 and 1.66 milliseconds, respectively.

In addition, after calculating the average running time of both approaches for each specification, we calculated the geometric mean between the ratios of these averages. These geometric means are reported for each benchmark under the column "Ratios". For example, on the specifications of SYN15, the heuristics executed NWS detection on average 12.84 times faster than the realizability-based approach when the original definition of WS was used, and on average 40.52 times faster when the definition that considers guarantees was used. They executed core computation on average 10.87 times faster with the original definition of NWS, and 30.35 faster with the definition that considers guarantees.

It is evident that our heuristics are significantly faster than the realizability-based approach, both for NWS detection and for core computation. As seen in the columns "max" and "avg", in many cases, the improvement is from minutes to less than a second. The improvement is even more considerable when taking guarantees into account. This is expected, as the inclusion of guarantees has very little impact on the complexity of our heuristics, while the realizability-based approach becomes much slower when the underlying GR(1) problem becomes more complex. We can therefore conclude that although our heuristics do not always detect NWS, it is beneficial to use them as the first tool to detect NWS.

7.5 Parametric Specifications

AMBA [4] and GenBuf [5] are well-known parametric specifications. Although they are written by expert researchers, we found that they are NWS (we checked 5 instances of different sizes of each). With regard to kind controllers, both 4FP and R+4FP report these specifications as realizable, meaning that although NWS, kind controllers can be constructed for them. With that said, only our approach ensures that the constructed controllers are indeed kind.

As for the heuristics, both specifications are detected as NWS by them. AMBA is detected with T1 and GenBuf is detected with T3.

7.6 Threats to Validity

Several factors can pose a threat to the validity of our results. First, our heuristics for detection of NWS were inspired by common mistakes we observed in the SYNTECH specification corpus. It is possible that the same mistakes do not occur in other specifications, and therefore the heuristics would not have a high success rate on them. To reduce this threat, we tested the heuristics on other NWS parametric specification from the literature, and witnessed that they successfully detect NWS in them as well.

Another threat relates to the variance in running times due to JVM garbage collection and BDD dynamic-reordering. To mitigate this threat, as we write in Sect. 7.2, we performed 10 runs of each experiment and reported average values.

Finally, our implementations may have bugs. To mitigate this threat, we performed a thorough validation. See Sect. 7.2.

8 RELATED WORK

Well-separation for GR(1) environments was introduced by Klein and Pnueli [13] in order to show when the GR(1) semantics of strict and implication realizability agree. Maoz and Ringert [20] extended Klein and Pnueli's work with more fine-grained analysis of cases, strategies, a core, implementation, and evaluation.

Bloem et al. [2] argue that most approaches, e.g., GR(1) synthesis [6], insufficiently handle assumption violations. They suggest and later define [3] cooperative synthesis levels where the highest level ensures that assumptions and guarantees are always satisfied. Ehlers et al. [9] present cooperative GR(1) synthesis for one of the levels that ensures that from every state the environment can satisfy the assumptions. Cooperative synthesis may fail when regular GR(1) synthesis does not.

Majumdar et al. [17] presented a 4 fixed-point algorithm for generation of strategies that do not violate justice assumptions by taking advantage of NWS even if possible. Their work is evaluated on a maze example specification. We extended their algorithm, via a reduction, to also ensure that safety assumptions are not violated. We implemented their algorithm in Spectra. We could not use their implementation or compare it to ours, since their implementation uses the Slugs specification language, while our corpus includes specifications written in Spectra. Due to the lack of language constructs such as patterns in Slugs, the implementations are incomparable.

D'Ippolito et al. [7] 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. In their setup, however, the specifications include only justice assumptions, not safety assumptions.

NWS is one example of a quality issue in specifications. Other quality issues in specifications for reactive synthesis have been discussed in the literature, e.g., inherent vacuity [22]. Perhaps sound but incomplete heuristics similar to ours could be proposed to accelerate inherent vacuity detection.

Finally, our choice to implement our work in Spectra IDE and not other GR(1) synthesizers, e.g., [1, 10], was motivated by Spectra's performance in comparison to these tools [11], its relatively friendly development environment, its support for NWS detection and localization [20], and the many specifications it comes with. Still, our contributions are relevant to GR(1) synthesis in general and are not limited to Spectra.

9 CONCLUSION

We presented two contributions related to NWS specifications for reactive synthesis. In the first contribution, we showed how to generate a kind controller, one that does not exploit NWS. We used a reduction to extend an existing algorithm which does so for specifications with NWS of type E-just, such that it generates kind controllers not only for NWS of type E-just but also for NWS of type E-safe. Our experiments showed that well-known benchmarks include specifications with NWS of both these types, which suggests that in many cases the original algorithm may not be helpful. The reduction adds little running time overhead, and synthesizes a kind controller if and only if such a controller exists.

In our second contribution, we presented sound (but incomplete) heuristics for fast detection and localization of NWS, which capture frequent mistakes that developers make and cause NWS. As witnessed in the evaluation, these heuristics execute very fast, and capture a high percentage of NWS specifications.

Synthesis users can utilize our tools to better tackle the challenge of NWS. First, instead of performing detection of NWS using the reduction to realizability, which often takes a long time to complete, they can use our heuristics to quickly detect and locate the NWS. If the check is positive, they have no need to run the existing slower detection. If the check is negative, they can continue with the complete detection. In addition, fixing NWS is sometimes very difficult. Thus, in cases where engineers find it difficult to fix the NWS of a specification, they can instead try to bypass the problem and synthesize a kind controller that realizes the specification. Synthesis will be slower, but it will ensure that the synthesized controller is kind and does not exploit NWS.

In future work, we hope to reach a point where synthesizing a kind controller has only little time overhead beyond that of a regular one. If it were the case, the regular check for realizability could be replaced by a check for kind realizability, as this is the more useful notion of realizability that engineers should be interested in. Moreover, we wish to further investigate specifications for additional frequent mistakes, in order to increase the hit ratio of the heuristics approach for NWS detection.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (grant No 101069165, SYNTACT).

REFERENCES

[1] 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. http://dx.doi.org/10.1007/978-3-642-14295-6_37

[2] Roderick Bloem, Rüdiger Ehlers, Swen Jacobs, and Robert Könighofer. 2014. How to Handle Assumptions in Synthesis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014. (EPTCS, Vol. 157)*. 34–50. https://doi.org/10.4204/EPTCS.157.7

[3] Roderick Bloem, Rüdiger Ehlers, and Robert Könighofer. 2015. Cooperative Reactive Synthesis. In *Automated Technology for Verification and Analysis*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). Springer International Publishing, Cham, 394–410.

[4] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Automatic hardware synthesis from specifications: A case study. In *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1–6.

[5] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science* 190, 4 (2007), 3–16.

[6] 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. http://dx.doi.org/10.1016/j.jcss.2011.08.007

[7] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. 2013. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (2013), 9:1–9:36. https://doi.org/10.1145/2430536.2430543

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

[9] Rüdiger Ehlers, Robert Könighofer, and Roderick Bloem. 2015. Synthesizing cooperative reactive mission plans. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2015, Hamburg, Germany, September 28 - October 2, 2015*. IEEE, 3478–3485. https://doi.org/10.1109/IROS.2015.7353862

[10] 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

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

[12] Ariel Gorenstein, Shahar Maoz, and Jan Oliver Ringert. 2024. Supporting artifact. https://doi.org/10.5281/zenodo.8312833.

[13] Uri Klein and Amir Pnueli. 2010. Revisiting synthesis of GR (1) specifications. In *Haifa Verification Conference*. Springer, 161–181.

[14] Dexter Kozen. 1983. Results on the propositional μ-calculus. *Theoretical computer science* 27, 3 (1983), 333–354.

[15] 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. http://dx.doi.org/10.1109/TRO.2009.2030225

[16] Dor Ma'ayan and Shahar Maoz. 2023. Using Reactive Synthesis: An End-to-End Exploratory Case Study. In *45th IEEE/ACM International Conference on Software Engineering*. IEEE, 742–754. https://doi.org/10.1109/ICSE48619.2023.00071

[17] Rupak Majumdar, Nir Piterman, and Anne-Kathrin Schmuck. 2019. Environmentally-friendly GR (1) synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 229–246.

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

[19] Shahar Maoz and Jan Oliver Ringert. 2015. Synthesizing a Lego Forklift Controller in GR(1): A Case Stud. In *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015 (EPTCS, Vol. 202)*, Pavol Cerný, Viktor Kuncak, and Parthasarathy Madhusudan (Eds.). 58–72. https://doi.org/10.4204/EPTCS.202.5

[20] Shahar Maoz and Jan Oliver Ringert. 2016. On well-separation of GR (1) specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 362–372.

[21] Shahar Maoz and Jan Oliver Ringert. 2021. Spectra: a specification language for reactive systems. *Software and Systems Modeling* 20, 5 (2021), 1553–1586.

[22] Shahar Maoz and Rafi Shalom. 2020. Inherent vacuity for GR(1) specifications. In *ESEC/FSE*. ACM, 99–110.

[23] Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. 2011. Distributed power allocation for vehicle management systems. In *CDC-ECC*. IEEE, 4841–4848. https://doi.org/10.1109/CDC.2011.6161470

[24] Necmiye Ozay, Ufuk Topcu, Richard M. Murray, and Tichakorn Wongpiromsarn. 2011. Distributed Synthesis of Control Protocols for Smart Camera Networks. In *ICCPS'11*. IEEE Computer Society, 45–54. https://doi.org/10.1109/ICCPS.2011.22

[25] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2005. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 364–380.

[26] 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.

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

[28] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.