Validating the Correctness of Reactive Systems Specifications Through Systematic Exploration

Dor Ma'ayan
Tel Aviv University
Israel

Shahar Maoz
Tel Aviv University
Israel

Roey Rozi
Tel Aviv University
Israel

(A) (B)

Figure 1: Our combinatorial exploration methodology helped user study participants expose bugs in specifications such as a car driving in fog (A) or crossing a junction during red light (B). These bugs, and more, were not detected by participants before using combinatorial exploration.

ABSTRACT

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. While the synthesized system is guaranteed to be correct w.r.t. the specification, the specification itself may be incorrect w.r.t. the engineers' intention or w.r.t. the requirements or the environment in which the system should execute in. It thus requires validation.

Combinatorial coverage (CC) is a well-known coverage criterion. Its rationale and key for effectiveness is the empirical observation that in many cases, the presence of a defect depends on the interaction between a small number of features of the system at hand.

In this work we propose a validation approach for a reactive system specification, based on a systematic combinatorial exploration of the behaviors of a controller that was synthesized from it. Specifically, we present an algorithm to generate and execute a small scenario suite that covers all tuples of given variable value combinations over the reachable states of the controller.

We have implemented our work in the Spectra synthesis environment. We evaluated it over benchmarks from the literature using a mutation approach, specifically tailored for evaluating scenario suites of temporal specifications for reactive synthesis. The evaluation shows that for pairwise coverage, our CC algorithms are feasible and provide a 1.7 factor of improvement in mutation score compared to random scenario generation. We further report on a user study with students who have participated in a workshop class at our university and have used our tool to validate their specifications. The user study results demonstrate the potential effectiveness of our work in helping engineers detect real bugs in the specifications they write.

ACM Reference Format:
Dor Ma'ayan, Shahar Maoz, and Roey Rozi. 2022. Validating the Correctness of Reactive Systems Specifications Through Systematic Exploration. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552425

1 INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [43]. Rather than manually constructing an implementation of a reactive controller and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation is automatically obtained for a given specification, if such an implementation exists. As the correct-by-construction promise is attractive, much progress has been achieved over the last two decades on reactive synthesis algorithms, tools, and applications, e.g., [4, 8, 15, 33, 34, 37]. In light of these, and as reactive synthesis

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10.
https://doi.org/10.1145/3550355.3552425

makes its first steps in industry (e.g., [50]), an old, fundamental question, becomes unavoidable and urgent: how do we know that the specifications used as input for the synthesis are correct?

What does it mean for a specification to be "incorrect"? A specification may be incorrect w.r.t. the intentions of the engineer who wrote it. She intended to write a constraint that means something but as a result of a typo or an incorrect understanding of some language construct, wrote a constraint that means something else. A specification may also be incorrect w.r.t. the system's requirements and the environment in which it will operate. The engineer correctly wrote what she intended to write, but her knowledge of the requirements or the environment is wrong or limited.

Regardless of the type of incorrectness, an incorrect specification invalidates the entire reactive synthesis pipeline: it will result in no implementation (in case the specification is unrealizable) or, worse, in an implementation that the synthesizer considers to be correct, but should not be used. What is the value of a correct-by-construction implementation if one cannot trust the correctness of the specification it was synthesized from?

Since a specification is declarative and potentially complex, it is difficult to validate it directly. We thus set out to examine the behaviors it induces by exploring the behavior of a controller that was synthesized from it, using a set of scenarios, i.e., bounded example executions. Scenarios are tangible and simple; as they will execute, they will make behaviors actually occur and thus instantiate the defects and expose the gap, if any, between the engineers intention and knowledge on the one hand, and the specification as written and the requirements and real environment on the other hand.

As controllers are large, they enable a huge number of scenarios. Thus, given limited resources, which scenarios should we generate and execute in order to maximize our chances of exposing defects?

Combinatorial Coverage (CC) is a well-known coverage criterion. Its rationale and key for effectiveness is the empirical observation that in many cases, the presence of a defect depends on the interaction between a small number of features of the system at hand. For example, according to [31], all possible pairs of parameter values can typically detect 47% to 97% of the bugs in a system under test. The application of CC requires a model, consisting of a set of variables, their respective possible values, and constraints on the value combinations. CC is parameterized with a parameter t, specifying the size of the combinations that need to be covered. The common case in practice, where $t = 2$, is termed pairwise coverage. When a set of assignments covers all valid value combinations of every t variables, the set is said to achieve 100% *t-way interaction coverage*.

In this work we present an approach for validating the correctness of a reactive system specification via systematic scenario-based combinatorial exploration of the behaviors of a controller that was synthesized from it. Specifically, as the controllers we deal with have an exponentially large state space, a set of scenarios that will cover all states is infeasible to generate and to execute. Instead, we follow a CC approach and aim at generating a small set of scenarios, a scenario suite, that covers all tuples of variable value combinations of size t over the reachable states of the controller.

A scenario in our context means a sequence of valid environment inputs for the controller. At each state in the sequence, the scenario chooses the environment inputs and the controller responds by updating the system variables (outputs) values according to the controller's deterministic logic. At each state both the environment and the system variables are recorded for creating the CC. A complete scenario suite consists of a number of scenarios that together visit a set of states that provides coverage of all of the tuples.

Finding the smallest combinatorial covering scenario suite is NP-hard (already in its original setup), and thus, also in our context of controllers, heuristic algorithms are required. We present an algorithm that uses heuristics and a greedy approach. It adds states sequentially while maximizing the added coverage of each added state. It is symbolic (and implemented using BDDs), in order to scale to specifications over many variables.

An interesting aspect of our work relates to the combinatorial model. One of the challenges in applying CC in practice is the creation of the combinatorial model, a model which defines the variables, their domains, and the constraints on their allowed values. CC requires a combinatorial model as input, and in its common use, when the model in manually created, its definition is a costly and error prone process [48]. In our context, however, we use the reachable states and transitions of the synthesized controller as an implicit model. There is no need for manual model definition.

Note that the validation of the specification against the intended requirements, in contrast to verification of an implementation against the specification, cannot be fully automated. There is no way to automate an oracle because there is no ground-truth model available to validate the formal specification against. Our work automates the exploration of the behaviors that the specification induces such that many different behaviors are covered by a small set of scenarios. Beyond this automation, human involvement is required, to examine the generated scenarios as they are executed and judge whether they meet the intended requirements.

We implemented our ideas on top of Spectra [35], a specification language and reactive synthesis environment. We present two evaluations. First, a technical evaluation with existing benchmarks, where we use a mutation approach, specifically tailored for reactive systems temporal specifications. The results show that for pairwise coverage, our CC algorithms are feasible and provide a 1.7 factor of improvement in mutation score compared to random scenario generation. Second, a user study with students who have participated in a workshop class and have used our CC tool. The user study results demonstrate the potential effectiveness of our work to help engineers in detecting real bugs in specifications. Fig. 1 shows screenshots from simulations executed by participants of our user study, demonstrating bugs they were able to expose using our CC tool and did not find before using it. See Sect. 5.

To our knowledge, our work is the first to present, implement, and evaluate the validation of reactive systems specifications for synthesis through a systematic combinatorial exploration of synthesized correct-by-construction controllers. We discuss related work on CC and on dealing with problems in specifications in Sect. 6.

2 PRELIMINARIES

We recall necessary background and link it to our context.

Binary Decision Diagrams

Binary decision diagrams (BDDs) [12] are a compact data structure for representing and manipulating Boolean functions, traditionally used in formal verification [17]. A BDD is a rooted, directed,

acyclic graph, in which nonterminal nodes represent Boolean variables, outgoing edges represent values for these variables, and terminal nodes represent Boolean decisions. Given values for the Boolean inputs, the result of the function is achieved by traversing the graph accordingly until a terminal node is reached.

Using BDDs we can efficiently perform operations on Boolean functions. Standard operations such as conjunction, disjunction, and negation can all be computed efficiently. The "exists" and "forall" quantifiers (e.g., $g(x) = \exists y f(x, y)$) are also easy to compute. Counting the number of possible assignments is done in time polynomial to the size of the BDD and iterating over all assignments takes constant time per assignment. Finally, an extension to multi-valued variables exists, enabling us to use multi-valued variables and still compute the mentioned operations efficiently. We take advantage of all these in our algorithms.

Synthesized Controllers and their Symbolic Representation

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system directly from its temporal logic specification. A specification uses environment (input) and system (output) controlled variables. It consists of assumptions and guarantees about what should hold in all initial states, in all states and transitions, and infinitely often in every infinite run of the system. Synthesis outputs a controller, where each state is an assignment to all variables. In each step, the environment chooses the next assignment to the input variables, and the controller responds by choosing the next assignment to the output variables. By construction, all possible runs of the synthesized controller satisfy the specification, i.e., roughly, if a run satisfies all assumptions, then it also satisfies all the guarantees.

Importantly, the synthesized controller can be represented symbolically using BDDs. This is achieved by using an *initials* BDD that represents all valid initial states and a *transitions* BDD that represents the controller's response to each input of the environment. This symbolic representation of the controller is common to many synthesizers, including Spectra [35]. Execution of the controller is done by reading an environment input and then using the transitions BDD to choose the next assignment to the system variables. Furthermore, the symbolic representation enables the efficient performance of operations such as computing the intersection/union of sets of states; computing all immediate successors of a set of states; playing a reachability game from one set of states to another; choosing a single state from a given set of states (choosing a satisfying assignment); and calculating the number of states in a set of states (counting satisfying assignments). We use all these in our algorithms and thus the efficiency and effectiveness of our work rely on these properties of the symbolic representation.

Combinatorial Coverage

Combinatorial Coverage (CC) is a well-known coverage criteria, requiring that all valid value combinations of size t of the parameters of a system are covered. The challenge is to achieve CC using a small set of assignments. We adopt the formalization from [45].

Let $P = p_1, \ldots, p_n$ be an ordered set of parameters, $V = V_1, \ldots, V_n$ an ordered set of value sets, where V_i is the set of values for p_i, C a set of constraints over P, and $t \geq 1$ is an interaction level. We consider assignments to all parameters in P. A valid assignment

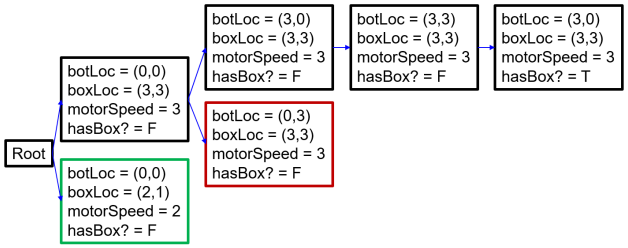

Figure 2: A (partial) scenario suite tree for the DeliveryBot specification. See Sect. 3.

is one that satisfies all constraints in C. The set of all valid assignments is denoted by $S(P, V, C)$. A subset of the valid assignments $S' \subseteq S$ covers all interactions of size t, iff every assignment of values to every subset of P of size t that occurs in some assignment in S, occurs in some assignment in S'. Many algorithms and tools exist for achieving CC using a small as possible set of assignments. Since in our setup, the synthesized controller is given symbolically, using BDDs, we are specifically interested in the symbolic BDD-based greedy heuristic algorithm from [45]. We use it as a starting point and extend it to the stateful context of our controllers.

3 OVERVIEW AND RUNNING EXAMPLE

Overview

In our work, we aim to validate a specification through a systematic combinatorial exploration of a controller that was synthesized from it. The synthesized controller is defined over environment and system variables, which together correspond to P in the traditional CC setting (see Sect. 2). Each variable has a respective value set in V, and the set of constraints C is implicitly defined by all the reachable states of the synthesized controller.

The key difference between the traditional CC problem and the one we face here, is that in addition to the constraints represented by C, i.e., the reachable states, we have another constraint on our ability to choose assignments to variables. Specifically, since the controller is stateful, one needs to create a sequence of valid inputs that together with the controller's reaction will achieve the desired coverage. Thus, formally, to the traditional setup described in Sect. 2, we implicitly add the following constraint: $\forall s \in S'$, either s is an initial state of the controller or $\exists s' \in S'$ s.t. s is a successor of s'.

Finally, we represent the resulting scenario suite as a tree. Each node in the tree represents an assignment to all variables, i.e., a state in a run of the controller, and each path from root to leaf in the tree represents a scenario, starting from an initial state and ending when reaching the leaf. We show an example of a partial scenario suite tree in Fig. 2. We elaborate on this example later in the paper.

Our CC algorithm guarantees that the resulting scenario suite is sound and complete: soundness means that each scenario is a valid run of the controller; completeness means that all the possible value combinations in the provided variable tuples that are reachable by the controller, will be reached by the scenario suite.

Specification and Scenario Suite Generation

We use a small running example in order to demonstrate our application of CC to reactive controllers and provide a rough overview of our algorithm's input and output. A formal presentation appears in Sect. 4. The example is representative of some of the kinds of applications of synthesis, see, e.g., [28, 40], but is much smaller than

the specifications we use in our evaluation (≈ 8000 reachable states as opposed to 10^6 up to 10^{16} reachable states).

Listing 1 presents part of a specification of a simple delivery robot, written in the Spectra language [35]. It describes a delivery robot moving on a 2-dimensional grid. Its job is to pick up boxes provided by the environment from anywhere on the grid and to bring them to (0, 0). In addition, the robot has a variable `motorSpeed`, indicating how many steps in the grid it may advance in each turn. In total, the specification has six variables: `boxLocX`, `boxLocY`, `motorSpeed`, `botLocX`, `botLocY`, and `hasBox`.

Is this specification correct? Perhaps the engineer who wrote it had a typo in one of the assumptions or guarantees? Perhaps she missed a critical guarantee? To help validate the correctness of the specification against the intended requirements, our tool provides an automatic and systematic means to efficiently explore the possible behaviors it induces. We use the Spectra synthesizer to synthesize a controller and run the CC algorithm with a set T containing all tuples of variables as input.

In our example, we chose $t = 2$ and thus the size $|T|$ would be 15, as there are 6 variables, resulting in 15 unique pairs between them. Notice that T contains tuples of variables and not tuples of values. The algorithm will cover all the valid (reachable) combinations of value assignments for each pair. So in this case, even though there are only 15 pairs of variables, the algorithm needs to cover a total of 342 different pairs of values, which are reachable in the synthesized controller. Of course, these pairs can be covered in a scenario suite that includes less than 342 states, because each state covers many variable value pairs.

For this input of a controller and a set of variable tuples to cover, our algorithm outputs a scenario suite in the form of a tree. Each node in the tree represents a state of the controller. Each path from root to leaf in the tree represents a scenario.

Example 3.1. Figure 2 shows an example tree of a (partial) scenario suite, covering 43 pairs of value combinations with 6 states in 3 scenarios. The top scenario (black) takes the bot to the box in (3, 3), advancing first in the X axis. The second (red) advances the bot in the Y axis first. The third scenario (green) advances the bot at a different motorSpeed to a different box location. The second scenario only adds a single pair of new values to the coverage ($botLocX = 0, botLocY = 3$). The third covers 12 new pairs.

```
1  spec DeliveryBot
2
3  // The location of the next box to pick up
4  env Int(0..5) boxLocX; env Int(0..5) boxLocY;
5
6  // The speed of the deliveryBot
7  sys Int(1..3) motorSpeed;
8  // The deliveryBot's location
9  sys Int(0..5) botLocX; sys Int(0..5) botLocY;
10 // Is the deliveryBot carrying a box
11 sys boolean hasBox;
12
13 // assumptions and guarantees
14 ...
15 // Guarantee: The box can only be dropped at (0,0)
16 gar G ((botLocX != 0 | botLocY != 0) & hasBox) ->
17       next(hasBox);
18 ...
```

Listing 1: Example DeliveryBot specification (excerpt). The complete specification is available in [2].

Importantly, the engineer could then execute all the scenarios in the scenario suite, and it is guaranteed that each pair of values from T will be covered by at least one state in at least one scenario.

In this small example, for pairwise coverage of all variable values, our algorithm outputs a suite of 21 scenarios and a total of 46 states.

Note that the scenario suites we generate for our DeliveryBot include no more than 50 states, which are enough to provide pairwise coverage of the more than 8000 reachable states of the controller. But do they suffice for detecting defects in the specification?

Detecting Specification Defects

As mentioned in the introduction, we aim to help engineers in detecting two general kinds of specification defects. The first happens when **the specification might not correctly express the intentions of the engineer who wrote it.** Indeed, while writing our example specification, we accidentally used & instead of | in the safety guarantee in line 16. This resulted in a controller that may drop a box in any cell in column 0 or in row 0, and not necessarily only in (0, 0). Thanks to the use of CC, this issue was quickly found: as we executed the scenario suite, we observed that in one of the scenarios the box was dropped at an incorrect spot, i.e., not in (0, 0) as intended. Importantly, pairwise coverage guaranteed that this wrong behavior will manifest in at least one of the scenarios.

The second type of specification defect happens when **the engineer may not know or understand the real environment in which the system would run.** Indeed, in our DeliveryBot example, there was a physical limitation that we were not aware of: if the robot carries a box at motor speed 3, the box falls off. Again, even with only pairwise coverage, our algorithm guarantees that the scenario suite it generates includes a state in which the robot holds a box and is moving at motor speed 3, because this is a combination of values from only two variables, `motorSpeed` and `hasBox`. Thus, when executing the scenario suite, this wrong behavior is indeed manifested.

Example 3.2. In Fig. 2, the first scenario (black) is 4 states long. Its last state drives the bot at motorSpeed 3 while holding a box. It will make the box fall and thus manifest the defect.

Overall, our running example demonstrates how the scenario suite generated with our approach helped find specification defects of the two kinds we consider. The complete specification and example executions (simulation videos) showing how these two defects were found are available in [2].

4 A CC ALGORITHM FOR SYMBOLIC CONTROLLERS

Main Technical Contribution

Our algorithm is an iterative greedy algorithm. At each step it chooses a new assignment to environment inputs in order to cover as many new valid tuple combinations. This is achieved by maintaining BDDs of all of the uncovered reachable states. At each step the algorithm starts with all possible successors in the symbolic controller and reduces the possibilities until one variable assignment is

chosen. Once a full coverage is reached, which is signalled when all the uncovered BDDs are false, the algorithm returns the scenario suite, represented as a tree of states, each of which is an assignment to all variables, environment inputs and system outputs.

Algorithm Inputs

The algorithm receives as input the symbolic controller SC for which to generate the scenario suite, and a set, T, of tuples of variables from the symbolic controller to cover.

Example 4.1. For the delivery robot in Lst. 1, the requirement to cover all pairs for all variables would result in $T = \{\langle \texttt{boxLocX}, \texttt{boxLocY}\rangle, \langle \texttt{boxLocX}, \texttt{motorSpeed}\rangle, \ldots, \langle \texttt{botLocY}, \texttt{hasBox}\rangle\}$, consisting of all the pairs of the six variables, totalling $|T| = 15$.

The symbolic controller, SC, is required to support the following functions: $getReachable(SC)$, which returns a BDD containing all the reachable states of the controller, and $getSuccessors(currentStates, SC)$, which receives as input a BDD, $currentStates$, representing a set of states, and returns a BDD representing a set containing all of the successors of $currentStates$.

For simplicity, we first present Alg. 1 without the lines marked in blue (without lines 9-14, 23-26, 28). We explain the blue lines at the end of this subsection.

Initialization

Throughout its execution, the algorithm maintains for each $t \in T$ a BDD $uncov(t)$, which represents all the reachable but yet uncovered states using the variables in the tuple t. We first compute all reachable states of the symbolic controller (line 1). Then, $uncov(t)$ is initialized in line 3. $Proj_t(bdd)$ is a function that projects a BDD on a set of variables t. We also initialize $chosenTree$, which will contain the final scenario suite (line 5). As explained previously, the resulting scenario suite will be represented as a tree of states, in which each path from the root of the tree to a leaf is a scenario.

The Main Loop

The algorithm includes a main loop, from line 7 to line 35. Each iteration of the main loop adds one node, $chosen$, to $chosenTree$ (i.e., adds one state to the scenario suite).

It starts with obtaining the union of all possible successors of all previously chosen states, as shown in line 8. The algorithm then iterates over all the uncovered BDDs in decreasing order of the satisfy count of each BDD (line 17). If $uncov(t) \cap collected \neq \emptyset$ we can remove all successors that do not cover any assignments of the current tuple t. This simple greedy heuristics allows us to quickly focus on useful possible successors, i.e., ones that will add to the required coverage. Iterating over the BDDs in decreasing order of satisfy count is a heuristic to help choosing an assignment that satisfies more combinations [45].

Finally, we choose one of the remaining successors at random (line 22), add it to the results tree (line 27), and update $uncov(T)$ in order to remove all variable combinations covered with our current choice (line 30). We also remove tuples that are already fully covered, as shown in line 32, as they are no longer needed.

Once all tuples are covered (as guaranteed when using the blue lines, see below), the algorithm returns the chosen states as a tree, where each child node represents a successor of its parent. The returned tree represents the scenario suite.

Algorithm 1 Compute Scenario Suite

input: The symbolic controller SC to cover, and coverage requirements, given as a set T of tuples of variables to cover.

```
        /* Initialization */
1:  reachable ← getReachable(SC)
2:  for t ∈ T do
3:      uncov(t) ← Proj_t(reachable)
4:  end for
5:  chosenTree ← ∅
6:  chosen ← root
        /* Main Loop */
7:  while T ≠ ∅ do
8:      successors ← getSuccessors(orAll(chosen), SC)
9:      allUncov ← orAll(uncov)
10:     reachLayers ← ∅
11:     if successors ∩ allUncov = ∅ then
12:         reachLayers ←
    playReachability(orAll(chosenTree), allUncov, SC)
13:         successors ← lastLayer(reachLayers)
14:     end if
15:     collected ← successors
16:     Sort T in decreasing order of satCount(uncov(t))
17:     for t ∈ T do
18:         if uncov(t) ∩ collected ≠ ∅ then
19:             collected ← uncov(t) ∩ collected
20:         end if
21:     end for
22:     chosen ← satOne(collected)
23:     if reachLayers ≠ ∅ then
24:         path ← findPath(reachLayers, chosen)
25:         chosenTree ← append(chosenTree, path)
26:     else
27:         chosenTree ← append(chosenTree, chosen)
28:     end if
29:     for t ∈ T do
30:         uncov(t) ← uncov(t) ∩ ¬chosen
31:         if uncov(t) = ∅ then
32:             T ← T \ t
33:         end if
34:     end for
35: end while
36: return chosenTree
```

Example 4.2. Consider the first scenario (black) in Fig. 2. In the first iteration, the algorithm looked at all possible initial positions and chose the best one (based on the heuristics in line 16). In the second iteration, the algorithm looked at all possible initial positions and all successors of the previously chosen state and chose the best variable assignment to add to the tree, based on the same heuristics. In the third iteration, the algorithm chose a successor to the state chosen in the first iteration, thus creating a new branch in the tree (red). In the forth iteration, the algorithm chose a new initial position, thus creating a new branch in the tree (green). We see how in each iteration, all possible choices that can be appended to the state tree are considered and the best one is chosen greedily.

Adding Reachability Games

One problem with Alg. 1 as presented without the blue lines, is that when the algorithm reaches a state that has no useful successors (i.e., no successor covers a tuple that was not yet covered), it aimlessly adds more states that do not improve the coverage. We mitigate this problem by adding reachability games.

We shall now explain the blue lines in Alg. 1. Lines 9-14 add a reachability game that occurs whenever the algorithm reaches a point in which all possible successors do not improve the coverage of the scenario suite. The check occurs in line 11. The reachability game is played from the set of all previously chosen states (including the root position) to the set of all of the uncovered states (line 12). The set of all previously chosen states is $orAll(chosenTree)$, which returns the union of all BDDs stored in $chosenTree$. We define $playReachability(from, to, SC)$ as a function that plays a reachability game and returns a list of BDDs $L_1, L_2, \ldots, L_k$ called layers. The returned layers satisfy $L_1 \subseteq from$, $L_k \subseteq to$ and $\forall i \geq 1\ L_i \subseteq successors(L_{i-1})$. The last layer will serve as the defacto set of successors for the current iteration.

We reach line 24 when a reachability game was played, after a successor is chosen. Instead of just appending the chosen state, we choose a path containing a single variable assignment at each step from the original successors to the effective successors, and appended to the scenario suite tree. The path is appended to the node whose successor is the first state in the chosen path (line 25).

Soundness, Completeness, and Complexity

Algorithm 1 is sound, i.e., it stops and returns a valid tree of states. It is also complete w.r.t. the reachable states of the controller, i.e., the returned tree provides a full coverage of the variables in the given set of tuples T that are reachable.

Theorem 4.3 (Alg. 1 is Sound and Complete). *For every symbolic controller SC, and every set of variable tuples to cover T, Alg. 1 stops and returns a valid tree of environment inputs CT, which covers all reachable tuples in T. Formally:*

$$\forall c \in CT : getChildNodes(c) \subseteq getSuccessors(c, SC)$$
$$\forall p \in getReachable(SC) \cap T : \ \exists C \subseteq CT \ s.t. \ p \subseteq C$$

We consider time complexity per node in the returned $chosenTree$:

Theorem 4.4 (Alg. 1 Time Complexity). *The time complexity of Alg. 1 is $O(m|T|log|T|)$ BDD operations, where m is the number of states in the generated scenario suite, i.e., $m = |chosenTree|$.*

The proofs appear in supporting materials [2].

5 IMPLEMENTATION AND EVALUATION

Implementation We implemented the CC algorithm on top of Spectra [1, 35], using BDDs [12] via CUDD 3.0 [46], which supports multi-valued variables. We maintain the uncovered combinations of T (see Sect. 4) in a sorted set of BDDs. Each BDD represents the uncovered combinations of one tuple of variables. The sorting of the set according to *satCount* (Alg. 1 line 16) is done only when a BDD is changed, and only for that BDD, resulting in less comparison operations than expected with a naive approach. In addition, the implementation maintains the current number of combinations left to cover, enabling us to stop the algorithm based on the coverage percent achieved so far, e.g., the engineer may request that the algorithm stops at 95% coverage.

The implementation is integrated as add-on to the Spectra IDE, where the engineer can write a specification, synthesize a controller, generate a scenario suite, and execute it, all within Eclipse.

Our evaluation consists of two parts. First, a technical evaluation that focuses on a comparison of our CC approach to random scenario generation over relevant criteria. Second, a user study that focuses on evidence for the effectiveness of our work in helping engineers expose bugs in the specifications they write.

5.1 Technical Evaluation

All specifications used in our evaluation, the raw data we collected, and means to reproduce our experiments, are available in [2]. We consider the following research questions:

- RQ1 How does our CC algorithm compare to random generation w.r.t. scenario suite size and achieving partial coverage?
- RQ2 How does our CC algorithm compare to random generation w.r.t. the potential of exposing specification defects?
- RQ3 How well does our CC algorithm scale with larger specifications and with more tuples to cover?

Corpus of Specifications

We use two sets of benchmarks from the literature. First, specifications from the SYNTECH benchmarks [22], available from the Spectra website. These specifications were written by 3rd year CS undergrads in class projects taught by the authors of [22]. The benchmarks have already been used in the literature [15, 36, 40]. From these benchmarks, we selected the largest ones that we were able to synthesize a controller from within a timeout of 5 hours. Thus, we experimented with four specifications. Tbl. 1 shows the number of multi-valued variables (as they appear in the specification), Boolean variables (after encoding), assumptions, guarantees, and reachable states for each of these.

Second, we use IBM's GenBuf [9], which has been extensively used in the GR(1) literature for evaluation [14, 16, 22, 27]. GenBuf is parametric and therefore suitable for examining scalability.

Validation

We implemented a test that iterates over a given scenario suite tree and makes sure that it achieves a full coverage, independent of the implementation that generated the scenario suite. We ran this test over the scenario suites created by our algorithm on all the specifications in our corpus. In addition, we have created a number of synthetic specifications and manually checked that the generated scenario suite is correct. This validation increases our confidence in the correctness of our implementation.

Experiment Setup

Our setup uses Spectra [35] for the synthesis of the symbolic controllers. We then apply our CC algorithm on the symbolic controller, computing the scenario suite based on the required configuration. In order to account for nondeterminism in the BDD library, we ran each experiment 10 times and reported averages.

Random Scenario Generation Algorithm

For the comparison with random generation, we define the random algorithm as follows. In each iteration, the algorithm picks a

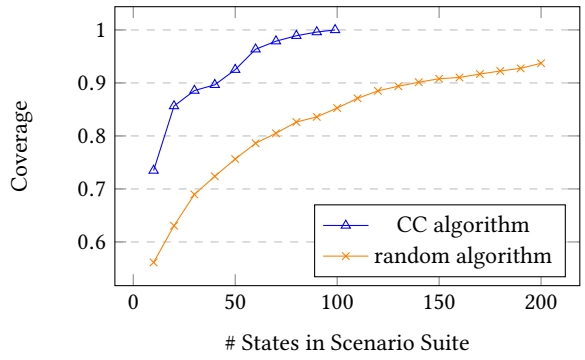

Figure 3: Comparison of size and coverage between CC algorithm and random generation using GenBuf10 with pairwise coverage.

random unvisited state that is a successor of one of the previously chosen states, and adds it to the scenario suite. This random algorithm is better than a naive random algorithm, as it never picks a previously visited state. Most importantly, its choices never violate the assumptions and guarantees in the specification. Finally, for each specification, we limit the random algorithm executions to twice the size of the scenario suite produced by our algorithms. This way we can compare the rates at which our algorithm and the corresponding random approach achieve the required coverage. We ran the random algorithm 10 times and reported average results.

Results for RQ1: Size and Coverage against Random

Figure 3 shows a comparison of size and coverage between our CC algorithm and the random algorithm we described above. We show the results for a representative specification; the results are very similar for all specifications in our corpus. Complete results are available in [2].

To answer RQ1: Our algorithm achieves full coverage with a rather low number of states. The random algorithm doesn't achieve full coverage even within twice the size of the scenario suite for which our algorithm already achieved full coverage. Our algorithm achieves most of the coverage during the first iterations; the last iterations are used to achieve the last few reachable combinations. If the cost of using a large suite is too high, one can use a smaller suite and still enjoy a high coverage.

Mutation Testing and Results for RQ2

Mutation testing [5, 25] is a well-known approach to evaluate the quality of a test suite. To estimate the effectiveness of our CC algorithm at exposing defects in real-world specifications, we use a novel mutation approach for specifications, based on a mutation operator that removes a single safety guarantee. This operator simulates cases where the specification is missing a guarantee.

Example 5.1. An example for a defect that is the result of a missing safety guarantee is the second defect presented in Sect. 3. The defect is that whenever the bot holds a box and is at motor speed 3 it drops the box. Its root cause is a missing safety guarantee: $G\,!(motorSpeed == 3\ \&\ hasBox)$.

Thus, for each specification in our corpus, we iterate over each non-redundant safety guarantee[1] and synthesize a symbolic controller without it. We use our algorithm to generate a scenario suite and then generate a scenario suite of the same size using the random algorithm. Finally, we examine both scenario suites, to see if they contain a state that violates the "left-out" safety guarantee, thus "killing the mutant" and finding the defect.

Note that in our setup of specifications, one cannot use existing well-known mutation operators that apply to code, as these may create unrealizable specifications or specifications with stronger guarantees, both of which will not be useful in our case. We have to be sure that the mutated specifications have weaker guarantees.

The rightmost columns of Tbl. 1 present the average mutation score for the different specifications. We see that in almost all cases, the mutation score of our CC algorithm is better compared to the random algorithm's generated scenario suite of the same size.

To answer RQ2: We observe an average improvement by a factor of 1.7 in mutation score for our CC algorithm compared to the random algorithm. This evidence suggests that our algorithm has higher potential for exposing specification defects.

Results for RQ3: Performance and Scalability

Table 1 presents the results of our experiments on different specifications with pairwise coverage.[2] For each specification we show the number of multi-valued variables as they appear in the specification (V), Boolean variables (BV), assumptions (A), guarantees (G) and the number of reachable states in the controller, the number of variable tuples we need to cover $|T|$, the total number of different reachable combinations for each tuple in T, the number of states in the scenario suite tree (nodes), the number of scenarios in the scenario suite tree (leafs), and the running time of the algorithm.

We analyze scalability in terms of the number of variables in the specification, which affects both the number of reachable states and the number of tuples to cover, as shown in Tbl. 1.

To answer RQ3: Overall, the algorithm scales well when increasing the specification size. The scenario suite size grows much slower than the size of T and the size of the specification. In terms of running time, the algorithm does run slower when scaling up.

Threats to the Validity of the Results

First, symbolic computations are not trivial and our implementation may have bugs. To mitigate this, we performed a thorough validation, see above. Second, even though the algorithm we deal with is deterministic, JVM and CUDD garbage collection add variance to running times. We therefore repeated each experiment 10 times and report average values. CUDD dynamic reordering may result in additional variance, so we used CUDD's default dynamic reordering, as common in related work, e.g., [10, 36]. Third, our algorithm may achieve different results based on the synthesized controller. To alleviate this, we used a variety of different specifications, varied in number of variables and reachable states. Fourth,

[1] According to [37], many specifications include vacuous guarantees. By definition, a mutation that removes a vacuous guarantee is useless.

[2] Pairwise, i.e., $t = 2$, is the most common in the CC literature. In supporting materials we include results for $t = 3$ as well. Other values are not common in the literature.

Table 1: Performance, generated scenario suites sizes, and mutation scores

	Spec & Controller Info					Tuples Info		Generated Scenario Suite			Mutation Score	
Specification	#V	#BV	#A	#G	# Reach. States	# Tuples	# Tuple Comb.	#States	#Scenarios	Time (sec)	CC	Rand
AirportShuttle (t=2)	36	38	14	21	3.96E+09	630	2,672	93	27	6.09	**92.7%**	77.3%
Junction2 (t=2)	36	44	27	51	7.93E+13	630	3,087	33	17	44.08	**71.4%**	21.4%
RoboticArm (t=2)	21	52	17	28	3.93E+06	210	2,961	110	67	97.19	86.2%	**92.4%**
SimpleVehicle (t=2)	34	53	32	49	1.77E+16	561	4,283	136	73	1205.20	**84.5%**	52.5%
GenBuf5 (t=2)	24	24	28	81	2.31E+05	276	1,085	36	12	0.65	**83.1%**	53.1%
GenBuf10 (t=2)	35	35	43	152	4.20E+07	595	2,326	100	41	4.48	**89.0%**	39.7%
GenBuf20 (t=2)	56	56	73	368	2.80E+11	1,540	5,961	172	90	59.55	**92.8%**	17.0%

we do not know if the mutation operator we use is representative of mistakes engineers would make in practice. Our operator is reasonable but rather coarse. In the future, we may try using human-like mistakes or refined LTL mutations. Finally, our corpus is limited to a small set of benchmarks. It is thus important to note that all publicly available specifications for reactive synthesis were published in final form only, with no documentation about earlier versions and the defects they may had (the SYNTECH benchmarks include multiple versions, but no defects documentation). Therefore, we are unable to report on our success in detecting real defects in these specifications. Instead, we evaluated the effectiveness of our work over existing benchmarks using a mutation approach. We complement this evaluation with the user study we present below.

5.2 User Study

To better evaluate the applicability of our approach in practice and to further understand its ability to help engineers expose real-world bugs, we conducted a user study where we provided the CC scenario generation tool to students and asked them to use it in order to try to find bugs in a specification they wrote. We ask the following research questions:

RQ1 How may engineers validate their specifications for synthesis without the CC scenario generation tool?

RQ2 Can engineers use the CC tool to find bugs in their specifications and detect bugs that were not found beforehand?

User Study Setup

We conducted the study on 10 pairs of students who participated in a semester-long workshop class (20 students in total). All participants were senior undergraduate students with a strong background in programming and computer science: they had already completed classes with Python, Java, and C projects, and took courses in data structures and algorithms. We chose to conduct our study on students since reactive synthesis is not widely used in the industry and since some previous literature shows that in many cases, students are not a significant threat in empirical software engineering studies [26]. Moreover, more than 50% of the participants already have student positions in the industry as software engineers. As compensation, and with approval of the university's IRB committee, we offered four bonus points for the workshop's final grade for those agreeing to participate in our study.

The user study was done as part of the last workshop assignment; at this stage, students already had experience with Spectra since they had already wrote several specifications, e.g., for a cleaner robot, synthesized controllers, and simulated their executions.

The user study consisted of two parts. In the first part, we gave the students a task to write a specification describing the behavior of a junction with cars, pedestrians, and traffic lights. The environment controls the inputs, which are sensors that report on coming cars and pedestrians who ask to cross (as well as on the existence of fog or a police request to manually control the traffic lights behavior etc.). The system controls the outputs, which set the colors of the different traffic lights (red or green). We provided the students with abstract guidelines on how the junction should work (e.g., the junction should be safe and not allow accidents, every pedestrian that wants to cross will eventually be able to cross, emergency vehicles have priority over ordinary vehicles etc.); the guidelines left a space for flexibility in the way the specification is written. As a baseline, however, we asked participants to write a correct specification that is also as optimal as possible (for example, a simple traffic light system that changes its lights in a cyclic order while ignoring environment input was forbidden).

We also provided the students a Java-based GUI skeleton of the junction to easily run their synthesized controller and observe its behavior. This GUI didn't mock or simulate specific behaviors, but just provided skeleton code to display the junction's current state graphically; participants were free to choose whether and how to use this GUI to validate the correctness of their specifications.

Participants had three weeks to complete the assignment. After three weeks, they submitted their correct implementation and a document that explained their design decisions and their interpretation of the more abstract parts of the assignment. In addition, we asked students to explain in this document what actions they took to validate the correctness of the submitted specification and of previous specifications they wrote as part of the workshop. Notice that participants were unaware of the CC feature at this stage, and they submitted a specification that they considered correct and was ready for our review and grading.

After collecting all data from the first part of the study, we continued to the second part and presented the CC tool in one of the workshop's class meetings. Then, we asked participants to validate their specifications again, this time, with the CC tool they saw in class. We instructed the students to report all bugs and unwanted behaviors they detected at this stage. All the communication with participants and the documentation was done using Slack.

Results for RQ1: Means used by participants to validate their specifications before we introduced them to our CC tool

To answer RQ1, we conducted a thematic analysis [11] to identify common validation strategies. From the documents and comments we received from participants, we can see that they invested significant time and thought into validating the correctness of their specifications, using various validation strategies.

The most common validation strategy mentioned by all the participants was implementing a random simulation that mocks up the environment according to their understanding of the problem.

Note that implementing an effective random simulation of the environment is not trivial, because it has to satisfy the assumptions written in the specification. When the environment violates an assumption, the Spectra execution engine outputs a warning to the console and from then on, nothing is guaranteed about the behavior of the controller. With our CC approach, the generated scenario suite is guaranteed to never violate the environment assumptions from the specification.

After implementing the random simulation, teams used the synthesized controller and observed its behavior against the random environment they created. Then, when they observed an unexpected behavior, they tried to print the current state of the system inside the simulation in order to debug the problem (we use Tx to denote that team-x mentioned it):

> "*When we saw in the simulation that something isn't behaving as we expected it to, we added system variables in order to "monitor" certain parts of the specification. In one of the times we did this, we added a system variable that is always equal to one of the counters so we can print it out every turn from the java code and check if it resets when it's supposed to while looking at the GUI interface.*" **(T6)**

However, many participants understood that such validation is insufficient since some behaviors and edge cases are rare and are not easily reproducible by simply observing a random simulation (T4,T5,T6,T9,T10):

> "*... we used mainly the GUI. Each and every time, we were looking for the specific scenario we wanted to check to happen, and then once it happened, we checked whether it worked as we expected it to work or not. One of the main difficulties in this method was that some problems ... only occurred once in many runs - that made us believe that things are working properly and we proceed with them, and only, later on, we found out that they are wrong.*" **(T9)**

To deal with this problem, the teams used additional strategies. T10 tweaked the environment's randomness in such a way that it will make otherwise rare events more frequent. T4 and T6 wrote specific scenarios, i.e., implementing a concrete behavior of the environment simulation, which mocks up edge cases:

> "*... when we wished to test something specific, we wrote specific java code in order to make it happen, for example, on the cleaner task, we wrote specific code to make sure the robot would pass in the orange zone*" **(T4)**

Another team added features incrementally and ran the simulation after each feature they added to inspect it's correctness:

> "*... after every feature we added (freeze mode, road constructions, fog), we tested it again while observing the simulation ...*" **(T5)**

Results for RQ2: CC helped participants expose bugs that they did not find without it

While using CC, i.e., in the second part of the user study, 9 out of 10 teams found bugs that they did not found before using the methods described in the answer for RQ1. We received good feedback from all teams on the usability of the tool:

> " *It was nice to see how the tool works, and actually prints a snapshot of all the variables in each state. This is really helpful and is exactly what we wished for when we were working on the Junction project at the first place. We know for sure it would have saved us lots of time.*" **(T2)**

As one detailed representative example, we consider T4. Before using CC, T4 validated their junction specification with many random and weighted random inputs and even wrote custom Java code to describe different scenarios they wanted to validate. Their specification is also quite complex, it has 55 variables, it spans over 432 lines, 51 guarantees, 36 assumptions, and it uses several advanced Spectra language features such as counters [35].

After running CC, T4 reported on two bugs they did not find beforehand. For the first bug they found, one of the scenarios generated by the CC algorithm showed a car turning west, even though pedestrians were crossing at the time, as seen in Fig. 1 (B). Here is how T4 describe the reason for the bug:

> "*The bug was we had confused the left turning south vehicles lane environment variable with the right turning south vehicles lane environment variable.*" **(T4)**

Then, T4 describe how quickly they found the bug using CC:

> "*When working with the previous simulator, before the testing tools, our main way of convincing ourselves that the spec is right, was to let sim run for some time, look at the GUI, and see that everything looks as we think it should look. Because sim is constantly randomly adding vehicles to the screen, and because there are many 'objects' in this environment that we need to keep track of (a lot of vars), and therefore many different combinations of vars ... we let this basic mistake remain present in our project without us noticing it until now. CC made us realize it very very quickly, and this, we think, shows the qualitative difference between 'playing' by ourselves with the simulation to find bugs in GUI against an automatic tool that systematically creates for us tests to find bugs ...*" **(T4)**

This bug is an example of the first type of bug, in which the specification does not express the intention of the engineer who wrote it. In a similar bug that was found using CC, the simulator showed a non-emergency car given green light and crossing while there was fog, which is a behavior that is not allowed according to the assignment's instructions. The GUI's state can be seen in Fig. 1 (A). This bug was also a bug of the first type and was a result of a mistake in the specification.

Note that when using CC, different teams found different bugs related to different aspects of the specification. For example, T5 detected weird behaviors related to their initial assumptions:

> "*After running the test suite, we observed some unexpected behavior. ... whenever a new test case started ... there were strange behaviors such as a car running over a road construction or two*

> *cars colliding. After investigating, we also noticed that on each of this test states freeze mode was on. This made us realize that we didn't handle correctly the scenario where freeze mode is on in the initial state … if in the initial state freeze mode is on, then a collision can occur. After adding the following assumption, the unexpected behaviors didn't occur anymore*" **(T5)**

In our user study setup, where the synthesized system was executed against a Java simulation written by participants themselves and not against a real environment, it was not possible to find bugs of the second type (where the engineer does not fully understand the requirements or the real environment in which the system will run). However, many teams found undesired behaviors in their Java simulator during the execution of the CC generated scenarios, which may be viewed as bugs of the second type.

Threats to Validity

To ensure **external validity**, we chose a **target population** that represents future potential tool users. Although participants used the tool only for one task, we believe the result could be generalized since it is a rather complex task in our domain. Furthermore, since the task's requirements had a place for interpretation, participants found different bugs in their specifications using the tool, which relate to their different design choices. Our study spanned a semester-long workshop class but included only 20 students. In the future it would be beneficial to include more participants and thus improve the expected generalizability of the results. To safeguard **internal validity** and make sure participants developed correct implementations, we put a special emphasis on correctness in this particular task, and made certain participants would not be aware of the new tool capabilities until we introduced it.

6 RELATED WORK AND DISCUSSION

Combinatorial Coverage Combinatorial coverage is a well-known coverage criterion, applied mainly in the context of functional testing [13, 20, 21, 24, 29, 51, 52]. Its rationale is supported by much empirical evidence [6, 30, 31, 42, 44, 49, 53, 55]. Different algorithms to achieve CC have been suggested, e.g., [18, 19, 32, 54], and it is supported by several academic and commercial tools.

As a starting point for our algorithm we chose the algorithm of [45], whose use of BDDs makes it a good fit to deal with the symbolic representation of the synthesized controllers we consider. However, we altered the construction from [45] to handle the stateful nature of the controller. Note that existing algorithms, including the one of [45], are unsuitable for the stateful nature of the problem we have. In our setup, each generated scenario is not one state but a sequence of states, and the constraints change after the addition of every state, based on its set of valid successors (described symbolically). Moreover, to be efficient and complete, we have to use reachability games as part of the algorithm (see Alg. 1 blue parts).

Some previous works apply CC to stateful scenario suites [3, 41]. They apply CC to sequences of events, and are limited to event-based systems, defined over a set of events alphabet. In contrast, our work applies to a different computation model of a reactive controller, defined over a large set of input and output variables.

Finally, fundamentally, unlike all works cited above, we apply CC to a system in order to find defects in the specification it was synthesized from, not to test its correctness w.r.t. a specification.

Validating the Correctness of Specifications for Synthesis Some works deal with problems in specifications for synthesis, e.g., w.r.t. unrealizability [14, 27, 36, 38]. Yet, no focus has been given to the specification's correctness w.r.t. the engineer's intention and knowledge of the requirements and environment in which the system would run. To our knowledge, our work is the first to attempt at validating correctness in this regard. Note that our work is incomparable to the works that deal with unrealizability. For unrealizable specifications no implementation exists, so there are no controllers to explore. Our work applies to realizable specifications. These may still be incorrect, which is where CC exploration enters.

7 CONCLUSION

We presented an approach for validating the correctness of a reactive system specification for synthesis, via systematic scenario-based combinatorial exploration of the behaviors of a controller that was synthesized from it. We presented an algorithm designed to achieve a small scenario suite by using heuristics and a greedy approach. We used benchmark specifications and mutations to evaluate our algorithm against random scenario generation. We used a user study with students who used our work to validate their specifications in order to demonstrate its potential effectiveness in detecting real bugs in specifications for synthesis.

Validation by exploration requires the scenarios to be executed and monitored for unexpected behavior, which could be a human intensive task. This further motivates us to develop an approach that minimizes the number of scenarios and states to be executed.

For performance, we take advantage of the symbolic representation of the synthesized controller and define and implement our CC algorithm symbolically. Many synthesizers use a symbolic representation for the controllers they output. In this sense, while we implemented our work on top of Spectra, taking advantage of its synthesizer, its generic execution mechanism for controllers, and its benchmark specifications, our algorithm is not limited to Spectra.

We consider directions for future work. First, the application of other approaches to validating the correctness of specifications for synthesis. For example, complementing the CC, global exploration approach, with a local, property-based exploration approach, where the input for scenario generation are regular expressions that a generated scenario has to satisfy. Second, we consider future work on improving the quality of the application of mutations in our context, e.g., altering guarantees with human-like mistakes or using refined LTL mutations [7, 23, 47]. Third, to address scalability, we consider means to replace our use of the symbolic controller with Spectra's just-in-time controller [39] .

Acknowledgements. We thank Jan Oliver Ringert for helpful advice. 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] [n.d.]. Spectra Website. http://smlab.cs.tau.ac.il/syntech/spectra/.

[2] [n.d.]. Supporting Materials Website. https://smlab.cs.tau.ac.il/syntech/validate/.

[3] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée C. Bryce. 2018. Combinatorial-based event sequence testing of Android applications. *Inf. Softw. Technol.* 99 (2018), 98–117. https://doi.org/10.1016/j.infsof.2018.03.007

[4] Gal Amram, Shahar Maoz, Itai Segall, and Matan Yossef. 2022. Dynamic Update for Synthesized GR(1) Controllers. In *ICSE*. ACM, 786–797. https://doi.org/10.1145/3510003.3510054

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. on Software Engineering* 32, 8 (2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[6] K. Z. Bell and M. A. Vouk. 2005. On effectiveness of pairwise methodology for testing network-centric software. In *ICICT*. IEEE, 221–235.

[7] Paul E. Black, Vadim Okun, and Yaacov Yesha. 2000. Mutation Operators for Specifications. In *ASE*. IEEE, 81. https://doi.org/10.1109/ASE.2000.873653

[8] 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

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

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

[11] Richard E Boyatzis. 1998. *Transforming qualitative information: Thematic analysis and code development.* SAGE.

[12] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[13] K. Burroughs, A. Jain, and R.L. Erickson. 1994. Improved quality of protocol testing through techniques of experimental design. In *SUPERCOMM/ICC*. 745–752.

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

[15] Davide G. Cavezza, Dalal Alrajeh, and András György. 2019. Minimal Assumptions Refinement for GR(1) Specifications. *CoRR* abs/1910.05558 (2019). arXiv:1910.05558 http://arxiv.org/abs/1910.05558

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

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. The MIT Press.

[18] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. on Softw. Eng.* 23, 7 (1997), 437–444.

[19] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Eng.* 34, 5 (2008), 633–650. https://doi.org/10.1109/TSE.2008.50

[20] M. B. Cohen, J. Snyder, and G. Rothermel. 2006. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes* 31, 6 (2006), 1–9.

[21] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-Based Testing in Practice. In *ICSE*. 285–294.

[22] 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

[23] Gordon Fraser and Franz Wotawa. 2009. Complementary Criteria for Testing Temporal Logic Properties. In *Tests and Proofs*, Catherine Dubois (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–73.

[24] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. 2006. An evaluation of combination strategies for test case selection. *Empirical Softw. Eng.* 11, 4 (2006), 583–611.

[25] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*. ACM, 654–665. https://doi.org/10.1145/2635868.2635929

[26] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734. https://doi.org/10.1109/TSE.2002.1027796

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

[28] 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

[29] D. R. Kuhn, Raghu N. Kacker, and Y. Lei. 2013. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC.

[30] D. R. Kuhn and M. J. Reilly. 2002. An Investigation of the Applicability of Design of Experiments to Software Testing. 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center.

[31] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Softw. Eng.* 30, 6 (2004), 418–421.

[32] Y. Lei and K.-C. Tai. 1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98)*. 254–261.

[33] Spyros Maniatopoulos, Philipp Schillinger, Vitchyr Pong, David C. Conner, and Hadas Kress-Gazit. 2016. Reactive high-level behavior synthesis for an Atlas humanoid robot. In *ICRA*, Danica Kragic, Antonio Bicchi, and Alessandro De Luca (Eds.). IEEE, 4192–4199. https://doi.org/10.1109/ICRA.2016.7487613

[34] 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

[35] Shahar Maoz and Jan Oliver Ringert. 2021. Spectra: a specification language for reactive systems. *Softw. Syst. Model.* 20, 5 (2021), 1553–1586. https://doi.org/10.1007/s10270-021-00868-z

[36] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. 2019. Symbolic repairs for GR(1) specifications. In *ICSE*. IEEE / ACM, 1016–1026. https://dl.acm.org/citation.cfm?id=3339632

[37] Shahar Maoz and Rafi Shalom. 2020. Inherent Vacuity for GR(1) Specifications. In *ESEC/FSE*. 99–110.

[38] Shahar Maoz and Rafi Shalom. 2021. Unrealizable Cores for Reactive Systems Specifications. In *ICSE*. IEEE, 25–36. https://doi.org/10.1109/ICSE43902.2021.00016

[39] Shahar Maoz and Ilia Shevrin. 2020. Just-In-Time Reactive Synthesis. In *ASE*. IEEE, 635–646. https://doi.org/10.1145/3324884.3416557

[40] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. 2019. Specification Patterns for Robotic Missions. *CoRR* abs/1901.02077 (2019). arXiv:1901.02077 http://arxiv.org/abs/1901.02077

[41] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 2012. Combining model-based and combinatorial testing for effective test case generation. In *ISSTA*. ACM, 100–110. https://doi.org/10.1145/2338965.2336765

[42] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. 2015. Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection. *IEEE Trans. Software Eng.* 41, 9 (2015), 901–924. https://doi.org/10.1109/TSE.2015.2421279

[43] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL*. ACM Press, 179–190.

[44] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *ICSM*. IEEE Computer Society, 255–264. https://doi.org/10.1109/ICSM.2007.4362638

[45] I. Segall, R. Tzoref-Brill, and E. Farchi. 2011. Using Binary Decision Diagrams for Combinatorial Test Design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. 254–264.

[46] Fabio Somenzi. 2015. CUDD: CU Decision Diagram Package Release 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[47] M. Trakhtenbrot. 2017. Mutation Patterns for Temporal Requirements of Reactive Systems. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 116–121. https://doi.org/10.1109/ICSTW.2017.27

[48] Rachel Tzoref-Brill. 2019. Chapter Two - Advances in Combinatorial Testing. *Adv. Comput.* 112 (2019), 79–134. https://doi.org/10.1016/bs.adcom.2017.12.002

[49] D. R. Wallace and D. R. Kuhn. 2001. Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data. In *ACS/ IEEE International Conference on Computer Systems and Applications*. 301–311.

[50] Eric Wete, Joel Greenyer, Andreas Wortmann, Oliver Flegel, and Martin Klein. 2021. Monte Carlo Tree Search and GR(1) Synthesis for Robot Tasks Planning in Automotive Production Lines. In *MODELS*. IEEE, 320–330. https://doi.org/10.1109/MODELS50736.2021.00039

[51] A. W. Williams. 2000. Determination of Test Configurations for Pair-Wise Interaction Coverage. In *TestCom*. 59–74.

[52] P. Wojciak and R. Tzoref-Brill. 2014. System Level Combinatorial Testing in Practice – The Concurrent Maintenance Case Study. In *ICST*. 103–112.

[53] Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. 2020. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. *IEEE Trans. Software Eng.* 46, 3 (2020), 302–320. https://doi.org/10.1109/TSE.2018.2852744

[54] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy combinatorial test case generation using unsatisfiable cores. In *ASE*. ACM, 614–624. https://doi.org/10.1145/2970276.2970335

[55] Z. Zhang and J. Zhang. 2011. Characterizing Failure-causing Parameter Interactions by Adaptive Testing. In *ISSTA*. 331–341.