2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

Which of My Assumptions are Unnecessary for Realizability and Why Should I Care?

Rafi Shalom
Tel Aviv University
Tel Aviv, Israel

Shahar Maoz
Tel Aviv University
Tel Aviv, Israel

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00030

***Abstract*—Specifications for reactive systems synthesis consist of assumptions and guarantees. However, some specifications may include unnecessary assumptions, i.e., assumptions that are not necessary for realizability. While the controllers that are synthesized from such specifications are correct, they are also inflexible and fragile; their executions will satisfy the specification's guarantees in only very specific environments.**

In this work we show how to detect unnecessary assumptions, and to transform any realizable specification into a corresponding realizable *core specification*, one that includes the same guarantees but no unnecessary assumptions. We do this by computing an *assumptions core*, a locally minimal subset of assumptions that suffices for realizability. Controllers that are synthesized from a core specification are not only correct but, importantly, more general; their executions will satisfy the specification's guarantees in more environments.

We implemented our ideas in the Spectra synthesis environment, and evaluated their impact over different benchmarks from the literature. The evaluation provides evidence for the motivation and significance of our work, by showing (1) that unnecessary assumptions are highly prevalent, (2) that in almost all cases the fully-automated removal of unnecessary assumptions pays off in total synthesis time, and (3) that core specifications induce more general controllers whose reachable state space is larger but whose representation more memory efficient.

I. Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [39]. GR(1) is a fragment of Linear Temporal Logic (LTL) that has an efficient symbolic synthesis algorithm [8]. GR(1) specifications include assumptions and guarantees that specify what should hold in all initial states, in all states and transitions (safeties), and infinitely often on every computation (justices). The expressive power of GR(1) covers almost all well-known LTL specification patterns [15], [25]. It has already been used in several application domains, e.g., to specify and implement autonomous robots [1], [23], [26], [46], control protocols for smart camera networks [37], distributed control protocols for aircraft vehicle management systems [36], and device drivers [41]. Several tools support GR(1) synthesis [4], [17], [29].

Specifications for reactive systems are not easy to write (e.g., [40]). Moreover, while writing the guarantees is difficult, writing the assumptions is even harder, as they are typically based on tacit, implicit domain knowledge (see, e.g., [5], [28]). On the one hand, when one writes too few or too weak assumptions, the specification becomes unrealizable. On the other hand, realizable specifications for synthesis may include too many or too strong assumptions. Indeed, as we show in our evaluation, many specifications from the literature include many more assumptions than necessary for realizability. Intuitively, while the controllers that are synthesized from such specifications are correct, they are also inflexible and fragile; their executions will satisfy the specification's guarantees in only very specific environments. In contrast, controllers that are synthesized from specifications with less assumptions would still be correct and, importantly, more general; their executions will satisfy the specification's guarantees in more environments.

Unnecessary assumptions are hard to manually detect. Moreover, two assumptions, each unnecessary on its own, may not be unnecessary together. Our first contribution is the implementation of a tool that detects them.

Following our ability to detect unnecessary assumptions, we introduce the notion of a *core specification*, one that includes the same guarantees but no unnecessary assumptions. We obtain a core specification from a realizable specification by computing an *assumptions core*, a locally minimal subset of assumptions that suffices for realizability. By definition, all assumptions that are outside the realizable core are unnecessary for realizability. While cores are typically used in the literature for fault localization, our use of cores is different. We use it to detect the necessary constraints for the realizability of the specification at hand.

Beyond the introduction of a tool for the detection of unnecessary assumptions, and the computation of core specifications, the significance of our work is based on two pillars. First, we provide evidence showing that **assumptions that are unnecessary for realizability are prevalent**: almost all (91%) specifications in well-known benchmarks include at least one unnecessary assumption, and in more than 70% of these benchmark specifications, at least half of the assumptions are unnecessary. Second, we show that **the fully-automated removal of unnecessary assumptions from these specifications takes time but pays off**: in almost all cases, removal of unnecessary assumptions plus synthesis from the resulting core specification, is faster than synthesis from the original specification.

In addition, we show that core specifications typically induce more general controllers whose reachable state space is larger but whose representation more memory efficient. We

Work licensed under Creative Commons Attribution 4.0 License.
https://creativecommons.org/licenses/by/4.0/

DOI 10.1109/ICSE48619.2023.00030

further show that applying various analyses to core specifications is typically much faster than applying the same analyses to the original specifications.

We implemented our ideas on top of Spectra, a rich specification language and open source tool set for reactive synthesis [29], [44]. We validated and evaluated our work on different benchmarks from the literature, including the SYNTECH benchmarks, hundreds of specifications written by senior undergraduate students in semester-long project classes, and various well-known specifications taken from the reactive synthesis literature, written by experts and researchers. We present the evaluation in Sect. VI.

It is important to note that the existence of unnecessary assumptions, even in specifications created by experts, is perhaps not surprising. It is well-known that requirements documents typically include different kinds of redundancies [45] and unnecessary assumptions are one example of these. However, absent automated tools to detect unnecessary assumptions, it is practically impossible to notice them. This further motivates our work.

Note that unnecessary assumptions are not necessarily wrong. They may correctly specify the expected behavior of the real environment in which the system will operate. We identify assumptions that are logically unnecessary for realizability, regardless of their correctness w.r.t. the real environment.

Our work relates to previous works on the computation of other kinds of cores for temporal specifications for synthesis, and on other quality issues in specifications (unrealizable cores [22], [32], non-well-separation [27], and inherent vacuity [31]). A definition of assumptions that are unnecessary for realizability appeared in [13], [16]. In [31] the impact of the removal of inherently vacuous specification elements (not only assumptions), showed a significant improvement of synthesis running times in some cases. Since our definition of unnecessary assumptions is broader (because it concerns realizability and does not require semantic equivalence), we can expect to find more of them by definition, with a greater impact on running times. To our knowledge, our work is the first to measure the prevalence of unnecessary assumptions in specifications for synthesis, to compute a realizable core, to use core specifications in order to synthesize more general controllers, and to demonstrate their advantages on existing benchmarks. We discuss related work in Sect. VII.

II. Running Example

As a running example for this paper we use a variant of a popular specification from the literature, presenting a robot that has to evade a moving obstacle. The example is small and simple, to fit the paper presentation. In our evaluation, described in Sect. VI, we have used larger and more complex specifications, taken from benchmarks.

A Spectra specification for the moving obstacle evasion problem is shown in Lst. 1 and Lst. 2. The example setting is an $n \times n$ grid world where a robot moves between cells

Listing 1
Specification: Robot Evading Moving Obstacle
(Definitions, Variables, and Predicates)

```
 1 spec RobotEvadingMovingObstacle
 2
 3 // Define board size
 4 define SIZE := 32;
 5
 6 // Define obstacle docking in lower right corner
 7 define obsDock := (obsX = SIZE-1) & (obsY = SIZE-1);
 8
 9 // EnvVars: Obstacle location and waiting state
10 env Int(1..(SIZE-1)) obsX; env Int(1..(SIZE-1)) obsY;
11 env boolean obsWait;
12
13 // SysVar: Robot location
14 sys Int(1..SIZE) robX; sys Int(1..SIZE) robY;
15
16 // Predicates constraining robot and obstacle movement
17 predicate moveRob(Int(1..SIZE) pos):
18   pos+1 =next(pos) | pos =next(pos) | pos-1 =next(pos);
19
20 predicate moveObs(Int(1..(SIZE-1)) pos):
21   pos+1 =next(pos) | pos =next(pos) | pos-1 =next(pos);
22
23 // Predicates for the positions of objects
24 predicate robAt(Int(1..SIZE) x, Int(1..SIZE) y):
25   robX=x & robY=y;
26
27 predicate obsNotAt(Int(1..SIZE) x, Int(1..SIZE) y,
28   Int(1..(SIZE-1)) obX, Int(1..(SIZE-1)) obY):
29    (x != obX | y != obY) &
30    (x != obX + 1 | y != obY) &
31    (x != obX | y != obY + 1) &
32    (x != obX + 1 | y != obY + 1);
```

(in Lst. 1, $n = 32$ denoted by the constant `SIZE` defined at l. 4). The robot (whose position is encoded by variables `robX` and `robY` at l. 14) must make sure to evade a larger, moving obstacle that occupies 2×2 grid cells (its upper left position is encoded by variables `obsX` and `obsY` at l. 10). The system controls the robot. The environment controls the obstacle. In each step, the obstacle and the robot can stay in place or move to any empty adjacent cell. The robot is more agile and can make two steps upon each step of the obstacle, i.e., the obstacle is forced to wait every other turn (see the variable `obsWait` at l. 11). We define the lower right position as the docking position of the obstacle (see the definition of `obsDock` at l. 7). The obstacle is initially at its docking position, and is assumed to visit this position infinitely often for maintenance.

There are four predicates we define for a more concise and readable specification. Two movement limitation predicates at l. 17 and l. 20 that allow a change of at most one in each coordinate. Predicate `robAt` (l. 24) determines if the robot is at a specific position, and predicate `obsNotAt` (l. 27) determines if a given board position is not one of the four obstacle positions, given its (usual) upper left position.

The above behaviors are encoded into assumptions and guarantees.[1] Six assumptions specify the expected behavior of the obstacle. The obstacle is initially at its docking position (l. 36), and it is expected to reach that position infinitely often during the run (l. 40). The Boolean `obsWait` variable is initially set to false (l. 44), flips every step (l. 48), and forces the obstacle not to move when true (l. 52). Finally,

[1]We use Spectra's `alw` and `alwEv` for LTL `G` and `GF` resp., see [29].

Listing 2
SPECIFICATION: ROBOT EVADING MOVING OBSTACLE (CONT., ASSUMPTIONS AND GUARANTEES)

```
34 // The obstacle is initially docking
35 asm initiallyObstacleAtLowerRightCorner:
36 ini obsDock;
37
38 // The obstacle must not go forever without maintenance
39 asm obstacleMustDockInfinitelyOften:
40 alwEv obsDock;
41
42 // The obstacle is initially not waiting
43 asm initiallyObsWaitFalse:
44 ini !obsWait;
45
46 // The obstacle waits every other turn
47 asm obstacleWaitSwitches:
48 alw (obsWait->next(!obsWait))&(!obsWait->next(obsWait));
49
50 // A waiting obstacle does not move
51 asm obstacleDoesNotMoveWhenObsWait:
52 alw obsWait->(next(obsX)=obsX & next(obsY)=obsY);
53
54 // The obstacle can move only one step in each direction
55 asm obstacleMovesAtMostOne:
56 alw moveObs(obsX) & moveObs(obsY);
57
58 // The robot is initially at top left corner
59 gar initiallyRobotAtTopLeftCorner:
60 ini robAt(1,1);
61
62 // The robot can move only one step in each direction
63 gar robotMovesAtMostOne:
64 alw moveRob(robX) & moveRob(robY);
65
66 // Robot never occupies obstacles's next position
67 gar robotAvoidsObstacle:
68 alw obsNotAt(robX,robY, next(obsX),next(obsY));
69
70 // Robot never occupies obstacle cells
71 gar robotNotOnObstacle:
72 alw obsNotAt(robX, robY, obsX, obsY);
```

we restrict obstacle movement to at most one step in both directions (l. 56).

Four guarantees specify the required behavior of the robot. The robot is initially at the top left corner (l. 60), and its movement has a similar restriction to that of the obstacle (l. 64). Finally, the robot is required to stand neither in any position that the obstacle will occupy at the next step (l. 68), nor any position it currently occupies (l. 72).

A. A Core Specification

The example specification is realizable and can be used in order to synthesize a controller.[2] Interestingly, however, this synthesized controller is unnecessarily specific: two of the assumptions in its specification are unnecessary for realizability.

Using our new tool, the engineer can identify an assumptions core, a locally minimal subset of assumptions that are necessary for realizability, ignore the other, unnecessary assumptions, and use the resulting core specification to synthesize a new controller which is more general; its executions will satisfy the guarantees in more environments.

Specifically in our example, our tool detects an assumption core that consists of the assumptions at lines 36, 48, 52 and 56, deeming the two remaining assumptions at lines 40 and 44 unnecessary for realizability. This means that the controller synthesized from the core specification, where these two unnecessary assumptions are ignored, will satisfy the guarantees in additional environments, specifically in environments where the obstacle may wait in the first state and where it may not need any maintenance.

[2]Indeed, we have used Spectra to do it and to execute this controller against a programmed moving obstacle that satisfies the assumptions in the specification. See the video in [43].

Listing 3
POLE POSITIONED AT (5,5)

```
// The obstacle never covers position (5,5)
asm obstacleNotAtPole:
alw obsNotAt(5, 5, obsX, obsY);

// The robot is never at position (5,5)
gar robotNotAtPole:
alw !robAt(5, 5);
```

B. An Extended Example

Suppose now that there is a pole at position (5,5) which prevents both the robot and the obstacle (any part of it) from occupying that space. Lst. 3 adds the relevant obstacle assumption and robot guarantee. For a specification that consists of Lst. 1 – 3 together, our tool detects a core that includes the assumption in Lst. 3 and the assumptions at lines 36 and 56 in Lst. 2. Put in another way, in this case, our tool finds that the assumptions in lines 40, 44, 48, and 52 are all unnecessary for realizability. This means that when one synthesizes a controller from the core specification, the robot will evade the moving obstacle even in an environment where the obstacle is as agile as the robot!

Finally, note that when the assumptions in lines 40, 44, 48, and 52, are ignored, the variable `obsWait` becomes redundant. Thus, in this sense, the synthesis problem presented to the synthesizer becomes smaller; technically, the GR(1) game presented to the synthesizer will not include this variable.

III. PRELIMINARIES

A. Linear Temporal Logic (LTL)

We use the standard definitions of *linear temporal logic (LTL)*, e.g., as found in [8], over future temporal operators **X** (next), **U** (until), **F** (finally), and **G** (globally), and past temporal operator **H** (historically).

An LTL formula φ is satisfiable iff there is a computation σ s.t. $\sigma \models \varphi$. 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.

B. GR(1) and Realizability

GR(1) is a fragment of LTL. A GR(1) specification contains initial assumptions and guarantees over initial states, safety assumptions and guarantees relating the current and next state, and justice assumptions and guarantees requiring that an assertion holds infinitely many times during a computation. We use the following abstract syntax definition of a GR(1) specification taken from [31].

Definition 1 (Abstract syntax of a specification)**.** *A GR(1) specification is a tuple $Spec = \langle V_e, V_s, D, M_e, M_s\rangle$, where V_e and V_s are sets of environment and system variables respectively, $D : V_e \cup V_s \to Doms$ assigns a finite domain to each variable*[3]*, and M_e and M_s are the environment and system modules. A module is a triplet $M = \langle I, T, J\rangle$ that contains sets of initial assertions $I = \{I_n\}_{n=1}^{i}$, safety assertions $T = \{T_n\}_{n=1}^{t}$, and justice assertions $J = \{J_n\}_{n=1}^{j}$ of the module, where $i = |I|, t = |T|$ and $j = |J|$. The set of elements of module $M = \langle I, T, J\rangle$ is $B_M = I \cup \{\mathbf{G}\, T_i\}_{i=1}^{t} \cup \{\mathbf{GF}\, J_i\}_{i=1}^{j}$.*

Given a set $\mathcal{Z}$ of variables, $\mathcal{Z}' = \{\mathbf{X}v | v \in \mathcal{Z}\}$ contains a copy of its variables at the next state. Let $M_e = \langle I_e, T_e, J_e\rangle$, $M_s = \langle I_s, T_s, J_s\rangle$, and $\mathcal{V} = V_e \cup V_s$. Then, the elements of I_e, T_e, J_e, I_s, T_s and J_s are propositional logic expressions over $V_e, \mathcal{V} \cup V_e', \mathcal{V}, \mathcal{V}, \mathcal{V} \cup \mathcal{V}'$ and $\mathcal{V}$ respectively.

GR(1) has efficient symbolic algorithms for realizability checking and controller synthesis, presented in [8], [38]. For this a game structure of a two-player game $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi\rangle$ is defined. The GR(1) game has a set of variables $\mathcal{V} = V_e \cup V_s$, environment and system variables ($\mathcal{X} = V_e$ and $\mathcal{Y} = V_s$ resp.), environment and system initial states ($\theta_e = \wedge_{d \in I_e} d$ and $\theta_s = \wedge_{d \in I_s} d$ resp.), environment and system transitions ($\rho_e = \wedge_{t \in T_e} t$ and $\rho_s = \wedge_{t \in T_s} t$ resp.), and a winning condition $\varphi = \bigwedge_{j \in J_e} \mathbf{GF} j \to \bigwedge_{j \in J_s} \mathbf{GF} j$.

A GR(1) specification is realizable, i.e., allows an implementation, iff the system wins the game. Roughly, this means that if the environment keeps all initial assumptions then the system should keep all initial guarantees, as long as the environment keeps all safety assumptions the system should keep all safety guarantees, and in all infinite plays, if the environment keeps all justice assumptions the system should keep all justice guarantees.

For this the algorithm of [8], [38] computes a winning region which is a set of winning states from which the system has a winning strategy. A winning strategy prescribes the outputs of a system for all possible environment choices that allows the system to win. The winning region is computed according to a fixed-point computation over transitions and justices alone. GR(1) realizability checks if for all initial environment choices the system can enter a winning state. The complexity of realizability checking is $O(nmN^2)$, where N is the size of the state space $2^{\mathcal{X} \cup \mathcal{Y}}$ and n and m are the number of justice assumptions and guarantees resp. GR(1) synthesis computes a winning strategy, if one exists.

C. Reachable States and Reachability Diameter

A synthesized controller can be viewed as a Kripke structure $M = (S, I, T, L)$, where S is a set of states, $I \subseteq S$ are the initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \to 2^{\mathcal{V}}$ labels states with assignments to variables in $\mathcal{V}$.

Symbolic controller synthesis [8], [38] represents the resulting controller symbolically, using two BDDs, one for the set of initial states over variables $\mathcal{V}$, and one for the transition relation over variables $\mathcal{V} \cup \mathcal{V}'$.

The number of reachable states of a controller M is the number of states reachable by a finite number of applications of the transition relation T, starting from any initial state.

The reachability diameter $rd(M)$ of a controller M is the minimum number of transitions required in order to reach all reachable states from the initial states (see its definition in [3]).

IV. Defining and Computing a Core Specification

To transform a given realizable specification into a corresponding core specification, one that includes the same guarantees but no assumptions that are unnecessary for realizability, we formally define and then show how to compute an assumption core, a locally minimal subset of the assumptions that suffices for realizability.

A. Defining an Assumption Core

We start by defining the general notion of a core and then continue to the specific definition of an assumption core.

Given a set E, and a monotonic criterion on subsets of E, a core is a local minimum that satisfies the criterion. Formally:

Definition 2 (Monotonic criterion)**.** *A Boolean criterion over subsets of E is monotonic iff for any two sets A, B such that $A \subseteq B \subseteq E$, if A satisfies the criterion then B satisfies the criterion.*

Definition 3 (Core)**.** *Given a set E and a monotonic criterion over its subsets, a set $C \subseteq E$ is a core of E iff C satisfies the criterion, and all its proper subsets $C' \subset C$ do not satisfy the criterion.*

Realizability is monotonic w.r.t. subsets of assumptions, i.e., adding assumptions to a realizable specification keeps it realizable. Intuitively, this is so because adding assumptions strengthens the constraints on the environment, and does not change the constraints on the system. Formally:

Proposition 1 (Realizability is monotonic)**.** *Given two specifications, $Spec_1 = \langle V_e, V_s, D, M_e^1, M_s\rangle$ and $Spec_2 = \langle V_e, V_s, D, M_e^2, M_s\rangle$, such that $B_{M_e^1} \subseteq B_{M_e^2}$. Then, if $Spec_1$ is realizable, $Spec_2$ is also realizable. Conversely, if $Spec_2$ is unrealizable then $Spec_1$ is unrealizable.*

We can now define an assumptions core and a core specification.

Definition 4 (Assumptions Core and Core specification)**.** *Consider a realizable specification, $Spec = \langle V_e, V_s, D, M_e, M_s\rangle$. An assumptions core is a core $C \subseteq B_{M_e}$ given realizability as the monotonic criterion. The specification $Spec' = \langle V_e, V_s, D, M_e', M_s\rangle$ such that $C = B_{M_e'}$ is a core specification of $Spec$.*

Example 1. *Recall the running example in Sect. II. Then, the assumptions of the original specification B_{M_e} have a core $C \subseteq B_{M_e}$, namely the assumptions at lines 36, 48, 52, and 56. These assumptions are the assumptions $C = B_{M_e'}$ of the core specification.*

[3]The use of any finite domain rather than only Boolean variables is straightforward and supported by many tools, including Spectra.

B. Computing an Assumptions Core

To compute an assumptions core, we use delta debugging [47] (`DDMin`), which finds a core of a set, given a monotonic criterion. `DDMin` recursively checks whether subsets of the original set meet the criterion. In our case, the set E is the original set of assumptions (namely, B_{M_e}) and the criterion is realizability. The complete description of `DDMin` is available in the supporting materials [43].

`DDMin` has been used in the literature to compute guarantees cores in unrealizable specifications [22], [32], but has not been previously used to compute an assumptions core.

`DDMin` has quadratic worst-case complexity and logarithmic best-case complexity in terms of $|E|$, i.e., its worst-case complexity is $O(|E|^2)$ and its best-case complexity is $O(log|E|)$.

Remark 1. *We chose to use `DDMin`, after we empirically compared its performance with three other algorithms. See Sect. VII (related work).*

V. Potential Impact of Core Specifications

We now discuss the potential impact of using core specifications instead of ones that have unnecessary assumptions. In Sect. V-A we motivate the expected impact in terms of computation times. In Sect. V-B we consider quality aspects of the controllers that are synthesized from them, specifically, reachable state space, reachability diameter, memory usage, and storage. In Sect. V-C we consider quality aspects of the specifications themselves, specifically, non-well-separation and unsatisfiability. In Sect. VI we will report on the results of measuring all of these for specifications and core specifications on a large corpus of benchmark specifications.

A. Computation Times

First and foremost, we expect specifications with less assumptions, especially justice assumptions, to induce faster computation times. For example, recall that the complexity of realizability checking is $O(nmN^2)$, where N is the size of the state space $2^{\mathcal{X}\cup\mathcal{Y}}$ and n and m are the number of justice assumptions and guarantees resp. That is, it is linear in the number of justice assumptions. Moreover, core specifications may include less variables, thus reducing the size of the state space. Similarly, running times of synthesis (controller construction), satisfiability, well-separation, and inherent vacuity, are expected to improve as well.

Thus, it is important to examine whether the above theoretical improvement in computation times indeed occurs in practice.

B. Properties of Controllers

We consider the following quality properties of controllers related to their size, simplicity, and efficient use: reachable state space, reachability diameter, memory usage, and storage.

One measure of the size of a controller is the size of its reachable state space, i.e., number of states reachable in runs starting from any initial state.

Removing unnecessary assumptions from a specification makes the environment less restricted. This may result in a controller that has a larger number of reachable states, because the environment is allowed to reach states that the original specification prevented it from reaching. However, it may also result in a controller that has a smaller number of reachable states, depending on the computed core and the synthesizer's chosen strategy. Thus, it is important to examine the actual impact of core specifications in this regard.

The reachability diameter (see Sect. III-C) is a measure of the simplicity of the controller. Again, it is interesting to examine the impact of removing unnecessary assumptions in this regard.

Example 2. *Recall the running example in Lst. 1 – 2 and the additional assumption in Lst. 3. Assume that based on the core composed of the assumptions in l. 36, l. 56, and the one in Lst. 3, we remove the assumptions at lines 40, 44, 48, and 52, to obtain a core specification. Recall that this makes the variable `obsWait` obsolete. In this case, the reachable state space is reduced from 3.1E7 to 1.42E7, i.e., by a factor of* ≈ 2. *The reachability diameter also shrinks from 64 to 32.*

If we begin with a specification that includes also the assumption in Lst. 4, and remove unnecessary assumptions based on the core composed of the above three assumptions, the reachable state space expands from 1.07E7 to 1.42E7 while the reachability diameter shrinks from 89 to 32.

The example above further motivates to examine the actual impact core specifications may have on the size and simplicity of synthesized controllers.

The memory usage and storage of a synthesized controller are additional important measures of its efficient representation and use. When one actually executes a synthesized controller, in practice, disk space is a challenge because the controller typically runs on a rather weak machine with limited power (robot, IoT device, not the machine where synthesis was executed), and disk space affects controller load time and memory usage, see [33].

The memory usage and storage of a synthesized controller depend on its representation. In our context of GR(1) synthesis, the synthesized controller is represented symbolically using two BDDs, one for the set of initial states, and one over variables and their next state copies, representing the transition relation. Thus, as a measure of the memory usage and storage of the controller we consider the number of nodes in these two BDDs and the size they occupy when stored on disk.[4]

Finally, recently, a more efficient representation of the result of GR(1) synthesis was suggested in [33], which avoids the computation of the symbolic controller and instead computes next states "just-in-time", only on demand, as they become necessary during execution. We measure the storage of the JIT controller by the size of the space it takes on disk (after variable reordering, see above).

[4]As these measures depend on BDDs' variable order, when we compute them in our evaluation, we use a reordering heuristic before we count the nodes and save to disk, see Sect. VI.

Listing 4
SMART OBSTACLE ASSUMPTION

```
asm obstacleIsSmart:
alw !obsWait ->
  ((obsX > robX -> next(obsX) = obsX - 1) &
   (obsX < robX - 1 -> next(obsX) = obsX + 1) &
   (obsY > robY -> next(obsY) = obsY - 1) &
   (obsY < robY - 1 -> next(obsY) = obsY + 1));
```

C. Quality Aspects of The Specifications

1) Non-Well-Separated Specifications: Some specifications are realizable because the system can use a strategy that makes the environment fail instead of fulfilling its guarantees.[5] Such a specification is called non-well-separated [21], [27].

Removing unnecessary assumptions may turn a non-well-separated specification into a well-separated one. However, unlike realizability, non-well-separation is not monotonic w.r.t. assumptions. Specifically this means that removing assumptions from a well-separated specification may also turn it into a non-well-separated one. See Theorem 1 in [27].

2) Unsatisfiable Specifications: A specification may be unsatisfiable, meaning that there is no computation that can satisfy both its assumptions and its guarantees. Formally, a specification $Spec = \langle V_e, V_s, D, M_e, M_s \rangle$ is unsatisfiable, if the LTL formula $\wedge_{\alpha \in B_{M_s} \cup B_{M_e}} \alpha$ is unsatisfiable. Though perhaps counter-intuitive, there are realizable yet unsatisfiable specifications. For example, it could be that the assumptions are already unsatisfiable, regardless of the guarantees. Such specifications are vacuously realizable.

Removing unnecessary assumptions cannot turn a satisfiable specification into an unsatisfiable one, so core specifications will not exacerbate the problem of unsatisfiability. However, removing unnecessary assumptions may, in some cases, turn an unsatisfiable specification into a satisfiable one. Thus, one may expect a potential positive impact in this regard.

VI. EVALUATION

We have implemented our ideas on top of Spectra [29], [44], with the `DDMin` performance heuristics from [18].

Means to run our implementation, all specifications used in our evaluation, and all data we report on below, are available in supporting materials for inspection and reproduction [42], [43]. We encourage the interested reader to try them out.

The following research questions guide our evaluation.

R0 Are specifications with assumptions that are unnecessary for realizability prevalent?

R1 Does the computation of core specifications pay off in terms of synthesis times?

R2 How do core specifications impact controller's size and simplicity?

R3 How do core specifications impact analyses running times?

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

[5]In event-based systems this was termed anomalous behavior [14]

A. Corpus of Specifications

We use two different types of specifications, `SYNTECH` benchmarks, which contain hundreds of specifications written by undergraduate students in project classes, and three sets of parametric specifications, written by experts and researchers. All are well-known and frequently used in the literature.

We use four `SYNTECH` benchmarks (15, 17, 19, and 20) [18], [44], which include specifications written by 3rd year undergraduate computer science students in semester-long project classes taught by the authors of [18]. Benchmarks `SYN15R` and `SYN17R` contain specifications of 10 autonomous Lego robots, and benchmarks `SYN19R` and `SYN20R` contain specifications of Java simulations the students wrote, e.g., for a four-way traffic light system. Thus, all `SYNTECH` specifications were written for the purpose of running an actual robot or a Java application, which made them useful for generating actual code. The `SYNTECH` benchmarks have been extensively used for evaluation purposes in the GR(1) literature, e.g., in [9], [12], [24], [30], [31], [32], [35]. According to [29], in these four `SYNTECH` benchmarks, the number of lines of specification (including comments) ranges from 32 to 602 (medians 201, 414, 169, 216.5 resp.); the total number of Boolean variables, i.e., the variable count after the translation to the Spectra kernel, ranges from 4 to 87 (medians 29.5, 45, 36, and 39 resp.); and the total number of assumptions and guarantees, excluding any assumptions or guarantees created by translations to the kernel, ranges from 5 to 105 (medians 42.5, 48, 31.5, 60.5 resp.). These statistics show that the `SYNTECH` benchmarks are neither small nor trivially simple. Note that as time progresses, Spectra specifications include more complicated constructs, and require more computational resources in order to synthesize and analyze.

In total, the specification sets we use, `SYN15R`, `SYN17R`, `SYN19R`, and `SYN20R` contain 61, 113, 18, and 57 specifications resp., **a total of 249 specifications**.

When considering controller properties and synthesis times below, we look at subsets of the above realizable specification sets that are both well-separated and whose core specification is also well-separated. The rationale behind this restriction of the specification sets in this context is that controllers synthesized from non-well-separated specifications may not be useful, see [27]. Specification sets `SYN15W`, `SYN17W`, `SYN19W`, and `SYN20W` contain 21, 64, 17, and 47 specifications resp., **a total of 149 specifications**.

We further use parametric specifications written by experts widely used in the literature [2], [6], [7], [8], [9], [10], [11], [12], [13], [18], [31], [33]:

- 4 `AMBA` [6] specifications, of different sizes (1 to 4 masters) (described in [13]).
- 4 `GENBUF` [7] specifications (5, 10, 20, and 30 senders)
- 4 `LIFT` specifications, from the original GR(1) reactive synthesis paper [8] (10, 20, 40, and 80 floors).

Note that the `AMBA`, `GENBUF`, and `LIFT` specifications, were written by formal methods researchers in order to demonstrate reactive synthesis and examine synthesizers' performance, rather than to generate code for an actual system. Moreover, they are parametric and generated by scripts (for a given number of masters / senders / floors). Thus, one may question to what extent they are representative of specifications that engineers would write in practice. It is still interesting to find if specifications written by experts contain assumptions unnecessary for realizability, and to observe the impact of their removal on running times of analyses. Regardless of the limitations of these specifications, we report on the effect of core specifications on their synthesis time and on controllers synthesized from them.

B. Validation and Experiments Setup

We have implemented an independent automatic test to check that every assumption core that we found is indeed a locally minimal subset of the assumptions that maintains realizability. We ran this test over all the specifications in our corpus.

We ran all experiments on an ordinary PC, Intel Xeon W-2133 CPU 3.6GHz, 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 use are average values of 10 runs, measured by Java 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.

C. Results: Unnecessary Assumptions in Specifications

Table I shows information about assumptions and assumption cores. Each of the first four rows contains the data for one of the sets of realizable `SYNTECH` specifications, and the last three rows contain information about parametric specifications. Column `#Spec` shows the number of specifications in the set. Columns `#Has` and `#Most` show how many specifications have at least one unnecessary assumption, and in how many of the specifications at least half of the assumptions are unnecessary, resp. The columns under `#Asm` show the average number of assumptions in the original specifications, and in the core specifications under `Orig.` and `Core` respectively. The columns under `#J.Asm` show similar data for justice assumptions in particular.

The results show that over 91% of the `SYNTECH` specifications have unnecessary assumptions (228 out of 249), and that in over 70% of them at least half of the assumptions are unnecessary (180 out of 249).

The average number of assumptions on the `SYNTECH` specifications drops from 10.96 to 3.4, a 69% reduction. Importantly, the justice assumptions average drops by a similar factor from 4.05 to 1.26. Justice assumptions are known to contribute the most to running times (see Sect. V-A).

For the parametric specifications, in all except one specification (11 out of 12), more than half of the assumptions are unnecessary. All of the assumptions in the `LIFT` specifications are unnecessary, and 74% of the assumptions in `AMBA` specifications are unnecessary. Both results exceed the rate of unnecessary assumptions in `SYNTECH` specifications. Given that

Table I
PREVALENCE OF UNNECESSARY ASSUMPTIONS

Spec Set	#Spec	#Has	#Most	#Asm		#J.Asm	
				Orig.	Core	Orig.	Core
SYN15R	61	57	46	5.47	2.36	2.91	1.19
SYN17R	113	105	79	11.71	3.76	4.34	1.36
SYN19R	18	16	14	15.38	4.94	2.83	1.88
SYN20R	57	50	41	13.94	3.24	5.08	0.96
AMBA	4	4	4	12.50	3.25	2	1.5
GENBUF	4	4	3	61.75	26.25	2	2.0
LIFT	4	4	4	112.50	0	0	0.0

Table II
COMPUTATION TIME OF ASSUMPTION REMOVAL PLUS SYNTHESIS FROM THE CORE SPEC. VS. SYNTHESIS FROM THE ORIGINAL SPEC.

Spec Set	0.1-1s	1-10s	10-100s	$\geq 100s$
SYN15W	1.76	-	-	-
SYN17W	0.80	0.24	1.23	-
SYN19W	0.74	0.92	0.85	0.19
SYN20W	0.61	1.42	0.80	0.75

Spec Set	Ratio
AMBA	1.03
GENBUF	18.32
LIFT	0.30

these specifications were written by experts, this is compelling evidence for the prevalence of unnecessary assumptions. In this case most justice assumptions are necessary, though there are originally very few or none of them.

To answer R0: For `SYNTECH` specifications 69% of all assumptions (and in particular 69% of the computationally expensive justice assumptions) are unnecessary for realizability. Unnecessary assumptions exist in almost all specifications in our corpus. In over 70% of the specifications at least half of the assumptions are unnecessary for realizability. In specifications written by experts most or all of the assumptions are unnecessary. Thus, assumptions unnecessary for realizability are highly prevalent.

D. Results: Impact on Synthesis Times

Finding a core specification takes time but results in simpler and smaller specifications. Does it pay off?

Table II compares the running time of assumptions removal plus synthesis from core specification vs. the running time of synthesis from the original specification. Synthesis is done in two parts. First, a realizability computation gathers the information about relevant fixed-points, and then this information is used in order to synthesize the controller symbolically.

The numbers in Table II are the geometric means of the ratios between the core computation plus core specifications synthesis running times divided by the synthesis running times for the corresponding original specifications[6]. Table II (left) is dissected into ranges according to the running times of the original specifications. The ranges are from 0.1 seconds to 1 second, from 1 second to 10 seconds, from 10 seconds to 100 seconds, and over 100 seconds. We use '-' for ranges that include no relevant specifications. In Table II (right) we have one such ratio for each set of four specifications.

[6]We use geometric means as appropriate when considering ratios, so that, e.g., the mean of 2.0 and 0.5 would be 1.0 and not 1.25.

Table III
CONTROLLER PROPERTIES (CORE VS. ORIG. RATIO)

Spec Set		Cont. Prop.		Cont. Size			JITC
		States	Diam	Disk	Ini	Tran	Disk
SYN15W	Q1	1	0.66	0.82	0.86	0.81	0.76
	Q2	1.02	1	0.92	0.91	0.93	0.95
	Q3	1.66	1	0.99	1	1	1.03
SYN17W	Q1	1	0.57	0.30	0.67	0.29	0.26
	Q2	1	0.83	0.96	0.94	0.96	0.81
	Q3	12.99	1	1.03	1	1.03	1.01
SYN19W	Q1	2.46	1	0.69	0.69	0.7	0.63
	Q2	7.94	1	0.82	1	0.79	0.84
	Q3	18.86	1	1	1	1.01	0.94
SYN20W	Q1	0.29	1	0.45	0.73	0.43	0.39
	Q2	0.77	1	0.82	0.98	0.77	0.71
	Q3	1	1	0.99	1.09	1	1.79
AMBA	GM	0.25	0.76	0.51	0.77	0.51	0.72
GENBUF	GM	1.11	0.88	1.39	0.72	1.39	0.96
LIFT	GM	1	0.97	0.06	0.38	0.05	0.42

Numbers lower than 1 show better performance. The lowest the ratios, the better it is to compute a core and synthesize using the core specification. For example, the number 0.19 on the third row in column $\geq$`100s`, is the geometric mean of the ratios of the above running times of `SYN19W` specifications for which the running time of synthesis given the original specification takes over 100 seconds. Thus, the value 0.19 indicates a five times improvement in total synthesis times on average.

Parametric specification show either a strong improvement (for the `LIFT` specifications), or no significant change (for the `AMBA` specifications). We discuss the outlier for `GENBUF` in Sect. VI-G.

The results show that despite the time spent on detecting the assumptions core, it is almost always faster to remove the unnecessary assumptions and synthesize a controller from the core specification than to synthesize directly from the original specification.

To answer R1: Computing a core specification pays off. For specifications that originally take over 100 seconds to synthesize, core computation plus core specification synthesis is up to five times faster.

E. Results: Impact over Controller Properties

Table III compares between controllers synthesized from an original specification, and those synthesized from the corresponding core specifications. All the numbers in the table are geometric means of ratios between the values of the core specifications and that of the original specification.

In the two columns under `Cont. Prop.` we show results of two properties of the symbolic controller. Column `States` shows the ratios of the actual reachable state space represented by the BDDs of the symbolic controllers. Note that those states are symbolically represented and not enumerated, which means that they are states that may occur within runs of the controller, but they do not each require memory as individual states. Column `Diam` shows the ratios between the reachability diameter of the controllers.

For each set we measure the impact over the size of the symbolic controller, which appears in the columns under `Cont. Size`. Column `Disk` shows that ratios of the disk space the controller occupies. Columns `Ini` and `Tran` show the ratios of the number of nodes in the BDD (symbolic) representation of the initial states and transition relation resp.

Finally, column `JITC` has the ratio for the disk space required to store just-in-time controller.

For each `SYNTECH` specification set, there are three rows showing quartiles. Rows labeled `Q#` show quartiles from lower to higher, e.g., `Q1` is the lower 25% value and `Q2` is the median. For example, the number 0.83 in the fifth row under `Diam` contains the median of the reachability diameter size of the symbolic controller of the core specification divided by the reachability diameter size of the controller of the original specification. This means that in `SYN17W` specifications, half of the symbolic controllers of the core specification have a reachability diameter at most 83% the size of the reachability diameter of the controllers of the original specification. For each parametric set there is one line showing the geometric mean of the ratio between the value for the original specification divided by the value for the core specification.

The results show that while the size of the reachable state space of the controller may become bigger (e.g., in most `SYN19W` specifications) or smaller (e.g., in most `SYN20W` specifications), with a tendency towards becoming bigger, the reachability diameter remains mostly the same with an inclination to become smaller. Thus, even though the controllers usually have more states, their behavior tends to become simpler.

This simplicity is reflected in the symbolic representation of the controller, and in the size of the JIT controller. The median disk space of both controllers, and of the number of nodes in the BDDs representing the transition relation and set of initial states is never bigger for the core specification. The median reduction in all these values reaches up to about 30%.

To answer R2: Controllers that are synthesized from core specifications take less space, and are simpler than the ones synthesized from the original specifications.

F. Results: Impact on Analyses Times

Table IV shows the effect of removing unnecessary assumptions on realizability checks, symbolic controller synthesis, and just-in-time controller synthesis times resp. Here, we report synthesis times given the output of the realizability checks, i.e., not counting the realizability checking time. All three categories have the same structure as Table II (left), i.e., showing the geometric means of the ratios between computation times for the core specification and for the original specification. For example, the number 0.02 on the second row in column `10-100s` under `Synthesis Computation`, is the geometric mean of the ratios of the above running times of `SYN17W` specifications for which the running time of synthesis (without realizability checking) given the original specification takes between 10 and 100 seconds. Thus, the value 0.02 indicates a 50 times improvement in synthesis times on average.

We observe that realizability checks are typically several times faster for core specifications. Synthesis from core specifications is on average also much faster, for both symbolic and JIT controllers. The improvement is more noticeable for JIT controller synthesis.

Table V shows the effect of removing unnecessary assumptions on running times of well-separation checks, satisfiability checks, computation of all specification elements (i.e., assumptions and guarantees) inherent vacuities resp. (an element of a specification is inherently vacuous if its removal does not change the semantics of the specification, see [31]). The table has the same structure as Table IV.

We observe that in almost all cases running times improve. This improvement is especially notable for checking well-separation, for which the improvement typically reaches several orders of magnitude.

Finally, Table VI shows the change in running times of analyses of parametric specifications, in rows `AMBA`, `GENBUF`, and `LIFT`. Columns `Real.`, `Synt.`, `WS`, `Sat.`, and `Vac.` show results for realizability checks, synthesis, well-separation, satisfiabilty, and vacuities respectively. All numbers are geometric means of running times of the core specifications divided by the running times of the original specifications. Numbers lower than 1 show better performance. For example, the number 0.01 in row `GENBUF` and column `WS` shows that the average well-separation check over the four `GENBUF` specificatins runs 100 times faster on the core specification. The results show that running times typically become several times faster.

To answer R3: Analyzing a core specification is typically several times faster and up to several orders of magnitude faster than similar analyses over the original specification.

G. Discussion and Additional Results

The evaluation provides strong evidence that (1) unnecessary assumptions are prevalent in specifications written by experts and non-experts alike, that (2) removing unnecessary assumptions makes synthesis and other analyses faster, and that (3) controllers synthesized without unnecessary assumptions not only work correctly in more environments (by definition), but are also simpler and take less memory to represent. We consider the value of point (2) above to be critical for the future of GR(1) synthesis. Since synthesis is computationally hard, slow specifications are either limited or completely abandoned. Making existing tools perform better is thus essential for both the creation and usefulness of future specifications.

Our experiments include one outlier to the above positive results, the `GENBUF` specification. Although all `GENBUF` specifications have many unnecessary assumptions (about 35 out of 61 on average, see Table I), the impact of their removal on total synthesis running time (including core computation) is negative (see Table II). That said, while the impact on controller properties and synthesis time alone is negative as well, the impact on other analyses is positive (see Tables III and VI). We are not sure what is the reason for this anomaly.

One may suggest that some of the assumptions we detected are not unnecessary for realizability but instead are part of a specification where the set of guarantees is incomplete. However, since the SYNTECH specifications we used for the evaluation were synthesized and applied for actual Lego robots and Java applications, the functionality of the resulting controllers was examined in practice (actual executions observed by the students who wrote the specifications), and since the parametric specifications we considered were written by experts in order to serve as benchmarks for synthesis, we have some confidence regarding the completeness of the guarantees.

Finally, we examined the impact of core specifications on quality aspects, well-separation and satisfiability. Recall that removing unnecessary assumptions may turn a non-well-separated specification into a well-separated one. However, it may also turn a well-separated specification into a non-well-separated one. Thus, the potential impact in this regard may vary. Also recall that removing unnecessary assumptions cannot turn a satisfiable specification into an unsatisfiable one, however, it may turn an unsatisfiable specification into a satisfiable one. Thus, one may expect a potential positive impact in this regard. See Sect. V-C.

In our experiments we found that the core specifications of all unsatisfiable specifications in our corpus are satisfiable. Thus, the core can assist the engineer in removing constraints while keeping the guaranteed functionality of the controller. We further found that in all benchmark sets (except one, `SYN17W`), the impact on well-separation is positive.

Table VII compares satisfiability and well-separation between the original corpus specifications and their corresponding core specifications. Each of the first four rows contains the data for one of the sets of realizable specifications, and the last row contains information about the entire corpus. Column `N` shows the number of specifications in the set. The columns under `#Sat` show the number of satisfiable original specifications and satisfiable core specifications under `Orig.` and `Core` resp. The columns under `#WS` show the number of well-separated original specifications under `Orig.`, the number of specifications improved from non-well-separated to well-separated under `I`, and the number of specifications damaged by becoming non-well-separated under `D`.

H. Threats to Validity

We discuss internal and external threats to the validity of our results. One internal threat is that symbolic computations are not trivial and our implementation of `DDMin` may contain bugs. To mitigate this threat, we performed a thorough validation using all specifications available to us, see Sect. VI-B. Another internal threat relates to the variance in running times due to JVM garbage collection and BDD dynamic-reordering. To mitigate this threat, although the algorithms we deal with are deterministic, we performed 10 runs of each experiment and report average values, see Sect. VI-B.

Table IV
COMPUTATION TIMES OF REALIZABILITY AND SYNTHESIS (CORE VS. ORIG. RATIO)

Spec Set	Realizability Checks				Synthesis Computation				JIT Synthesis Computation			
	0.1-1s	1-10s	10-100s	$\geq 100s$	0.1-1s	1-10s	10-100s	$\geq 100s$	0.1-1s	1-10s	10-100s	$\geq 100s$
SYN15W	0.35	-	-	-	0.68	-	-	-	0.92	-	-	-
SYN17W	8.1E-3	0.06	-	-	0.61	0.20	0.02	-	0.15	0.01	-	-
SYN19W	0.02	0.21	-	0.20	0.78	0.95	0.79	0.14	0.83	0.44	-	0.38
SYN20W	0.01	0.49	-	-	0.43	0.80	0.76	0.63	0.20	0.10	-	-

Table V
COMPUTATION TIMES OF ADDITIONAL ANALYSES (CORE VS. ORIG. RATIO)

Spec Set	Well-Separation				Satisfiability				Vacuities			
	0.1-1s	1-10s	10-100s	$\geq 100s$	0.1-1s	1-10s	10-100s	$\geq 100s$	0.1-1s	1-10s	10-100s	$\geq 100s$
SYN15W	-	-	-	-	-	-	-	-	-	-	-	-
SYN17W	0.03	2.2E-3	-	-	0.18	0.02	-	-	0.10	0.04	1.8E-3	-
SYN19W	6.9E-3	2.3E-3	1.1E-3	-	0.98	1.10	0.48	0.10	-	0.89	0.82	0.09
SYN20W	5.6E-3	4.2E-4	9.9E-5	-	0.67	0.90	0.28	-	0.25	-	0.28	-

Note that an assumptions core is a local minimum and thus may not be unique. As an example, the specification that combines Lsts. 1-3 from Sect. II has two assumptions cores. First, the assumptions at lines 36, 48, 52, and 56, and second, the assumptions at lines 36, 56, and line 3 from Lst. 3. Thus, another internal threat is that the results may vary depending on the computed core. To address this concern, we computed all assumption cores in our corpus (using a variant of `Punch` [32], applied to subsets of assumptions), and found that in our case, 77% of all SYNTECH specifications (192 out of 249) have a single core. This means that in most cases the computed assumptions core is independent of the choice and implementation of the minimization algorithm.

An external threat relates to the possible generalization of the results. We have based the evaluation on a corpus consisting of the `SYNTECH` specifications and parametric specifications from the literature, see Sect. VI-A. We do not know to what extent these sets of specifications are representative of specifications engineers would write in practice. Still, we believe that the use of many specifications from several different sources and of different nature, partly mitigates the generalization threat.

VII. RELATED WORK

The existence of unnecessary assumptions. Unnecessary assumptions are one example of the more general well-known phenomenon of redundancies in requirements (see, e.g., [45]). However, our work is the first to detect these in formal specifications for synthesis and to measure the impact of removing them.

Table VI
COMPUTATION TIMES OF ANALYSES (CORE VS. ORIG. RATIO)

	Real.	Synt.	WS	Sat.	Vac.
AMBA	0.51	0.46	0.15	0.99	1.07
GENBUF	0.61	3.07	0.01	0.53	0.37
LIFT	0.19	0.16	0.03	0.02	8.8E-3

Table VII
IMPACT OF CORES OVER QUALITATIVE ASPECTS OF SPECIFICATIONS: SATISFIABILITY AND WELL-SEPARATION

Spec Set	N	#Sat		#WS		
		Orig.	Core	Orig.	I	D
SYN15R	61	53	61	22	9	1
SYN17R	113	108	113	84	15	20
SYN19R	18	18	18	17	1	0
SYN20R	57	57	57	47	3	0
Corpus	249	236	249	170	28	21

Detecting unnecessary assumptions. The detection of assumptions that are unnecessary for realizability has been suggested before. In [13], the authors define a notion of minimally sufficient sets of assumptions. They also define a notion of a helpful assumption, one whose addition or removal makes a difference in the realizability of some subset of guarantees. They evaluate the performance of computing a minimally sufficient set of assumptions on several `AMBA` and `GENBUF` specifications. Unlike our work, they compute redundant assumptions one by one (`LinMin`), not by a divide and conquer algorithm like `DDMin`. As we mention later in this section, in supporting materials [43] we compare `LinMin` and `DDMin` on our corpus and show that the latter is almost always significantly faster than the former. Moreover, the work in [13] does not examine the potential impact of removing redundant assumptions.

In [16], the authors define a notion of superfluous assumptions, ones whose removal does not change the realizability of the specification, does not add winning states to the system, and does not change the reactive distance from some position to some goal. They suggest to check for superfluous assumptions one by one. They do not report empirical results. Unlike our work, this work does not examine the prevalence of unnecessary assumptions, does not compute a realizable core, and does not examine the potential impact of removing them.

Assumptions that are unnecessary for realizability have also been considered in the context of repairing unrealizable specifications by assumption inference. Maoz et al. [30] compute a repair core, which is a locally minimal subset of the inferred repair assumptions that suffices for repair. Cavezza et al. [12] repair an unrealizable specification by adding a minimal set of assumptions that guarantee realizability. While the former applies the minimization after a candidate repair was computed, the latter applies it as part of the computation of a candidate repair. Unlike both of these works, our work takes as input an originally realizable specification, not an unrealizable one that requires repair. Moreover, we minimize the complete set of assumptions written by the specifier, not a subset consisting of inferred assumptions.

Minimization algorithms. Different set minimization algorithms have been considered in the literature, including delta debugging [47] (`DDMin`), `QuickXplain` [20], [34], which is an incremental recursive divide and conquer algorithm, and linear minimization (`LinMin`), which goes over elements of the input set one by one, and removes an element iff the criterion holds for the set without the element. All three algorithms find a core given a monotonic criterion. Our work does not present a new minimization algorithm. We chose to use `DDMin` with the performance heuristics described in [18], after we empirically compared the performance of these three algorithms, with the memoization described in [32], on our corpus. The comparison showed that `LinMin` is slower than all other algorithms on average on most categories. It also showed that `DDMin` is either competitive or significantly better compared to the others. The comparison also considers a domain-specific algorithm named `IncCore`, inspired by `QuickCore` [32]. The details of the comparison appear in [43].

Additional quality aspects. The existence of assumptions that are unnecessary for realizability, or ones that are not weakest [11], may be viewed as a form of inherent vacuity [19]. Yet, previous work on inherent vacuity in specifications for reactive synthesis [31] considered only the semantic equivalence case of inherent vacuity, not the weakening/strengthening case. Assumptions that are vacuous according to [31] are a special case of the ones we consider here. Thus we are able to detect more of them.

VIII. Conclusion and Future Work

We introduced the notion of core specifications, ones that include no assumptions that are unnecessary for realizability. We then showed how to compute an assumption core, a subset of assumptions that are necessary for realizability, and thus transform any specification into a corresponding core specification. Synthesis from core specifications results in more general controllers, i.e., ones that will satisfy their guarantees not only in all environments that satisfy the assumptions listed in the original specification, but in many more environments.

We implemented our work and evaluated it on benchmarks from the literature. The evaluation shows that almost all specifications in well-known benchmark specification sets include at least one unnecessary assumption, that in more than half of these at least half of the assumptions are unnecessary, and that in almost all cases the fully-automated removal of unnecessary assumptions pays off in total synthesis time and induces more general controllers whose reachable state space is larger but whose representation more memory efficient.

Our work has **important technical and methodological implications for the builders of synthesizers and for the engineers who use them**. First, synthesizer builders should consider automatic marking of unnecessary assumptions, as warnings, like the ones we have added to Spectra, so that engineers become aware of them and can decide whether to keep them, change them, or remove them. Second, while in realizability checking one should use the complete, original specification as written by the engineer, when synthesizing a controller for actual deployment in production, users should be advised to compute a realizable core and use the core specification for synthesis. Finally, engineers who write specifications for synthesis should be aware that the existence of unnecessary assumptions in their specifications, even if these indeed hold in the relevant environment, may negatively affect the performance of the synthesizer and the quality of the synthesized controllers.

Acknowledgements

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

References

[1] J. Alonso-Mora, J. A. DeCastro, V. Raman, D. Rus, and H. Kress-Gazit. Reactive mission and motion planning with deadlock resolution avoiding dynamic obstacles. *Auton. Robots*, 42(4):801–824, 2018.

[2] G. Amram, S. Maoz, and O. Pistiner. GR(1)*: GR(1) Specifications Extended with Existential Guarantees. In *Formal Methods - 23rd International Symposium, FM 2019, Proceedings*, Lecture Notes in Computer Science. Springer, 2019.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

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

[5] 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*, pages 34–50, 2014.

[6] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. pages 1188–1193. EDA Consortium, San Jose, CA, USA, 2007.

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

[8] 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.

[9] M. Brizzio, R. Degiovanni, M. Cordy, M. Papadakis, and N. Aguirre. Automated repair of unrealisable LTL specifications guided by model counting. *CoRR*, abs/2105.12595, 2021.

[10] D. G. Cavezza and D. Alrajeh. Interpolation-based GR(1) assumptions refinement. In *TACAS*, volume 10205 of *LNCS*, pages 281–297, 2017.

[11] D. G. Cavezza, D. Alrajeh, and A. György. A weakness measure for GR(1) formulae. In *FM*, volume 10951 of *LNCS*, pages 110–128. Springer, 2018.

[12] D. G. Cavezza, D. Alrajeh, and A. György. Minimal assumptions refinement for realizable specifications. In *FormaliSE@ICSE 2020: 8th Int. Conf. on Formal Methods in Software Engineering*, pages 66–76. ACM, 2020.

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

[14] 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.

[15] 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.

[16] R. Ehlers and V. Raman. Low-effort specification debugging and analysis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, pages 117–133, 2014.

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

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

[19] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In H. Chockler and A. J. Hu, editors, *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, volume 5394 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2008.

[20] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

[21] 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.

[22] 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.

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

[24] A. Kuvent, S. Maoz, and J. O. Ringert. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *ESEC/FSE*, pages 362–372, 2017.

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

[26] 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.

[27] S. Maoz and J. O. Ringert. On well-separation of GR(1) specifications. In *FSE*, pages 362–372. ACM, 2016.

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

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

[30] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for GR(1) specifications. In *ICSE*, pages 1016–1026, 2019.

[31] S. Maoz and R. Shalom. Inherent vacuity for GR(1) specifications. In *ESEC/FSE*, pages 99–110. ACM, 2020.

[32] S. Maoz and R. Shalom. Unrealizable cores for reactive systems specifications. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 25–36. IEEE, 2021.

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

[34] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, pages 592–607. Springer, 2013.

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

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

[37] N. Ozay, U. Topcu, R. M. Murray, and T. Wongpiromsarn. Distributed synthesis of control protocols for smart camera networks. In *2011 IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, Chicago, Illinois, USA, 12-14 April, 2011*, pages 45–54. IEEE Computer Society, 2011.

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

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

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

[41] L. Ryzhyk and A. Walker. 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.*, pages 84–99, 2016.

[42] R. Shalom and S. Maoz. Supporting artifact, 2023. https://doi.org/10.5281/zenodo.7528202.

[43] R. Shalom and S. Maoz. Supporting materials website, 2023. https://smlab.cs.tau.ac.il/syntech/asmcores/.

[44] Spectra. Spectra Website. https://smlab.cs.tau.ac.il/syntech/spectra/, –.

[45] A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[46] E. Wete, J. Greenyer, A. Wortmann, O. Flegel, and M. Klein. Monte carlo tree search and GR(1) synthesis for robot tasks planning in automotive production lines. In *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021*, pages 320–330. IEEE, 2021.

[47] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.