The Challenges in Specifying and Explaining Synthesized Implementations of Reactive Systems*

Hadas Kress-Gazit
Sibley School of Mechanical and Aerospace Engineering
Cornell University, Ithaca, NY, USA
hadaskg@cornell.edu

Hazem Torfah†
Reactive Systems Group
Saarland University, Saarbrücken, Germany
torfah@react.uni-saarland.de

In formal synthesis of reactive systems an implementation of a system is automatically constructed from its formal specification. The great advantage of synthesis is that the resulting implementation is correct by construction; therefore there is no need for manual programming and tedious debugging tasks. Developers remain, nevertheless, hesitant to using automatic synthesis tools and still favor manually writing code. A common argument against synthesis is that the resulting implementation does not always give a clear picture on what decisions were made during the synthesis process. The outcome of synthesis tools is mostly unreadable and hinders the developer from understanding the functionality of the resulting implementation. Many attempts have been made in the last years to make the synthesis process more transparent to users. Either by structuring the outcome of synthesis tools or by providing additional automated support to help users with the specification process.

In this paper we discuss the challenges in writing specifications for reactive systems and give a survey on what tools have been developed to guide users in specifying reactive systems and understanding the outcome of synthesis tools.

1 Introduction

Synthesis is a procedure in which an implementation of a system is automatically constructed from a logical specification. The resulting implementation is correct by construction and no further coding tasks are needed. Synthesis allows developers to focus on determining *what* a system should do rather than *how* it should do it. The task of the developer thus is shifted from writing a program that implements the system to writing a specification for it. This comes with the big advantage of allowing systems to be analyzed at early design stages and disposes of tedious and costly implementation efforts in later stages.

In the last decade, the theoretical ideas of synthesis have been translated into several tools (cf. [34, 24, 22, 9, 8, 23]). The tools have made it possible to tackle real-world design problems, such as the synthesis of an arbiter for the AMBA AHB bus, an open industrial standard for the on-chip communication and management of functional blocks in system-on-a-chip designs, or the IBM generalized buffer, which was synthesized from a specification written in PSL [7]. Nevertheless, synthesis tools have barely been used outside the scientific community. Developers are hesitant to use automatic synthesis, and rather rely on self-created and self-maintained code, or use established legacy code. A common argument against synthesis is the high structural complexity of the resulting implementation. In most cases, synthesized implementations are not easy to follow and do not allow to structurally reason about the functionality of the system nor backtrack any mistakes introduced in its specification. The outcome of synthesis tools thus remains as a black box for developers that is hard to explore manually and where retracing relevant aspects of the input specification becomes infeasible.

*This work was partly funded by the European Research Council (ERC) Grant OSARES (No. 683300)
†Corresponding author

B. Finkbeiner and S. Kleinberg: Third International Workshop
on Formal Reasoning about Causation, Responsibility,
and Explanations in Science and Technology (CREST 2018)
EPTCS 286, 2019, pp. 50–64, doi:10.4204/EPTCS.286.5

One might argue that it is not the role of synthesis to provide understandable implementations more than correct ones. However, the correctness of synthesized implementation is only relative to the provided specification. In other words, the quality of the resulting implementation is only as good as its input specification. Understanding the functionality of the outcome is thus vital for writing correct and high quality specifications. Tool support for refining specifications is thus vital for a correct synthesis outcome that indeed fulfills all the user's design intents.

In this paper, we investigate the challenges in writing specifications for reactive systems and understanding automatically synthesized implementations. The setting we are interested in is given in Figure 1, where given a set of inputs from the environment (sensors) and a set of outputs of the system (actuators), a synthesis tool constructs an implementation of the system that satisfies a high-level specification written over the inputs and outputs (correct reaction of actuators to sensor information). Specifications are

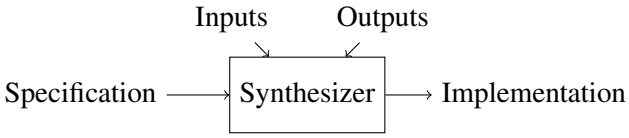


Figure 1: The Synthesis Problem

usually given as formulas in a temporal logic that define the relations between inputs and outputs. Implementations are realizations of these relations represented as transducers (Mealy or Moore machines).

Issues with the specifications in the synthesis process are captured by the problems of unrealizability and completeness of specifications. When writing specifications, one might over-specify the systems, such that, no implementation can realize the system's specification. One might also under-specify parts of the system which results in synthesized implementations that satisfy the given specification but still do not meet the designers intents, i.e., they behave not as the designer expected for certain input scenarios. Challenges on the implementation side involve the understandability of the resulting implementation and transparency regarding why the specific implementation was chosen from the set of all possible correct implementations.

We present a series of works that have addressed the construction of more structured and understandable implementations. In general, we can summarize the concerns using two questions: (1) How do we assure that the synthesized implementation, which is one of many, is one that corresponds to the user's expectations? (2) How do we support the user in writing correct specifications, that include all the relevant aspects needed for the construction of an implementation with all of the intended functionality?

We give an overview on challenges that we face on both the specification and implementation side of the synthesis process. We describe methods that are used for the analysis of specifications. Either by pointing out erroneous cores in a specification or indicating what assumptions have not been considered by the user. Tools can, for instance, return minimal specification revisions to make an unrealizable specification realizable. Dually, they should also identify vacuous parts of specifications, and help to eliminate ambiguities in the specification. Regarding the outcome of synthesis tools, we raise the question of how to determine the quality of a resulting implementation: Are there other artifacts that can be additionally generated to aid the user in understanding or validating the implementation and the specification? Are there understandable witnesses that validate the black-box implementation obtained as the output of the synthesis tool? What further metrics can be used to debug specifications, reason about implementations and facilitate the composition of the implementation in a larger system?

2 Writing Specifications for Reactive Systems

Reactive systems are those systems that react to inputs from an environment. A specification of a reactive system thus defines how an implementation of the system should behave in response to inputs from this environment. A specification usually includes assumptions on the environment, which define the scope in which the implementation should behave correctly, and guarantees that define the correct behavior of the system under those assumptions. A synthesis procedure tries then to construct an implementation that fulfills all the guarantees under the assumptions declared over the environment.

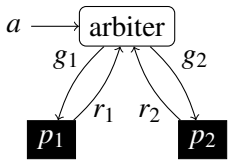
In general, two types of problems may occur when specifying a reactive system. One might over-specify the system making the specification become unrealizable, i.e., there is no implementation that satisfies the specification. One might under-specify the system by leaving out many relevant details, that are crucial for a synthesized implementation that behaves as the user expected.

The first type of error is detected by the synthesis tools. If a specification is unrealizable, the synthesizer is not able to return an implementation and may return a counterexample that captures the change in the inputs that will make the system fail. The second type of error is harder to detect because the synthesizer terminates with an implementation of the system, but gives no further information about how the unspecified parts of the implementation were constructed.

In the following we discuss both types of specification errors and show how synthesis tools can potentially leverage each of these errors for the purpose of correcting specifications.

2.1 Unrealizability of Specifications

A specification is unrealizable, either because it is unsatisfiable, i.e., the value of the specification is equal to false due to inconsistencies in the specification, or it is unrealizable because there exists no implementation that behaves correctly over all input sequences of the environment. Consider for example an arbiter over two processes as given in Figure 2, and consider a specification for the arbiter given by the conjunction of the LTL formulas *response* and *access*:



$$\textit{response}\colon \bigwedge_{i\in\{1,2\}} \square(r_i \rightarrow \diamondsuit g_i)$$

$$\textit{access}\colon \square(\neg a \rightarrow \neg g_1 \wedge \neg g_2)$$

Figure 2: An arbiter over two processes and a specification for the arbiter given as a conjunction of the formulas *response* and *access*.

The arbiter controls the access of the processes p_1 and p_2 into a shared resource. The processes p_1 and p_2 request access to the resource via the signals r_1 and r_2, respectively. The arbiter grants access to the processes using the respective signals g_1 and g_2. A further external signal a determines when the resource can be accessed. The signals r_1, r_2 and a compose the inputs of the environment and g_1 and g_2 are the outputs of the system.

The specification `response` $\wedge$ `access` is unrealizable. The environment can always set the input signal a to false, forbidding the arbiter from sending any grants g_1 or g_2. Thus, there is no implementation that satisfies the specification *response* for an input of the environment, where a request r_i has been sent

to the arbiter by one of the processes and where the signal a is always false[1].

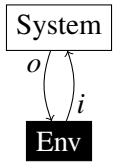
In this section we present a list of artifacts for explaining unrealizability, methods for detecting unrealizable cores of specifications, and how to modify unrealizable specifications to get realizable ones.

2.1.1 Detecting and mitigating unrealizability

Checking the realizability of specifications can be seen as a game theoretic problem where two players, the environment and the system, interchangeably produce input and outputs over an infinite duration [12]. Without loss of generality we assume that in our setting the system player starts the game, by initializing the values of the atomic propositions of the system. An implementation of a system for a given specification is a winning strategy for the system player in that game. A specification is realizable if there exists a winning strategy for the system. A specification is unrealizable if for each strategy of the system, there is an input sequence of the environment where the strategy loses the game, i.e., for which the strategy is forced to produce an output that violates the specification. Consider the unrealizable specification given in Figure 2. No matter what strategy the systems chooses, the environment challenges the strategy with the input sequence $\{r_1\}\{\}^\omega$, where the process p_1 sends a request to the arbiter but where the signal a is always set to false. As any correct strategy must give a grant, but at the same time is not allowed to, because the signal a is always false, no strategy is able to fulfill both the specifications `response` and `access`.

If a specification is unrealizable, then there is a set of input sequences for which no matter what strategy the system chooses, the strategy will produce a violating output sequence on at least one of those input sequences. We call such a set of input sequences a counterexample set for the unrealizable specification. In the setting considered in this paper, finding a counterexample set can be done by solving the synthesis game. For more general settings such as distributed architectures or settings with incomplete information the problem is in general undecidable [42].

The counterexample set can grow infinitely large. Consider for example the following architecture and LTL specification:



$$\varphi = \Diamond(o \leftrightarrow i)$$

Figure 3: An unrealizable LTL with an infinite set of counterexamples

The specification requires the environment to send an input i if and only if the system outputs o. This specification is unrealizable as the system has no control over the environment[2]. A counterexample set for the specification is given by the set $\Gamma = (2^{\{i\}})^\omega$, and there is no finite set $\Gamma' \subset \Gamma$ that is a counterexample set for φ. Assume there is a finite set Γ' that is a counterexample set for φ. Because, Γ' is finite, there is a position j such that all prefixes of length j of the sequences in Γ' are pairwise different. As the sequences in Γ' can be distinguished at position j we can choose a strategy for the system that assigns the value of o at position $j+1$ to true if the input i at this position is true, and otherwise sets o to false. This strategy satisfies the property φ over all sequences in Γ' and thus Γ' cannot be a counterexample set for φ [28].

[1]The specification is an example of an unrealizable nevertheless satisfiable specification. An example input sequence that has a corresponding satisfying output sequence is for example the sequence $\{a, g_1, r_1\}^\omega$.

[2]Remember the system moves first.

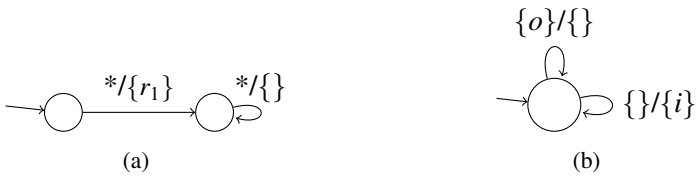


Figure 4: Counterstrategies for the unrealizable specifications `access` $\wedge$ `response` in Figure 2 and φ in Figure 3 given as Mealy machines.

A convenient finite representation of the possibly infinite set of counterexamples can be given by a *counterstrategy*. A counterstrategy is a winning strategy for the environment, and it is computed by solving the synthesis game for the environment player instead of the system player. A counterstrategy for the unrealizable specification in Figure 2 is given in Figure 4(a). The strategy responds to the first output of the system by assigning the input r_1 to true and assigns all subsequent inputs to false independent of the chosen outputs by the system. A counterstrategy for the specification φ in Figure 3 is given in Figure 4(b). The strategy assigns the input signal i to true if the system outputs false and to true otherwise. In this way the system will never fulfill the specification φ.

Complex specifications may lead to large and complex counterstrategies that are difficult to inspect manually. In many cases, there is no need to consider the whole counterstrategy to infer what parts of the specification are unrealizable. A smaller set of input sequences might already suffice to decide the unrealizability of the specification. Some techniques rely on pruning parts of the counterstrategy that are irrelevant for its unrealizability in order to make the counterstrategy more readable [10]. Further works suggested to only return a sufficient set of input scenarios of the environment instead of returning the whole counterstrategy. An alternative, for example, are countertraces [36], which are fixed input traces for which there is no output trace fulfilling the specification. One problem with countertraces nevertheless is that they are hard to compute and sometimes they do not exist. In case of safety properties one can compute a finite counterexample set of finite sequences using the symbolic method presented in [28]. The finite sequences resemble scenarios where the system violates the safety property after finitely many steps. The method involves a procedure that incrementally increases the bound on the size of input sequences until a counterexample set is found. The big advantage of this method is that it also provides a semi-decision procedure for the unrealizability problem over undecidable distributed architectures.

Treating unrealizability can also be done by directly analyzing the specification itself, for example by identifying unrealizable cores of the specification (e.g. [45, 35, 21, 39]). An unrealizable core is a sub-specification that is unrealizable on its own. Consider our arbiter example again and let:

$$\varphi_1 = \square(r_1 \rightarrow \diamondsuit g_1)$$
$$\varphi_2 = \square(r_2 \rightarrow \diamondsuit g_2)$$
$$\varphi_3 = \square(\neg a \rightarrow \neg g_1 \wedge \neg g_2)$$

The specification contains the following minimal unrealizable cores: $C_1 = \{\varphi_1, \varphi_3\}$ and $C_2 = \{\varphi_2, \varphi_3\}$. To make the specification realizable, one has to resolve both the conflicts C_1 and C_2. This can be done by either weakening the specifications φ_1 and φ_2, for example, by relaxing the eventuality to $\square(r_1 \rightarrow (\neg a \vee g_2) W g_1)$ and $\square(r_2 \rightarrow (\neg a \vee g_1) W g_2)$ using the *weak until* operator W. In this way, the requests r_1 and r_2 must be answered by the respective grants, as soon as the access signal a becomes true, otherwise the specification specifically states that no grants are to be given. Another possibility to

make the specification realizable is by restricting the behavior of the environment. The main reason why the specification is not realizable is because the environment can choose not to set the signal a to true. However, this assumption on the behavior of the environment is not necessarily realistic. We can add a further assumption that states that the environment will grant access to the shared resource an infinite number of times, namely $\varphi_4 = \square\diamondsuit a$, making the specification realizable.

Detecting unrealizability is only the first step. As important is assisting the developer in repairing the specification. A series of works [3, 13, 38, 2, 15, 17, 43, 14, 31] introduced frameworks that leverage the artifacts above to turn an unrealizable specification into a realizable one.

As a specification for a reactive systems includes assumptions on the environment and guarantees to be fulfilled by the system, making a specification realizable can be done by either strengthening the assumptions on the environment or weakening the guarantees of the system. Strengthening the assumptions on the environment is done by adding further assumptions that remove certain scenarios for which the specification is unrealizable. Weakening the guarantees is done by tolerating additional behaviors of the system. Most approaches rely on a counterexample-guided refinement loop to learn the new assumptions [3, 13, 38, 2, 15, 17]. In each refinement loop a counterstrategy is used to extract new assumptions for the environment. Some approaches try to directly learn assumptions on the environment by first computing a safety assumption that removes a minimal set of environment edges from the game graph, and then computing a liveness assumption that puts fairness conditions on some of the remaining environment edges [14, 31].

An interactive approach to identifying the cause of failure in an unrealizable specification was presented in [43]. Here, a game-based approach is presented where the user attempts to fulfill a robot specification against an adversarial environment. The idea of the approach is to highlight bad portions of the specification and identify example executions for the environment that make the system fail.

2.2 Incomplete Specifications

A common error when specifying systems is to under-specify them. In this case, synthesis tools will return an implementation for the given specification but that may still behave different than the user expected. Revisiting the two process arbiter given in Figure 2, assume we want to synthesize an implementation for the arbiter that is mutually exclusive and where every request is guaranteed to be answered eventually. A specification for the arbiter can be given by the respective LTL formulas $\square(\neg g_1 \vee \neg g_2)$, and $\square(r_1 \rightarrow \diamondsuit g_1)$ and $\square(r_2 \rightarrow \diamondsuit g_2)$. A possible outcome of the synthesis tool could be an implementation as given in Figure 5(a). The implementation returns immediately a grant g_1 every time there is a request r_1 and a grant g_2 whenever there is a request r_2. If both request r_1 and r_2 occur at the same time, the implementation prioritizes process p_1 by first giving a grant g_1 and as soon as process p_1 is done, it grants p_2 access to the shared resource. The decision to give p_1 priority was made by the synthesis tool. If the user is not happy with prioritizing process p_1 then an additional specification must be added by the user to handle simultaneous requests more adequately.

The implementation in Figure 5(a) is not the only realization of the arbiter's specification. Figure 5(b) shows another implementation for the arbiter that interchangeably returns grant g_1 and g_2 without considering what requests were made by the processes. This means that the grants are given even if no requests were made by the processes, which is not necessarily what the user intended by the specification. This further means that the specification was not explicit enough on whether a grant depends on the requests, as in the previous implementation. To avoid the construction of such implementations, the specification must be refined.

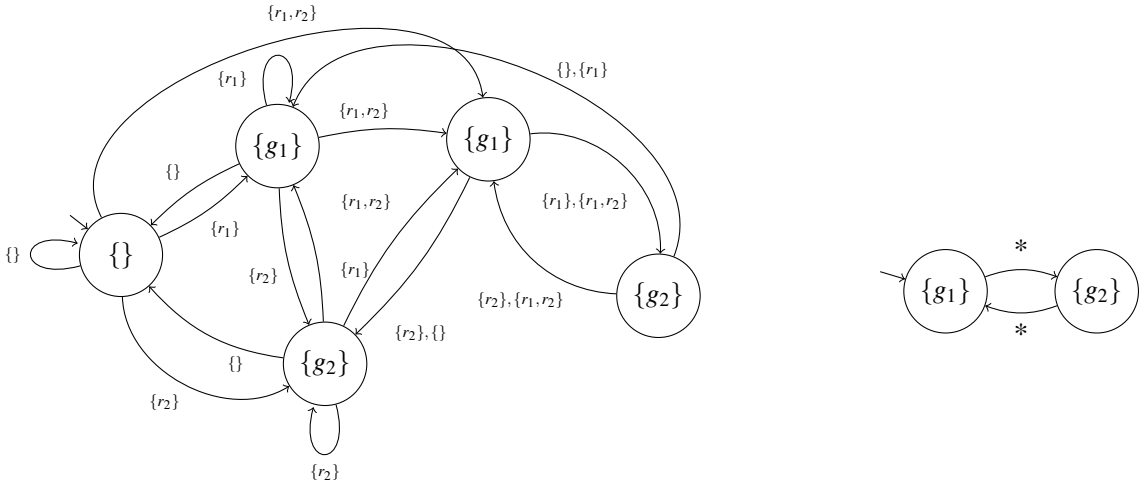


Figure 5: Two different implementations for the specification $\square(\neg g_1 \vee \neg g_2) \wedge \square(r_1 \rightarrow \diamondsuit g_1) \wedge \square(r_2 \rightarrow \diamondsuit g_2)$.

One possible modification could be to change the specifications describing the responsiveness of the arbiter to $\square(r_1 \leftrightarrow \diamondsuit g_1)$ and $\square(r_2 \leftrightarrow \diamondsuit g_2)$. In this way it is more likely that an implementation such as the one in Figure 5(a)is enforced.

Completeness of specifications cannot be defined formally, as it is dependent on the user's design intents. Nevertheless, with respect to this, we can say that a specification is complete if no implementation that satisfies the specification is incorrect with respect to the user's intent. In general, debugging an incomplete specification is a multistage refinement process. In the following we present some methods on how to aid the user throughout this process to construct a complete specification.

2.2.1 Detecting vacuity in specifications

Different synthesis procedures result in different implementations for the same specification. The reason for that is that parts of the implementation that are not explicitly defined by the specification are completed by the underlying decision procedure of the synthesis tools. For example, in the implementation in Figure 5(a), the synthesis procedure decided to set the values of the atomic propositions g_1 and g_2 to false in the initial state, as the specification did not explicitly state what the values of these atomic propositions should be. Another synthesis procedure could have chosen different values as long as mutual exclusion is ensured.

To understand which parts of the implementation were forced by the specification and which parts were decided by the synthesis procedure, one has to perform a coverage analysis on the resulting implementation. Intuitively, an atomic proposition of a state in a transition system is covered by the specification if changing the value of the atomic proposition in that state falsifies the specification [20]. For example changing the value of the atomic proposition g_1 in the initial state of the transition system in Figure 5(a) from false to true does not falsify the specification. Thus, the value of g_1 in the initial state is

not covered by the specification. In the transition system in Figure 5(b) on the other hand, changing the value of g_1 does violate the specification.

Definitions of coverage range from qualitative definitions like the above to quantitative versions based on certain metrics [4, 5, 32, 20, 19]. A variant of coverage is one based on causality. In the implementation in Figure 5(b) choosing g_1 to be true in the initial state forced g_2 to be true in the other state. Thus, the decision made in the other state is caused by the decisions made in the initial state. If changing the value of an atomic proposition a in one state q does not falsify the specification, one should check whether there is a set of states Y, such that, changing the value of a and the values of atomic propositions in Y falsifies the specification. If this is the case, then choosing the current values of the atomic propositions in Y has a causal relation to choosing the value of a in q.

Using the various coverage definitions the designer can examine synthesized implementations and modify the specification accordingly. This requires several synthesis and refinement steps until a complete specifications is reached that enforces a desired implementation for the system, for example, to get an implementation as in Figure 5(a) instead of another implementation like in Figure 5(b). By taking a closer look into the arbiter's specification and the usual mechanism of requests and grants, it is clear to a human observer that the user intended grants to be given upon request from the processes. A system that receives no requests from the processes should not send out unnecessary permissions to enter the shared resource. A smart synthesis algorithm will construct an implementation that considers each part of the specification entered by the user and avoids implementations like the one in Figure 5(b), which vacuously satisfy the specification by ignoring parts of the specification, in this case the values of the signals r_1 and r_2. We say that an implementation non-vacuously satisfies a specification if it satisfies the specification but not any strengthening of the specification [6]. Instead of synthesizing any transition system that satisfies the specification, a good synthesis procedure would construct a non-vacuous implementation that covers all parts of the specification [6].

In general it is useful to inform the user on the decisions made during the synthesis process. This helps understand which parts were implemented independently by the synthesis procedure and which parts were forced by the specification. Synthesis tools thus need to provide additional relevant information accompanied with each synthesized implementation. A first step towards this direction is the construction of skeletons for specifications [29]. Skeletons are transition systems, where states are labeled with a three-valued assignment to the output variable: in each state an output can be true, false, or open, which means that the specification allows implementations with either value for a variable in that state. States with open variables indicate that additional constraints may be added to complete the specification according the user's intent. Skeletons can additionally be constructed with each synthesized implementation. For example, a skeleton for the transition system in Figure 5(a) is given in Figure 6. Notice that from the skeleton we can read that the implementation of the initial state and the decision to prioritize process p_1 are marked with "?", indicating that these choices were made by the synthesis procedure and were not explicitly determined by the specification.

2.2.2 Monitoring the implementation

In many cases the environment assumptions may not be known to the user in full, which results in implementations with incorrect behavior. Many violations of the environment assumptions can be detected during runtime or during simulation. To better understand the violations, one can deploy monitors that give feedback on what caused the violation of these assumptions and modify the specification of the system accordingly. In an automated feedback-based process, the specification of the system is augmented with new environment assumptions that are computed at runtime. Whenever the automated process fails,

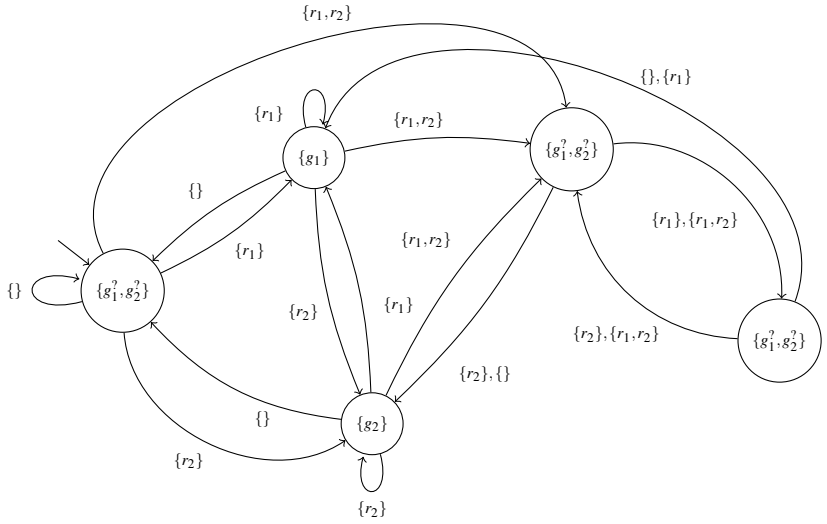


Figure 6: A skeleton for the implementation in Figure 5(a) and the specification $\square(\neg g_1 \vee \neg g_2) \wedge \square(r_1 \rightarrow \diamondsuit g_1) \wedge \square(r_2 \rightarrow \diamondsuit g_2)$.

feedback is provided to the user, who is then asked to resolve the conflict by modifying the specification [46, 47].

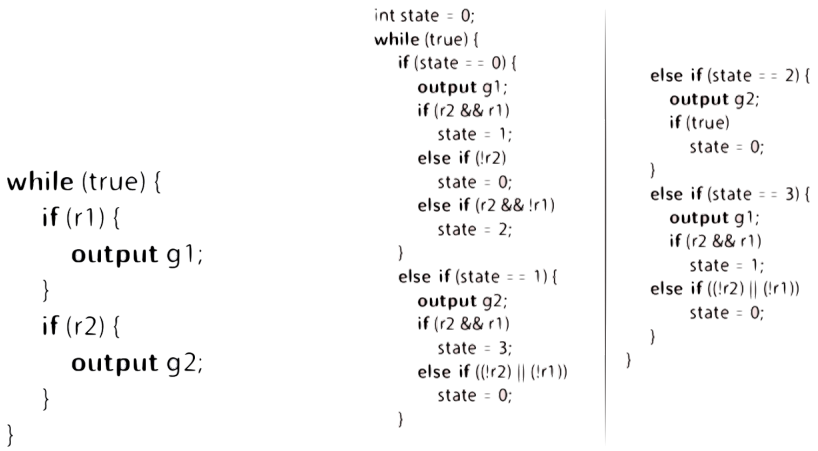
3 Analyzing the Outcome of Synthesis Tools

In most cases, the structure of an implementation produced by a synthesis tool is very complex and hard to examine, and thus it is a challenge to convince the user that a synthesized implementation indeed does what it is actually supposed to do by just looking at it. Figures 7 and 8 show examples of synthesized and manually written implementation of two and three client arbiters. Notice that increasing the number of clients by one results in a large blow up in the synthesized implementation.

In order to make an implementation more understandable, synthesis tools must either synthesize structurally less complex implementations or provide the user with additional information that make the resulting implementation easy to follow. In the following we show some of the improvements that have been made to make the outcome of synthesis tools more understandable for the developer.

3.1 Representation of Implementations

A synthesized implementation of a system from its specification is given by a transducer (a Mealy or Moore machine). Due to its large state space, there is a general trend to represent transducers succinctly by binary decision diagrams (BDD) or circuits [33]. Such artifacts give symbolic representations of transducers that are easy to process but have the drawback of not mirroring the original functional choices of an implementation. Looking at a binary decision diagram, the developer will not be able understand the functionality of the implementation easily. Many works have been devoted to minimizing or simplifying BDDs [1, 37, 30], but such operation are however notoriously difficult. Some also tried



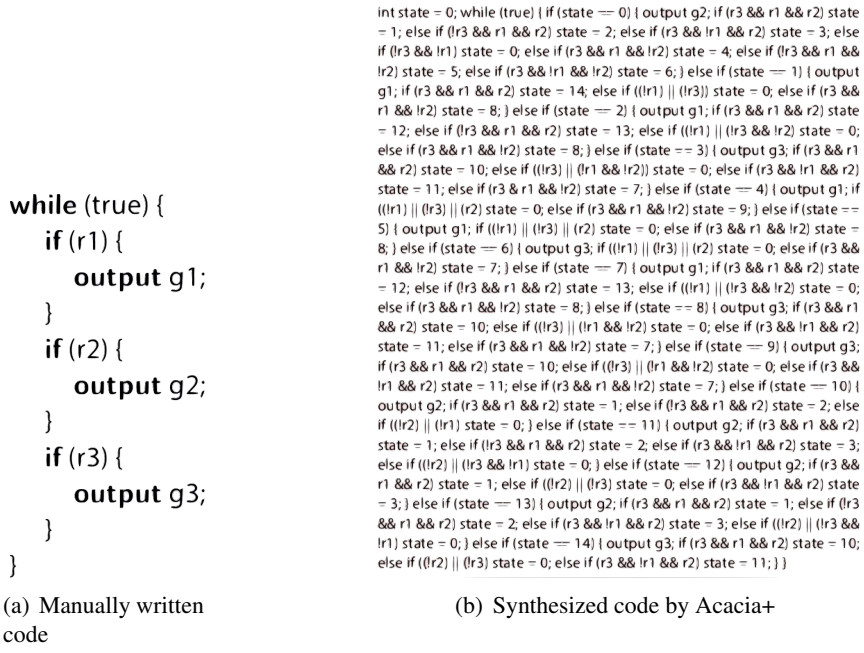
```
while (true) {
    if (r1) {
        output g1;
    }
    if (r2) {
        output g2;
    }
}
```

(a) Manually written code

```
int state = 0;
while (true) {
    if (state == 0) {
        output g1;
        if (r2 && r1)
            state = 1;
        else if (!r2)
            state = 0;
        else if (r2 && !r1)
            state = 2;
    }
    else if (state == 1) {
        output g2;
        if (r2 && r1)
            state = 3;
        else if ((!r2) || (!r1))
            state = 0;
    }
    else if (state == 2) {
        output g2;
        if (true)
            state = 0;
    }
    else if (state == 3) {
        output g1;
        if (r2 && r1)
            state = 1;
        else if ((!r2) || (!r1))
            state = 0;
    }
}
```

(b) Synthesized code by Acacia+

Figure 7: A manually written vs. a synthesized program for a two client arbiter



```
while (true) {
    if (r1) {
        output g1;
    }
    if (r2) {
        output g2;
    }
    if (r3) {
        output g3;
    }
}
```

(a) Manually written code

```
int state = 0; while (true) { if (state == 0) { output g2; if (r3 && r1 && r2) state
= 1; else if (!r3 && r1 && r2) state = 2; else if (r3 && !r1 && r2) state = 3; else
if (!r3 && !r1) state = 0; else if (r3 && r1 && !r2) state = 4; else if (!r3 && r1 &&
!r2) state = 5; else if (r3 && !r1 && !r2) state = 6; } else if (state == 1) { output
g1; if (r3 && r1 && r2) state = 14; else if ((!r1) || (!r3)) state = 0; else if (r3 &&
r1 && !r2) state = 8; } else if (state == 2) { output g1; if (r3 && r1 && r2) state
= 12; else if (!r3 && r1 && r2) state = 13; else if ((!r1) || (!r3 && !r2) state = 0;
else if (r3 && r1 && !r2) state = 8; } else if (state == 3) { output g3; if (r3 && r1
&& r2) state = 10; else if ((!r3) || (!r1 && !r2)) state = 0; else if (r3 && !r1 && r2)
state = 11; else if (r3 & r1 && !r2) state = 7; } else if (state == 4) { output g1; if
((!r1) || (!r3) || (r2) state = 0; else if (r3 && r1 && !r2) state = 9; } else if (state ==
5) { output g1; if ((!r1) || (!r3) || (r2) state = 0; else if (r3 && r1 && !r2) state =
8; } else if (state == 6) { output g3; if ((!r1) || (!r3) || (r2) state = 0; else if (r3 &&
r1 && !r2) state = 7; } else if (state == 7) { output g1; if (r3 && r1 && r2) state
= 12; else if (!r3 && r1 && r2) state = 13; else if ((!r1) || (!r3 && !r2) state = 0;
else if (r3 && r1 && !r2) state = 8; } else if (state == 8) { output g3; if (r3 && r1
&& r2) state = 10; else if ((!r3) || (!r1 && !r2) state = 0; else if (r3 && !r1 && r2)
state = 11; else if (r3 && r1 && !r2) state = 7; } else if (state == 9) { output g3;
if (r3 && r1 && r2) state = 10; else if ((!r3) || (!r1 && !r2) state = 0; else if (r3 &&
!r1 && r2) state = 11; else if (r3 && r1 && !r2) state = 7; } else if (state == 10) {
output g2; if (r3 && r1 && r2) state = 1; else if (!r3 && r1 && r2) state = 2; else
if ((!r2) || (!r1) state = 0; } else if (state == 11) { output g2; if (r3 && r1 && r2)
state = 1; else if (!r3 && r1 && r2) state = 2; else if (r3 && !r1 && r2) state = 3;
else if ((!r2) || (!r3 && !r1) state = 0; } else if (state == 12) { output g2; if (r3 &&
r1 && r2) state = 1; else if ((!r2) || (!r3) state = 0; else if (r3 && !r1 && r2) state
= 3; } else if (state == 13) { output g2; if (r3 && r1 && r2) state = 1; else if (!r3
&& r1 && r2) state = 2; else if (r3 && !r1 && r2) state = 3; else if ((!r2) || (!r3 &&
!r1) state = 0; } else if (state == 14) { output g3; if (r3 && r1 && r2) state = 10;
else if ((!r2) || (!r3) state = 0; else if (r3 && !r1 && r2) state = 11; } }
```

(b) Synthesized code by Acacia+

Figure 8: A manually written vs. a synthesized program for a three client arbiter

to use similar but more structured versions of BDDs to make the representation more explanatory [11]. However, the structure remains too complex to explore manually.

In general, the desire is not only to construct small but also structurally simple and understandable implementations. To achieve this goal, algorithms are needed, which perform optimally not only in the input specification, but also in the structural complexity of the implementation, so called output-sensitive algorithms [26]. The first output-sensitive reactive synthesis algorithm was bounded synthesis [27]. In bounded synthesis, the number of states of the implementation to be synthesized is an additional param-

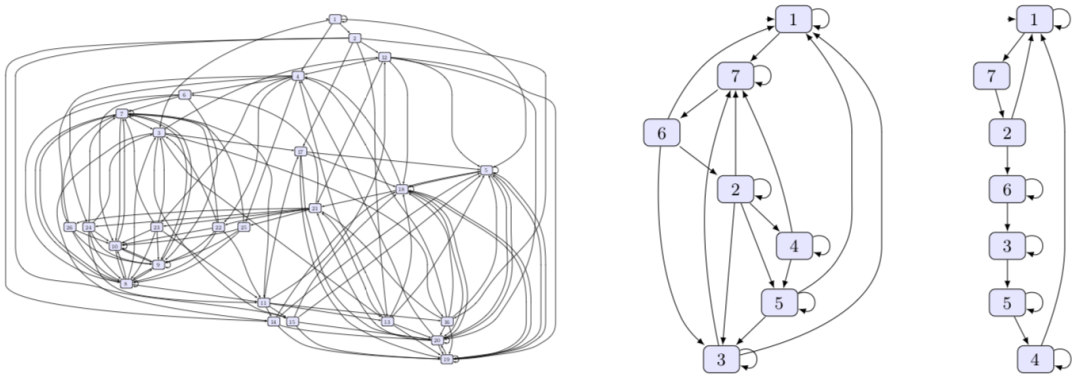


Figure 9: [25] Three implementations of the TBURST4 component of the AMBA bus controller. Standard synthesis with Acacia+ [9] produces the state graph on the left with 14 states and 61 cycles. Bounded synthesis produces the graph in the middle with 7 states and 19 cycles. Bounded cycle synthesis, has 7 states and 7 cycles, which is the minimum.

eter to the synthesis algorithm. Minimal solutions are thus ensured by synthesizing implementations for incrementally increasing bounds. Further metrics that help reduce the structural complexity of the implementations were introduced in [25]. In addition to the size, the number of cycles in the state graph of the transducer is limited by a given bound. Reducing the number of cycles makes an implementation much easier to understand. Figure 9 shows the different structural complexities of transducers synthesized using the bounded size, bounded cycle and a non-output sensitive algorithm.

Other works have tended to reduce the complex synthesis result to a much more understandable version by approximating its behavior. In many cases, the user is not interested in implementation details of fine granularity, and thus, one can abstract these details in the presentation of the transducer. Some methods, especially in the context of probabilistic systems, tend to extract the important parts of the implementation by pruning non-relevant behavior according to a notion of importance. An example of such an approach was presented in [10], where the importance of a state in an implementation is determined by the probability of visiting the state by the strategy.

In an inverse fashion one can incrementally construct the complex implementation starting with a coarse abstraction and gradually refine it with respect to a given partial order, that forces a correct construction. Inspired by counterexample guided abstraction refinement, a series of incremental synthesis procedures have been investigated [40, 41, 44]. In each stage, refinement suggestions give information on what behavior is added or excluded from the implementation. Allowing to observe each refinement step gives a clearer picture regarding the behavior of the implementation.

In all the approaches above, the product of the synthesis procedure is a representation of a transducer. Although transducers are easy to process, they are not necessarily adequate for presenting the synthesis result to the user. The main reason for that is, that in many domains a transducer is not a standard model users tend to work with in their daily projects. Developers of cyber-physical systems for example are familiar with dataflow models. Approaches adapting the idea of synthesizing dataflow models compatible with Simulink[3] or SCADE have become a target of investigation[4]. Instead of directly

[3]https://de.mathworks.com/products/simulink.html

[4]http://www.ansys.com/Products/Embedded-Software/ANSYS-SCADE-Suite.

synthesizing a transducer as in standard LTL synthesis, an actor-based controller using a computational model of synchronous dataflow (SDF) is considered [16, 18]. An actor-based controller defines input and output ports and a set of actors and their wiring. The advantage of actor-based controller is that they abstract implementations details that might not be necessary at first for understanding the behavior of the controller.

4 Conclusion

In this paper, we discussed a number of challenges in automatic synthesis of reactive systems. We presented a list of errors that may happen during the specification process and tools for handling the unrealizability and incompleteness of specification, such as identifying unrealizable cores and vacuous parts of the specification. We also described what obstacles one encounters when trying to understand the outcome of the synthesis process. We explored different artifacts that can be generated to debug specifications and to reason about implementations. Finally, we described different representations for implementations; depending on the domain expertise of the specification designers, synthesis tools should consider which representation would be most beneficial for their target users.

This paper should be seen as an initiator for a broad discussion on how far synthesis has come and how to make it more attractive for users. Also what further tools are needed to aid the user in the specification process and how to make the outcome of the synthesis process more readable and understandable.

References

[1] S. B. Akers (1978): *Binary Decision Diagrams*. *IEEE Trans. Comput.* 27(6), pp. 509–516, doi:10.1109/TC.1978.1675141.

[2] Rajeev Alur, Salar Moarref & Ufuk Topcu (2013): *Counter-Strategy Guided Refinement of GR(1) Temporal Logic Specifications*. doi:10.1109/FMCAD.2013.6679387.

[3] Rajeev Alur, Salar Moarref & Ufuk Topcu (2015): *Pattern-Based Refinement of Assume-Guarantee Specifications in Reactive Synthesis*. In Christel Baier & Cesare Tinelli, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 501–516, doi:10.1007/978-3-662-46681-0_49.

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

[5] Shoham Ben-David, Fady Copty, Dana Fisman & Sitvanit Ruah (2015): *Vacuity in practice: temporal antecedent failure*. *Formal Methods in System Design* 46(1), pp. 81–104, doi:10.1007/s10703-014-0221-0.

[6] Roderick Bloem, Hana Chockler, Masoud Ebrahimi & Ofer Strichman (2017): *Synthesizing Non-Vacuous Systems*. In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pp. 55–72, doi:10.1007/978-3-319-52234-0_4.

[7] Roderick Bloem, Stefan Galler, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Automatic hardware synthesis from specifications: A case study*. In: *In Design, Automation and Test in Europe (DATE*, doi:10.1109/DATE.2007.364456.

[8] Roderick Bloem, Hans-Jrgen Gamauf, Georg Hofferek, Bettina Knighofer & Robert Knighofer (2012): *Synthesizing Robust Systems with RATSY* 84. doi:10.4204/EPTCS.84.4.

[9] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2012): *Acacia+, a Tool for LTL Synthesis*. In P. Madhusudan & Sanjit A. Seshia, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 652–657, doi:10.1007/978-3-642-31424-7_45.

[10] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Andreas Fellner & Jan Křetínský (2015): *Counterexample Explanation by Learning Small Strategies in Markov Decision Processes*, pp. 158–177. Springer International Publishing, Cham, doi:10.1007/978-3-319-21690-4_10.

[11] Tomas Brazdil, Krishnendu Chatterjee, Jan Kretinsky & Viktor Toman (2018): *Strategy Representation by Decision Trees in Reactive Synthesis*. In: *TACAS*, Springer, doi:10.1016/S0304-3975(98)00009-7.

[12] J. Richard Buchi & Lawrence H. Landweber (1990): *Solving Sequential Conditions by Finite-State Strategies*, pp. 525–541. Springer New York, New York, NY, doi:10.1007/978-1-4613-8928-6_29.

[13] Krishnendu Chatterjee & Thomas A. Henzinger (2007): *Assume-Guarantee Synthesis*. In Orna Grumberg & Michael Huth, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 261–275, doi:10.1007/978-3-540-71209-1_21.

[14] Krishnendu Chatterjee, Thomas A. Henzinger & Barbara Jobstmann (2008): *Environment Assumptions for Synthesis*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 147–161, doi:10.1007/978-3-540-85361-9_14.

[15] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann & Arjun Radhakrishna (2010): *Gist: A Solver for Probabilistic Games*. In Tayssir Touili, Byron Cook & Paul Jackson, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 665–669, doi:10.1007/978-3-642-14295-6_57.

[16] Chih-Hong Cheng, Yassine Hamza & Harald Ruess (2016): *Structural Synthesis for GXW Specifications*. In: *International Conference on Computer Aided Verification*, Springer, pp. 95–117, doi:10.1007/978-3-319-89960-2_21.

[17] Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess & Stefan Stattelmann (2014): *G4LTL-ST: Automatic generation of PLC programs*. In: *International Conference on Computer Aided Verification*, Springer, pp. 541–549, doi:10.1007/978-3-319-08867-9_36.

[18] Chih-Hong Cheng, Edward A Lee & Harald Ruess (2017): *autoCode4: Structural Controller Synthesis*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 398–404, doi:10.1007/978-3-662-54577-5_23.

[19] Hana Chockler, Joseph Y. Halpern & Orna Kupferman (2008): *What Causes a System to Satisfy a Specification? ACM Trans. Comput. Logic* 9(3), pp. 20:1–20:26, doi:10.1145/1352582.1352588.

[20] Hana Chockler, Orna Kupferman & Moshe Y. Vardi (2003): *Coverage Metrics for Formal Verification*. In Daniel Geist & Enrico Tronci, editors: *Correct Hardware Design and Verification Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 111–125, doi:10.1007/978-3-540-39724-3_11.

[21] A. Cimatti, M. Roveri, V. Schuppan & A. Tchaltsev (2008): *Diagnostic Information for Realizability*. In Francesco Logozzo, Doron A. Peled & Lenore D. Zuck, editors: *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–67, doi:10.1007/978-3-540-78163-9_9.

[22] Rüdiger Ehlers (2011): *Unbeast: Symbolic Bounded Synthesis*. In Parosh Aziz Abdulla & K. Rustan M. Leino, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 272–275, doi:10.1007/978-3-642-19835-9_25.

[23] Rüdiger Ehlers & Vasumathi Raman (2016): *Slugs: Extensible GR(1) Synthesis*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, *Lecture Notes in Computer Science* 9780, Springer, pp. 333–339, doi:10.1007/978-3-319-41540-6_18.

[24] Peter Faymonville, Bernd Finkbeiner & Leander Tentrup (2017): *BoSy: An Experimentation Framework for Bounded Synthesis*. In: *Proceedings of CAV*, *LNCS* 10427, Springer, pp. 325–332, doi:10.1007/978-3-319-63390-9_17.

[25] Bernd Finkbeiner & Felix Klein (2016): *Bounded Cycle Synthesis*. *Lecture Notes in Computer Science* 9779, Springer Berlin Heidelberg, doi:10.1007/978-3-319-41528-4.

[26] Bernd Finkbeiner & Felix Klein (2017): *Reactive Synthesis: Towards Output-Sensitive Algorithms*. In Alexander Pretschner, Doron Peled & Thomas Hutzelmann, editors: *Dependable Software Systems Engineering*, *NATO Science for Peace and Security Series, D: Information and Communication Security* 50, IOS Press, pp. 25–43, doi:10.3233/978-1-61499-810-5-25.

[27] Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. *International Journal on Software Tools for Technology Transfer* 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[28] Bernd Finkbeiner & Leander Tentrup (2015): *Detecting Unrealizability of Distributed Fault-tolerant Systems*. *Logical Methods in Computer Science* 11(3), doi:10.2168/LMCS-11(3:12)2015.

[29] Bernd Finkbeiner & Hazem Torfah (2016): *Synthesizing Skeletons for Reactive Systems*, pp. 271–286. Springer International Publishing, Cham, doi:10.1007/978-3-319-46520-3_18.

[30] M. Fujita, Y. Matsunaga & T. Kakuda (1991): *On variable ordering of binary decision diagrams for the application of multi-level logic synthesis*. In: *Proceedings of the European Conference on Design Automation.*, pp. 50–54, doi:10.1109/EDAC.1991.206358.

[31] Yuichi Fukaya & Noriaki Yoshiura (2015): *Extracting Environmental Constraints in Reactive System Specifications*. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Marina L. Gavrilova, Ana Maria Alves Coutinho Rocha, Carmelo Torre, David Taniar & Bernady O. Apduhan, editors: *Computational Science and Its Applications – ICCSA 2015*, Springer International Publishing, Cham, pp. 671–685, doi:10.1007/978-3-319-21410-8_51.

[32] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho & Xudong Zhao (1999): *Coverage Estimation for Symbolic Model Checking*. In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, ACM, New York, NY, USA, pp. 300–305, doi:10.1145/309847.309936.

[33] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Perez, Jean-Francois Raskin, Ocan Sankur & Leander Tentrup (2017): *The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants and Results*. In: *SYNT 2017*, *EPTCS* 260, pp. 116–143, doi:10.4204/EPTCS.260.10.

[34] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer & Roderick Bloem (2007): *Anzu: A Tool for Property Synthesis*. In Werner Damm & Holger Hermanns, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 258–262, doi:10.1007/978-3-540-73368-3_29.

[35] Robert Könighofer, Georg Hofferek & Roderick Bloem (2011): *Debugging Unrealizable Specifications with Model-Based Diagnosis*, pp. 29–45. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-19583-9_8.

[36] R. Knighofer, G. Hofferek & R. Bloem (2009): *Debugging formal specifications using simple counterstrategies*. In: *2009 Formal Methods in Computer-Aided Design*, pp. 152–159, doi:10.1109/FMCAD.2009.5351127.

[37] Wolfgang Lenders & Christel Baier (2005): *Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams*. In Alden H. Wright, Michael D. Vose, Kenneth A. De Jong & Lothar M. Schmitt, editors: *Foundations of Genetic Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–20, doi:10.1007/11513575_1.

[38] W. Li, L. Dworkin & S. A. Seshia (2011): *Mining assumptions for synthesis*. In: *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, pp. 43–50, doi:10.1109/MEMCOD.2011.5970509.

[39] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell P. Marcus & Hadas Kress-Gazit (2015): *Provably correct reactive control from natural language*. *Auton. Robots* 38(1), pp. 89–105, doi:10.1007/s10514-014-9418-8.

[40] P. Nilsson & N. Ozay (2014): *Incremental synthesis of switching protocols via abstraction refinement*. In: *53rd IEEE Conference on Decision and Control*, pp. 6246–6253, doi:10.1109/CDC.2014.7040368.

[41] Hans-Jörg Peter & Robert Mattmüller (2009): *Component-based Abstraction Refinement for Timed Controller Synthesis*. In Theodore Baker, editor: *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), December 1 - December 4, 2009, Washington, D.C., USA*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 364–374, doi:10.1109/RTSS.2009.14.

[42] A. Pnueli & R. Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, SFCS '90, IEEE Computer Society, Washington, DC, USA, pp. 746–757 vol.2, doi:10.1109/FSCS.1990.89597.

[43] V. Raman & H. Kress-Gazit (2013): *Explaining Impossible High-Level Robot Behaviors*. *IEEE Transactions on Robotics* 29(1), pp. 94–104, doi:10.1109/TRO.2012.2214558.

[44] G. Reissig, A. Weber & M. Rungger (2017): *Feedback Refinement Relations for the Synthesis of Symbolic Controllers*. *IEEE Transactions on Automatic Control* 62(4), pp. 1781–1796, doi:10.1109/TAC.2016.2593947.

[45] Viktor Schuppan (2012): *Towards a notion of unsatisfiable and unrealizable cores for LTL*. *Science of Computer Programming* 77(7), pp. 908 – 939, doi:10.1016/j.scico.2010.11.004. Available at `http://www.sciencedirect.com/science/article/pii/S0167642310002030`. (1) FOCLASA09 (2) FSEN09.

[46] Kai Weng Wong, Rüdiger Ehlers & Hadas Kress-Gazit (2014): *Correct High-level Robot Behavior in Environments with Unexpected Events*. In: *Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014*, doi:10.15607/RSS.2014.X.012. Available at `http://www.roboticsproceedings.org/rss10/p12.html`.

[47] Kai Weng Wong & H. Kress-Gazit (2015): *Let's talk: Autonomous conflict resolution for robots carrying out individual high-level tasks in a shared workspace*. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 339–345, doi:10.1109/ICRA.2015.7139021.