Using Reactive Synthesis: An End-to-End Exploratory Case Study

Dor Ma'ayan
Tel Aviv University
Israel

Shahar Maoz
Tel Aviv University
Israel

Fig. 1: Our semester-long workshop class participants specified, synthesized, and executed various reactive controllers using Spectra, such as a patrolling robot (A), a traffic lights system (B), and an autonomous vacuum cleaner (C).

***Abstract*—Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. Despite its attractiveness and major research progress in the past decades, reactive synthesis is still in early-stage and has not gained popularity outside academia. We conducted an exploratory case study in which we followed students in a semester-long university workshop class on their end-to-end use of a reactive synthesizer, from writing the specifications to executing the synthesized controllers. The data we collected includes more than 500 versions of more than 80 specifications, as well as more than 2500 Slack messages, all written by the class participants. Our grounded theory analysis reveals that the use of reactive synthesis has clear benefits for certain tasks and that adequate specification language constructs assist in the specification writing process. However, inherent issues such as unrealizabilty, non-well-separation, the gap of knowledge between the users and the synthesizer, and considerable running times prevent reactive synthesis from fulfilling its promise. Based on our analysis, we propose action items in the directions of language and specification quality, tools for analysis and execution, and process and methodology, all towards making reactive synthesis more applicable for software engineers.**

I. Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [62]. 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. Given the great promise of correct-by-construction systems, much research progress has been achieved over the last two decades on reactive synthesis theory, algorithms, tools, and applications [5], [8], [36], [46], [47]. Despite its many advantages, though, the use of reactive synthesis in real-world settings in the industry is very limited.

When using reactive synthesis, the primary artifact that the engineers write, read, debug, and maintain is a formal declarative specification, not code. Thus, the development process and the activities it includes should be very different than traditional software development processes and activities, and also different from other synthesis techniques such as programming by example, e.g. [29]. We argue that the limited real-world usage of reactive synthesis techniques is partly due to the poor understanding of this development process and the activities it requires. For example, what are the attributes of high-quality specifications? Which parts of a specification better be written before others? What language constructs should a specification language contain? What are the unique challenges developers face when using a reactive synthesizer, and do existing synthesizers address them? Detailed answers to such questions, backed by evidence, can draw a multi-disciplinary roadmap for future research and engineering efforts in the field. For example, future formal methods studies could focus on developing algorithms to support the required analyses, modeling and language studies could explore new abstractions and corresponding language constructs for specification languages, and HCI studies could develop and evaluate new interaction techniques with reactive synthesizers. Hopefully, all these, combined into an appropriate development methodology, will make reactive synthesis technology more applicable for software engineers.

We conducted an exploratory case study of reactive synthesis with 20 participants in a semester-long university workshop class. Throughout the semester, participants

DOI 10.1109/ICSE48619.2023.00071

used Spectra[1][2] [50], a state-of-the-art reactive synthesis specification language and toolset, to complete various specification and simulation tasks. We collected detailed documentation of the entire class's committed specifications, git logs, and all the communication with the participants in Slack, and then analyzed the data using grounded theory. We contribute a description of the process of writing reactive synthesis specifications, as well as the difficulties developers face while writing specifications. Based on our analysis, we identify opportunities and propose action items in the directions of language and specification quality, tools for analysis and execution, and process and methodology, all towards making reactive synthesis more applicable for software engineers.

It is essential to point out that while recent studies identified the need for human-centered design of programming languages [9], [12], [58] and formal methods techniques [37], there was no such study in the context of reactive synthesis. Moreover, there is a large body of work on HCI and synthesis; however, these works are focused mostly on inductive synthesis techniques such as programming by example (PbE) and programming by demonstration (PbD), which are fundamentally different than reactive synthesis from a formal specification. To our knowledge, **we are the first to perform a large, open-ended study of a reactive synthesizer using a complete toolset**. Most previous works in reactive synthesis focused on a specific problem and were evaluated through the narrow lens of examining the performance of a specific new algorithm. In contrast, **our study uses a qualitative methodology to consider the broader context of the end-to-end reactive synthesis development process, from a formal specification to a running system**.

While this paper focuses on the barriers to using reactive synthesis by software engineers who are not necessarily formal methods experts, this is only one of several open problems toward a broader adoption of reactive synthesis tools. Other challenges, such as performance, are actively investigated in the literature [32]. We discuss related work in the next section.

II. Background and related work

Reactive synthesis is the problem of translating a logical specification into an implementation that realizes it. Fig. 2 compares reactive synthesis to traditional software testing or verification. While in traditional development, one prepares a program or a model and then tests or verifies it against a specification, in reactive synthesis, the program or model is automatically generated from a given specification.

Thanks to the attractiveness of correct-by-construction implementations, reactive synthesis has enjoyed extensive academic interest. Pnueli and Rosner were the first to suggest a reactive synthesis algorithm [62] for specifications written in *linear temporal logic* (LTL) [61]. However, reactive synthesis of LTL is double exponential in the length of the formula,

[1]Spectra's website: http://smlab.cs.tau.ac.il/syntech/

[2]Spectra's source code: https://github.com/SpectraSynthesizer

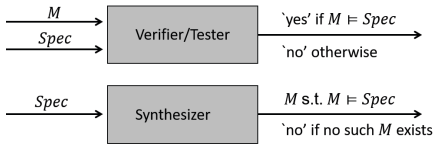

Fig. 2: In traditional system testing and verification, an implementation M is put against a specification (top), while in synthesis, a correct implementation is synthesized from a specification (bottom)

which is not considered practical. GR(1) is a fragment of LTL that has a more efficient synthesis algorithm [6] and is expressive enough for many common properties such as most of the Dwyer et al. patterns [17] as shown in [47]. Therefore, many GR(1) synthesizers were developed in academia, including LTLMop [24], SGR(1) [7], [16], Slugs [18], RATSY [5], and TuLiP [22], [74]. Spectra [50] is a more recent, state-of-the-art reactive synthesizer with many language features and analysis tools that are not (or only partially) supported in all the other mentioned tools.

Beyond synthesis itself, some challenges in reactive synthesis development were identified in the past, and algorithmic efforts were made to address them. These include unrealizability [8], [11], [35], [51], [53], non-well-separation [34], [48], and inherent vacuity [52].

Our work is different from such works since, to the best of our knowledge, **we are the first to inspect how these challenges and others are manifested in their full context of an end-to-end reactive synthesis development process, from the point of view of the developers who are using the synthesizer**, rather than the narrow lens of evaluating the performance of a specific algorithm.

Although both reactive synthesis and formal verification use specifications as their main input, specifications for formal verification are different. Formal verification takes a specification and a model as an input. Reactive synthesis takes only a specification as an input, and the model is generated by the synthesizer. Moreover, not only the development process is different, but the nature of the specifications themselves is different. Specifications for verification can be written and examined property by property. In specifications for synthesis, all properties must be considered together.

Like reactive synthesizers, model finders such as Alloy [31] take a specification as input. However, the specification language and the systems Alloy aims to model are different. As in reactive synthesis, most studies of Alloy focused on new algorithms for specific problems and not on the engineers' point of view during the end-to-end development process of writing an Alloy specification. There is growing awareness of the importance of applying human-computer research methods in such contexts [37], [64], [67]. For example, a work by Danas et al. [15] explored different possible outputs of model finders from a human perspective. We take inspiration from these works and apply similar ideas to reactive synthesis.

Reactive synthesis is a deductive approach, i.e., synthesizing an executable artifact from a complete logical specification.

Other program synthesis techniques have taken an inductive approach, e.g., input-output examples (PbE) [14], demonstrations (PbD) [56], natural-language descriptions, and properties and scenarios [72] as the form of input for the synthesizer. Some of these synthesis approaches enjoyed progress over the years to make them closer to real-world applications [3], [21], [27], [60]. For example, millions of Microsoft Excel users use FlashFill [27], a PbE string transformation system.

Some of the above approaches, such as PbE, were developed around user needs and involved human-centered evaluation [59], [60], [75], [76]. In addition, many user studies have revealed several major usability issues and challenges in these synthesis systems [28], [39], [40], [57], which led to the design of better synthesis systems.

In contrast to these approaches, reactive synthesis requires a complete specification as input and should directly produce a correct-by-construction implementation as output; therefore, it is inherently different from these synthesis techniques and targets different types of systems, computation models, users, and use-cases. It also raises different challenges; for example, while most user studies about PbE discuss challenges with the examples provided to the synthesizer, these do not exist in reactive synthesis, and while the problem of unrealizability is not relevant to PbE, it is very relevant and extensively studied in reactive synthesis. That said, we take inspiration from these works and apply human-centered research methodologies to reactive synthesis.

Papers that discuss the challenges of reactive synthesis as well as some case studies have appeared in the SYNT workshop (since 2014 [10]). For example, Ryzhyk and Walker described their experience and lessons in developing Termite, a reactive synthesizer [65]. Their lessons include the need for specifications reuse, the observation that synthesis does not replace human expertise, and the need for adopting an incremental development approach in order to enable specifications' debugging. Our findings confirm some of these observations. While their lessons are based on their own experience as the developers of the tool, our findings are based on the experience of software engineering students that used our tool.

III. Methodology

This section describes our research methodology, which we designed in light of the ACM SIGSOFT guidelines [63] for case studies[3] and grounded theory[4]. We describe our research setting, the data collection, and the analysis method we used.

A. Research setting: Spectra workshop class

As the main instrument for data collection, we organized a semester-long workshop class in our university.

We chose to conduct the case study with Spectra since it is a state-of-the-art reactive synthesis system and seems to be the readiest system to be used by software engineers. Spectra is a mature, highly-maintained system with over eight years of extensive development with many advancements, all implemented as part of the Spectra ecosystem, in Eclipse plug-ins, most of them in a user-ready state. That said, it is essential to emphasize that in this case study, Spectra is the means and not the end. Our end is to explore and learn about reactive synthesis and not conduct an in-depth evaluation of Spectra as a language and development environment.

All the 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. They also 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 [33]. Moreover, more than 50% of the participants already have student positions in the industry as software engineers. Our participants represent the population target of this study well: professionally educated software engineers with some experience but without prior knowledge or experience in formal methods techniques.

Twenty out of twenty-two students registered to the workshop class signed the consent forms and agreed to participate in this study. As compensation, with approval of the university's IRB, we offered four bonus points for the workshop's final grade for those agreeing to participate in the study.

The class spanned a full semester of 13 weeks. In the first four weeks, we introduced participants to reactive synthesis and most of Spectra's features. Spectra includes many features; some are implemented but not documented at the user level, and some are not ready for real-world use. Thus, as a research design decision, we examined all Spectra features known to us and introduced the participants only to those features that seemed mature and usable enough for real-world context. For example, based on our own experience with Spectra and based on feedback from the authors of the JVTS work [38] and the unrealizability repair work [51], we decided to exclude these two techniques and tools from the current study.

We presented the semantics of Spectra language constructs and complete specifications to the participants, but we did not explain how reactive synthesis works. Specifically, we intentionally did not discuss LTL, GR(1), BDDs, and fixed-point algorithms in class, since we wanted the participants to understand the meaning of Spectra specifications but treat the synthesizer as a "black box". As homework practice during the first four weeks, participants completed the Spectra hands-on online tutorial from ICSE'21[5], which includes nine modules of about 10 minutes each, plus exercises, covering language, analysis, and execution [49].

After the four introductory class meetings, during the remaining nine weeks of the semester, **we asked the participants to complete four development tasks in teams of two**

[3]https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/CaseStudy.md

[4]https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/GroundedTheory.md

[5]https://youtube.com/playlist?list=PLGyeoukah9Nbx1QquUmZGdLulFZIsiRlZ

students each. We will now provide a brief description of the four tasks:

Simple Robot A robot that should travel from target A to target B while avoiding some fixed obstacles.

Patrolling A robot that should patrol between three different random locations while avoiding fixed obstacles. All participants implemented three different variants of this task, with different patrolling behaviors (strict order of visits, non-strict order, and change of colors while visiting targets).

Cleaning Robot An automatic vacuum cleaner that should collect trash from random locations, avoid obstacles, is restricted from visiting certain areas, and also manages its energy consumption.

Junction A controller for a junction's traffic lights system that should safely handle cars and pedestrians coming from four different directions while also dealing with bad weather and prioritizing emergency vehicles.

These development tasks are diverse, in different levels of difficulty, and are inspired, in part, by common specification examples from the literature [2], [55]. Fig. 1 shows some simulation runs of the controllers that the participants synthesized to complete these tasks.

A simulation environment is essential for running the synthesized controllers. Thus, we provided the participants with skeleton implementations of the graphical user interfaces of the simulations. However, importantly, we left the actual simulation behavior, the design decisions on how to use the simulation UI, and the integration with the synthesized controller open to the participants in order to see how different teams will adopt different validation strategies.

The complete protocol of the workshop class with the description of the tasks as given to the participants is in the supplementary material [42], [43].

B. Data collection

We asked participants to document all their work during the semester in several ways. To this end, **we designed a communication mechanism that allowed us to track the participants' progress and receive as much information as possible, using two main platforms to collect various kinds of data**.

Slack is a popular communication platform for software development organizations. We used private Slack channels with each team as the main communication medium. We asked the participants to describe their work there, with a weekly progress report as the minimum and encouraged them to ask any questions about Spectra or the tasks there. We also used the private channels to provide feedback on the team's work and ask follow-up questions about what they wrote. We asked some follow-up questions differentially according to the team's progress and topics that emerged from the discussion. We also asked all the teams some identical follow-up questions to reflect on tasks and on the overall workshop experience. **In total, we collected 2,809 Slack messages with the teams, together with 20 accompanying documents.**

In addition, the participants frequently committed their code to private **GitHub** repositories that we set up. Each team had a private repository, which was not accessible to the other teams. We asked participants to commit frequently with descriptive commit messages. **In total, we collected 599 versions of specifications (i.e., commits to GitHub) for 88 specifications** (for some tasks, some teams implemented and committed several variants). Tbl. I reports statistics for the specifications that the participants committed for the different tasks.

Our analysis covers all the Slack chats, all the commit logs and messages, as well as the specifications committed by the participants. We will now describe our analysis methodology.

C. Analysis methodology

The analysis of our exploratory case study is based on grounded theory as described by Corbin and Strauss [13]. Grounded theory is frequently used for qualitative analysis in software engineering studies [66], [71], [73], with works such as [30], [68] describing best practices in conducting such studies. **We followed these guidelines carefully.**

We started from general research questions and goals, and refined them and the research setting as the study evolved. For example, we started the class with no advance planning of all the tasks we will give; every few weeks, based on the initial analysis of data we collected so far, we decided on the design of the following tasks. We did so when we started to observe saturation regarding specific themes. E.g., when we observed "more of the same" feedback regarding unrealizability and its accompanying analysis tools, we decided to create more diverse tasks that we considered to require advanced language constructs such as triggers and counters.

We analyzed all textual data collected using *open coding* with MAXQDA [6]: we assigned codes to lines, paragraphs, or fragments of the data we collected, and refined the coding system as the study progressed. We started conducting an open coding session on the second week of the class when the first batch of data was obtained. Then, we performed several sessions of *axial coding*, i.e., linked the initial codes under new emerging categories. Finally, using *selective coding*, we identified the high-level themes on which we report in the results section. We used *memos* written during the data collection and analysis process and during research meetings to develop and organize the code system.

Importantly, **while conducting the described analysis, we frequently triangulated our findings from our coding with the actual specifications that participants developed**; in many cases, participants referenced a specific commit so we could triangulate (1) a slack message, (2) a git log, and (3) a relevant version of the specification, in order to draw meaningful conclusions that are grounded both in participants' feedback and in what actually happened. For example, when a Slack comment mentioned the difficulty of interpreting a counter-strategy for a specific unrealizable specification, we identified the corresponding specification, and executed

[6]VERBI Software. MAXQDA 2020. Software. 2019. maxqda.com.

Task	#Lines		#Sys-Vars		#Env-Vars		#Aux-Vars		#Asm		#Guar		#Defines		#Predicates		#PastLTL		#Counters		#Patterns	
	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD	MD	SD
Simple Robot	47.0	23.7	6.0	3.6	0.0	10.9	0.0	1.9	0.0	1.5	6.5	7.9	3.0	6.8	3.0	5.9	0.0	0.0	0.0	0.0	0.0	1.0
Patrol-I	63.5	23.2	8.4	2.8	18.0	0.0	1.6	2.6	5.5	2.7	9.0	3.4	5.8	2.4	4.6	3.1	0.0	0.0	0.0	0.0	0.7	1.3
Patrol-II	110.0	27.3	14.5	4.7	18.0	0.0	5.7	5.3	8.5	3.1	14.0	17.7	5.5	2.3	5.2	3.2	0.7	1.9	0.0	0.9	1.9	2.4
Patrol-III	100.5	29.6	25.9	4.5	18.9	0.3	14.6	5.9	8.5	4.1	18.0	7.4	5.8	2.3	4.8	3.1	0.0	0.0	0.0	0.5	0.8	1.9
Cleaner	181.0	50.9	43.2	9.7	8.9	0.6	24.2	12.4	6.0	1.9	29.0	12.2	8.5	7.2	4.1	2.6	6.2	12.1	4.0	1.8	1.4	2.4
Junction	200.5	69.3	31.0	8.5	23.5	0.9	14.5	8.4	17.0	13.6	30.0	13.7	5.5	28.0	1.0	1.0	0.0	4.0	0.0	0.7	0.0	3.8

TABLE I: General statistics on the specifications committed by the participants, median (MD) and standard deviation (SD)

realizability checking and counter-strategy generation in order to better understand and validate our understanding of the comment.

IV. Results

This section presents the results from our analysis of the data we collected. It is organized according to the high-level themes that emerged from our analysis. Our interpretation of the results and their implications appear in Sect. V.

A. Inherent issues in reactive synthesis

1) When may reactive synthesis be useful?: We received feedback from the participants on which tasks they think are preferable to solve using reactive synthesis rather than more traditional software development. Participants mention that reactive synthesis is preferable when efficiency is less critical than correctness (T4,T10) (Tx marks that team x mentioned it), i.e., when one would prefer a mathematically correct solution over a fine-grained control on the efficiency of the solution. Participants also note that reactive synthesis is preferable for systems with considerable dependency between the controller behavior and the environment (T1,T2,T4,T5). E.g., T5 based their claim on a comparison between different workshop tasks:

> "*...the junction task was the most suitable task for Spectra programming... in the junction task, the system was highly dependent on the environment, and almost all guarantees involved env variables which dictated the system's behavior (e.g., a traffic light can turn green only if there is a car waiting). On the contrary, in all other tasks, the system behavior was much less dependent on the environment, and we think that regular programming would have been much easier to write and test.*" **(T5)**[7]

Some participants compared reactive synthesis to traditional software development and mentioned its great expressive power when used wisely:

> "*...this can turn out to be very useful on a project that demands the use of machines and other objects that operate in a pre-defined space and has a unique and well-defined mission to fulfill. The benefit of writing 20-30 lines of code that express the equivalent of hundreds of lines of code in other languages is very useful*" **(T10)**

2) Specification development requires a mindset shift from traditional software development: Reactive specifications' expressiveness and usefulness in describing complex systems require a mindset shift compared to traditional software development since writing formal specifications for synthesis is a considerably different task from traditional programming. All the workshop class participants mentioned this difference, for example:

> "*...concentrating on what we want to achieve is not enough. Moreover, sometimes we do not even need to specify what we want to achieve, but we must be very accurate in describing what we do not want to happen. This forces a new way of thinking, a more complex one, kind of out of the box*" **(T2)**

Some participants considered the different mindset that reactive synthesis requires to be beneficial. Some participants mentioned that in reactive synthesis development, thinking about the correctness of the desired outcome is an inherent part of the development process (T1,T2,T4,T8) and, therefore, forces users to "see the bigger picture and think like a product owner" right at the beginning (T1). They considered it to be different from traditional software development, in which one may develop a program that will work for some cases without thinking beforehand on its correctness in all cases:

> "*...when you write code, you may find yourself in a situation where the code runs, and it completely breaks. In spectra, much pre-design thought must be made, and it results in a better outcome right at the beginning.*" **(T10)**

Many participants struggled with this mindset shift, as reflected in their frequent Slack progress updates and questions. They mentioned that there are many ways to logically model a system's desired behavior. As evidence, see the high variance in Tbl. I in the number of system variables, assumptions, and guarantees to describe a given problem; these many possible ways to model a problem might turn out to be challenging:

> "*...(in reactive synthesis)...you arrive at a certain point where you realize the way you chose to model the problem is wrong and then must practically start all over again. In other software programming environments, you usually don't have to re-think about the basic choices you've taken at the beginning and usually can make a quick fix (sometimes "ugly" solution that works), while here these cases are more complex – still possible from time to time, but we do feel the language "forces" you to do things right and not just find a quick solution to your problem.*" **(T10)**

3) Unrealizability is a severe and frequent problem: Unrealizability is a well-studied topic in reactive synthesis, with multiple proposed solutions. However, no study so far observed the user aspects of unrealizability and the influence on the development process of specifications. Our findings confirm that unrealizability is an acute and frequent problem that drastically influences the development process of specifications. For example, 12% of the specifications committed

[7]We quote all participants comments without editing them, e.g., we do not correct grammatical mistakes.

by the participants were unrealizable, and all teams frequently mentioned unrealizability problems in their Slack conversations and git logs, reaching a total of 242 different occurrences (meaning that about 10% of all discussions were around unrealizability issues participants faced).

Participants often found the realizability of the specifications they write to be fragile (T2,T4,T5,T7,T9), meaning that even a small change in a single statement may turn a realizable specification into an unrealizable one, for example:

> "*We added a guarantee to make sure the counter does not change unless the robot is cleaning or it is in zero state. This was required as, without it, the tank counter kept changing during the whole run. Unfortunately, adding this guarantee made our spec unrealizable, and we couldn't solve it.*" **(T9)**

To handle this sensitivity of specifications, some participants adopted an iterative development approach: they added new logic to the specifications carefully and in small steps to make sure that they remain realizable after each new addition (T4,T5,T7,T10):

> "*. . . this way, if the specification was realizable before, but it is not realizable now, we knew that there was something wrong with the guarantee that we just added.*" **(T5)**

Moreover, they used Spectra analysis tools iteratively, in a "TDD" fashion to detect and fix unrealizability problems:

> "*This week we worked in a similar to "TDD" method – we added new guarantee to specification (regarding freeze mode or manual mode for example) and used realizability check as a "test" which is "failing" if the specification is not realizable. If so – we used unrealizable core calculation to understand the problem and "fix" the specification accordingly.*" **(T7)**

We will further discuss the usability of these tools in the supporting features and tools subsection of the results below.

4) Non-well-separation: Non-well-separated specifications are specifications in which the synthesizer may solve the synthesis problem by creating a controller that can force environment assumption violations [34], [48].

Non-well-separation issues were mentioned extensively by many participants (T2,T3,T4,T5,T7,T10). However, they were less frequent than unrealizability. In contrast to unrealizability, which means that a controller could not be created, non-well-separated specifications can be synthesized into a controller, and so in many cases, participants found the problem only after they observed the controller behavior for some time during its executions:

> "*We found out we had lots of bugs we didn't know about, and our specification didn't work. It was realizable, but when we ran it, non of the guarantees were met because of strange behavior. . . . maybe our spec is not well-separated. Indeed that was the case.*" **(T2)**

Careful inspection of the actual environment implementation in the simulation code was found to be helpful in order to fix non-well-separation issues (T5,T10):

> "*. . . After reviewing again the Java code we wrote, we found the problem: we expected that when the robot reaches origin, the engine problem is fixed. This is a logically false assumption because the environment cannot predict that the next state of robot (which is sys variable) will be at origin and thus turn off the engine problem (which is env variable) before reaching there. It made us understand that when the robot reaches origin, only in the next state the engine problem should go off and not in the current. Fixing it made our program well separated and work as we expected.*" **(T5)**

5) The synthesizer as a "black-box": For this study, we intentionally chose participants who represent the knowledge of a typical software engineer: one with a strong background in writing code and familiarity with basic CS concepts but without deep knowledge in formal methods and synthesis. We found that the lack of deep knowledge on how the synthesizer works influenced the ability of participants to develop specifications easily. For example, participants mentioned that the development process becomes harder since they could not estimate the synthesis or analysis time of their specification; more specifically, they mentioned that they do not have the tools or knowledge to understand what parts of their specification make the synthesis inefficient (T1,T4,T9,T10):

> "*We feel that many times we've made a simple change of the spec (adding a counter instead of a pattern, for example), and the running time became too large . . . if it's possible to evaluate the complexity of each construct and add an estimated computation time may help a lot during the process of writing a spec.*" **(T10)**

Other than efficiency, the lack of knowledge on how the synthesizer works influenced participants' ability to debug and fully understand specific problems since they do not have the tools to understand the problem deeply. As a result, sometimes, they just changed specifications in a "trial-and-error" fashion until they found a solution that worked for their needs (T2,T7):

> "*. . . we went on the "obvious" solution - triggers. Surprisingly, the specification became not realizable. It was the strangest thing because there seems to be no reason for the robot not to stay there or wait for the green light. The core was also not helpful. We tried a few versions of a trigger expression, and then again decided to use a counter instead. This time, we don't know yet what the problem was. But changing to a counter and a boolean variable start (meaning we are at the start of the program and are still in our first 8 stages), made the problem disappear. So of course we kept it that way.*" **(T2)**

B. Supporting features and tools

1) Existing analysis techniques for unrealizabilty and non-well-separation do not scale: Participants used two kinds of solutions for unrealizability and non-well-separation issues: detection of cores [48], [53], an strategies. Due to the high frequency of these issues, participants used these methods extensively and iteratively and, in many cases, found them helpful in understanding the source of the problem (T2,T4,T7,T9,T10), for example:

> "*After writing those guarantees we checked realizability and it was not realizable. The core in this case, helped us*

immediately to find the first problem – if there is alw fog so we can't guarantee that cars will eventually drive. Hence, we add an assumption that alwEv there is no fog. Then, still the spec was unrealizable but this time with another problem. Here also the core helped us a lot to find the contradiction – if there is alw road constructions so we can't guarantee that cars from south will drive eventually straight. Hence, we add an assumption that alwEv there is no road construction. On the third try, still we got Unrealizability and again the core helped us to understand that if there is alw fog and then road constructions so again we can't guarantee that cars from the south will drive eventually straight..." **(T2)**

However, in many other cases, participants found cores and strategies to be less helpful due to (1) long computation times, especially since participants used these tools very frequently as part of their development process (T2,T4,T6,T8,T9,T10), and (2) difficulty to interpret the core or the strategy due to a large number of states or statements (T4,T5,T8,T10).

For example, T10 described a counter-strategy with more than 200 states and a non-well-separation strategy with over 150 states which were practically impossible to follow:

"*...we calculated a counter-strategy with over 200 states, so really trying to understand what happens beyond the "metadata" is difficult, and in such cases, we just prefer to change tactics...*" **(T10)**

"*The (non-well-separation) strategy spectra found had over 150 states so it was quite impossible to follow.*" **(T10)**

And T4 described an unrealizable core with 14 guarantees, which takes more than 30 min to compute:

"*...we get a core of 14 statements all over the spec. Because spec has become quite big now, each... check takes at least half an hour. As a result, debugging difficulty has increased tenfold. Beforehand, we attributed part of these checks assistance in debugging to responsiveness: Meaning, even if seeing an unrealizable core doesn't explicitly tell you the problem, it directs you to what statements you need to focus on. Then, you can try and alter these statements, and if it's still unrealizable, you can again ask to see an unrealizable core, and so on. But now, this iterative debugging approach is not applicable. It feels like the tools... are rendered close to impracticable whenever the spec grows to a project of this size. Knowing that a check we need will freeze our work for a half-hour at least forces us to rely less on the checks.*" **(T4)**

As a result, T4 also describes their manual solution to deal with unrealizability, which was also in use by other teams:

"*...you then can gray-out each paragraph and add them back incrementally (in different orders) in order to recognize what statements bring about a conflict.*" **(T4)**

Previous research has reported that unrealizable cores tend to include about 18% of the specification guarantees [53]. Our results show how challenging it is to understand large cores.

2) On simulations, exploration, and correct-by-construction: The central promise of reactive synthesis is to provide a correct-by-construction implementation of controllers. These controllers interact with an environment that satisfies the specification's assumptions. Therefore, a central part of the development process of specifications is to explore their behavior while reacting to environment inputs. Spectra offers two ways to explore the controller's behavior: (1) A *controller-walker*, an exploration and debugging mechanism that implements capabilities similar to a traditional debugger, such as breakpoints and a step-by-step execution (Fig. 3). (2) A *Java API* to feed the controller with the next environment input and get the next controller output.

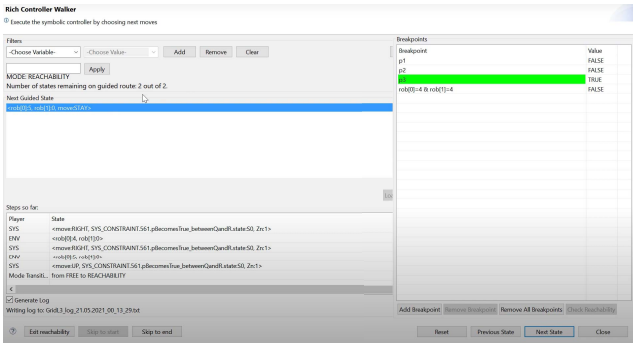

Fig. 3: The rich controller walker is a Spectra debugging tool with similar capabilities to traditional software debugging tools, e.g., breakpoints and step-by-step execution. Our findings show that in terms of user-interface usability, this traditional debugging approach, in its present design, does not scale to specifications over many variables.

Several teams used the walker to explore different behaviors of their synthesized controllers (T2,T4,T5,T7):

"*...we used the rich walker a lot for our testings. We first thought about all possible cases (i.e., moving inside the orange zone, moving from the orange zone to the white zone, and so on). Then, we used the 'walk as both players' option to control the simulation completely. We tested all edge cases we could think of by using this tool. If a problem occurred, we headed back to the spectra code and looked for the logic that caused this problem, and fixed it. We repeated this process over and over again until the desired behavior was reached.*" **(T5)**

However, the walker turned out to be helpful only for small specifications; in particular, it was not valuable and intuitive for specifications with many variables:

"*...the rich walker is a great tool for debugging, but when there are too many variables, it makes it too hard to use.*" **(T5)**

Due to the limited usefulness of the controller-walker, all participants used the Java API as the main way to validate and explore the behavior induced by their specifications.

Perhaps surprisingly, despite the correct-by-construction promise, all participants reported that they found bugs in their specifications, although they were realizable, well-separated, and initially considered correct by their developers. That is, the implementation was correct w.r.t. the

specification, but the specification did not correctly express the intended requirements. For example, participants frequently had commit messages like: "*... is realizable & well separated, but still not behaving as expected.*" **(T2)**

Others mentioned in their Slack messages some unexpected behaviors of a "correct" specification:

"*We noticed we had a mistake in our freeze mode implementation (we used asms when we were supposed to use gars). The mistake didn't affect the realizability and the well-separation of the spec, so it wasn't easy to notice that these lines are problematic...*" **(T8)**

During the workshop class, participants had the freedom to validate the correctness of their specifications in any way they found appropriate. The most common validation strategy was implementing some kind of a random simulation that mocks up possible environment behaviors according to the participant's understanding of the problem. Note that this is not easy, because in order to be useful for simulation, the implemented environment behavior must satisfy the assumptions written in the specification. When the environment violates an assumption, the Spectra controller executor outputs a relevant warning to the console and there is no promise that the remainder of the execution will satisfy the guarantees.

Given the random environment they implemented, participants executed it against the synthesized controller and observed the resulting behaviors. However, some of the teams understood that such validation is insufficient since some occurrences of events and edge cases are rare:

"*... 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 that, participants used additional strategies. For example, T10 tweaked the environment's randomness in such a way that 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 they wanted to test.

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

C. Specification language and modeling

Since writing formal specifications using pure linear temporal logic is challenging, higher-level languages with predefined abstraction mechanisms were developed. Spectra contains two kinds of such language constructs (beyond its kernel, see [50]) Basic constructs include non-temporal language features that are pure syntactic sugars, i.e., arrays, defines, and predicates. Advanced constructs include temporal language features such as Past LTL operators [26], counters, "triggers [4], [19], and most Dwyer et al. patterns [17], [47]. These are implemented by automatic encoding into the synthesis problem, using added auxiliary variables. Complete documentation of all the Spectra language constructs can be found in the language documentation, available from the Spectra website.

1) System vs. Environment: We found that some participants struggled to distinguish between the roles of the system and the environment in the synthesis problem, especially in the first few weeks of the semester, although Spectra explicitly distinguishes between system and environment variables and between system guarantees and environment assumptions (T7,T8,T9,T10). This difficulty manifested at two levels. First, at the modeling level, i.e., the distinction between variables that should fall under the system responsibility and ones that should fall under environmental responsibility:

"*At the beginning, we didn't manage to differentiate good enough between the system variables and the environment variables... we thought over again about the modeling of the problem and chose to model the obstacles as the environment and the robot as the system.*" **(T10)**

Second, at the properties level, i.e., the distinction which properties should be specified as environment assumptions and which should be specified as system guarantees:

"*Sometimes, the environment violates the assumption on the first step. The robot visits targets as part of the patrol while there is an engine problem and does not return to (0,0) "immediately". After lots of trial and error, we realized that the trigger should be a gar rather than an assumption. We changed it, and the spec worked well. The previous problems are now fixed.*" **(T7)**

"*We noticed we had a mistake... we used asms when we were supposed to use gars...*" **(T8)**

2) Use of advanced language constructs: Table I summarizes the usage statistics of some of these language constructs. We can see extensive usage in basic and advanced language constructs, which serves as evidence for their importance as an abstraction level above LTL and pure GR(1).

Moreover, despite the advanced language constructs available to them beyond the kernel of `ini`, `alw`, and `alwEv` assumptions and guarantees, participants did not consider the language to be complex. For example:

"*The simplicity of the language – the fact that the language is simple to learn and has a limited number of constructs and keywords that allow you to express very large variety of specs is a huge advantage for spectra.*" **(T10)**

3) Challenges with advanced language constructs: Besides the benefits of using advanced language constructs, participants faced some challenges in using them, many of which relate to theme 4.1.5 (the synthesizer as a "black-box"). For example, participants often found the Dwyer et al. patterns, as available in the Spectra patterns library, to be unclear:

"*The fact we can use patterns to simplify a complicated function seems to be very useful, but to be honest, we didn't really manage to understand those patterns deeply. We think it might be helpful to give the patterns more intuitive names*

or to dedicate some time (in class) to go over the patterns and give examples of the use of each one of them." **(T2)**

As another example, some participants struggled with the exact semantics of counters and patterns, although we presented them in class, with examples, and provided what we considered to be meaningful documentation:

"*We also found out that sometimes it's not simple to understand the meaning of prev / current / next states when dealing with counters (meaning – if we say: when counter reaches x do something – when will it be done? On the next state? On the current state?). The problem is that if we don't time it correctly, the environment can suffer a violation of the assumptions.*" **(T10)**

"*. . . for example does P becomes true after Q means that it becomes true for one state, or is it forever, will it become true on the next or current state*" **(T7)**

Since participants had trouble to fully understand the exact meaning of these language constructs, in some cases they abandoned using them in favor of other constructs (T5,T9):

"*. . . we haven't used patterns. We tried using them a couple of times, it always led to unrealizable results, and we've always eventually found a different solution.*" **(T5)**

"*We have tried to understand why did the first pattern we used didn't work but we still don't understand the reason. To find a better solution we went through the Spectra deck and once we saw the "ONCE" operator we immediately understood that this is the right operator to resolve this issue. This was actually intuitive. . .* " **(T9)**

V. Discussion and future research action items

Our results describe unique strengths and weaknesses of reactive synthesis. With appropriate solutions to some of the open challenges in the field, reactive synthesis may become an important component in the software engineer's future arsenal of tools. We envision reactive synthesis used by software engineers who are not familiar with how formal methods work but are familiar with what they mean and how to use them; our workshop class outcomes indicate the applicability of this goal. The work is a step toward our larger vision to make reactive synthesis more usable and practical.

This section identifies opportunities and proposes action items for making reactive synthesis more natural and applicable for software engineers. We base these action items on the results described in Sect. IV.

A. Language and specification quality

Our results reflect the role of language constructs as a layer of abstraction between the user intentions and the logic.

While some language constructs do a great job in hiding the specification's complexity, we have indications that language constructs that are "too abstract" or hide too much of the complexity of specifications might achieve the opposite goal and make understanding and debugging of the specification more challenging. Therefore, **more work on language constructs translating the real engineer's intentions into a formal input for the synthesizer is needed**. Patterns, such as [17], [55] may be a good starting point for such future abstraction mechanisms, but as our results show, there is still a place for improvement in **bringing the accurate meaning of a pattern to a user while keeping its implementation abstract**, for example, by better usage examples of patterns, or advanced visualizations that will help in understanding their semantics.

Moreover, research on specification languages could be **inspired by current and past research efforts in program comprehension**, as published in the ICPC conference series [1]. Such works should be done with careful attention to task selection and experiment design, in order to avoid potential bias [20].

Despite the similarity between specification and code, there are significant differences. While program comprehension cognitive models take into account the flow of programs [69], in specifications, this flow does not exist, and there is no meaning to the order of declarations. Littman et al. [41] defined program comprehension as knowing the objects the program manipulates and the actions it performs, as well as its functional components and the causal interactions between them. Although similar, specifications have unique properties and language constructs that distinguish them enough from traditional imperative code and therefore prevent direct adoption of concepts. In particular, the tasks of locating a program code line that implements a given command, explaining the semantics of a given code block, or manipulating a given code snippet in order to change its meaning, are very different than the corresponding tasks of locating an assumption or guarantee that specifies a given property, explaining the meaning of a given guarantee or manipulating it in order to change its meaning based on new requirements. One reason for these differences may be that in imperative programming, the developers specify what should be done, as opposed to reactive synthesis specifications, in which the developers should also explicitly identify and specify what should not be done. Another difference is that in specifications, developers also explicitly specify assumptions about the behavior of the environment; this makes specification development and reading significantly different from development and reading of code.

Empirical research on the comprehension of specifications (e.g., what makes elements in a specification easy or difficult to comprehend?) could lead to **new means to measure specifications' complexity and quality** (e.g., nesting depth of Boolean operators in assumptions and guarantees, and more generally, the definition and detection of anti-patterns [44]). These may be followed by means for improvement, for example, **proposing refactoring techniques (e.g., split guarantee)**, warnings, and tools to promote and enforce writing conventions (e.g., should all assumptions be separated from all guarantees or should related assumptions and guarantees, say, ones that use a similar set of variables, appear close to each other in the specification document?).

A potentially significant outcome of research efforts on the comprehension of specifications can be translated to a better design of specification languages, perhaps inspired by the work

on natural programming [12], [58], a PL and HCI design methodology that is aimed to "make it possible for people to express their ideas in the same way they think about them".

B. Tools for analyses and execution

We observed that participants used specification analysis extensively. First, we noticed that the available tools and techniques for addressing unrealizability and non-well-separation issues deal with relevant and frequent problems; this shows that previous research efforts indeed targeted problems that exist in practice (although they were not necessarily based on empirical evidence). However, our findings show that **current tools output is in many cases unclear to the engineers**, which often makes it unusable on a large scale.

To better translate the specification analysis tools' output for user comprehension, **better representations** are required. For example, it may be possible to consider visualization and interactive techniques or textual explanations to replace or to extend the current output of counter-strategies and cores; **it may be the case that although they are incomplete, concrete assignments and a selected path to a dead end would be more helpful to engineers in explaining unrealizability than existing techniques.** All these should be developed and evaluated with the target software engineers in mind.

In addition, our observation that in many cases, **specifications that are realizable, well-separated, and considered correct by their developers are discovered to be incorrect after synthesis**, suggests the need for better exploration and debugging tools. Exploration tools such as the rich controller walker, and the combinatorial coverage (CC) algorithm for scenario-suite generation [45] implemented in Spectra serve as a good starting point; however, they still **suffer from scalability problems in both running times and in clarity of user interface (when the specification has "too many" variables).** Future work should explore better user interfaces for this task. For example, create semi-automated abstractions that will present the user only with the relevant information for the current execution state, e.g., by temporarily hiding irrelevant variables.

C. Process and methodologies

Our results indicate that **writing specifications for reactive synthesis raises different challenges and involves different activities compared to traditional programming**. Therefore, methodologies and activities that seem straightforward and are well-studied for traditional software development are not necessarily applicable when developing specifications. While the existence of these differences by itself may be well-known, the evidence and discussion of these differences in activities may lead to the design of better tools and methodologies.

First, the field could benefit from the **exploration of overall development methodologies**. One example of such a methodology could be an incremental approach in which in each iteration the engineer adds a small set of guarantees, makes the specification realizable again by adding assumptions, and only then continues to add another small set of guarantees (a similar incremental methodology was suggested in [65]). An alternative approach could be to start by writing all the assumptions in order to establish non-well-separation early and only then add all the guarantees. Each approach may have its strengths and weaknesses. Different approaches may fit different development setups.

Second, one could **identify the activities involved by exploring a development process for specifications**. Examples of possible activities that we already identified in this study are formulating and distinguishing between the system and environment-controlled variables of the synthesis problem, checking whether an added guarantee turns a specification from realizable to unrealizable, checking the non-well-separation of a complete environment model, or step-by-step simulation of the synthesized controller over a scenario of interest. Another example activity is **structured specification review, inspired by code review practices** (what are the qualities one should examine during a specification review and what tools can assist in performing the review?). Systematically identifying these activities and their further exploration will help to focus research efforts around the specific activities involved in the development process, and inform the specific needs and the development of new tools and techniques.

D. Performance

Despite advancements in heuristics to improve the performance of reactive synthesis and related analyses [25], **considerable running times are still a significant bottleneck** towards broad adoption of reactive synthesis.

Performance issues are not new in the context of reactive synthesis, and multiple solutions have been proposed over the years. For example, just-in-time reactive synthesis [54], giving up on completeness with bounded synthesis [23], and pure symbolic analysis of counter-strategies for unrealizable specifications [38]. Our results call for further research into these methods and new ones, including, e.g., modularity or different means of parametrization.

Identifying common development activities as proposed in Sect. V-C may open up opportunities for designing heuristics that take advantage of the unique context of specific activities, for example, the proposed iterative methodology of adding one guarantee at a time and not adding new ones until the specification becomes realizable again, calls for an **incremental unrealizability analysis techniques that build on the results of previous analyses**.

Regardless of the efforts to find better heuristics and synthesis algorithms, the frequent usage of advanced language constructs (backed by the relatively high number of auxiliary variables in Tbl. I) and combined with the inherent lack of transparency of many advanced language features, due to abstraction, may hint that **at least in some cases, using advanced language constructs may be a form of accidental rather than essential complexity**, or otherwise hint that developers do not use variables cost-effectively. Future studies may explore these phenomena and propose ways to detect such cases and address them using refactorings or indicative

warnings for developers. Moreover, as stated by participants, even general warnings regarding expected high running times could be helpful in this context.

VI. Threats, Evaluation, and Quality Criteria

This section discusses different evaluation criteria for our study and potential threats to its validity.

Construct validity. (the appropriateness of operational measures for the concepts being studied) We used multiple sources of data (e.g., Slack communications, git commit messages, specifications, and follow-up questions) to establish a clear chain of evidence to our findings. By triangulating the data from these sources, we reduced each source's weaknesses and got a broader picture of the process.

Internal validity. (the establishment of causal relationships) We addressed internal validity in the data analysis phase, by strictly following the Straussian methodology [13] regarding the coding phases and analysis.

Reliability. (the degree to which the research method produces stable and consistent results) We ensured reliability as follows:

- We make the case study protocol and collected specifications available as part of a replication package (see [42], [43]), to maximize *replicability* and *transparency*.
- We used a saturation strategy: we kept exploring certain aspects of reactive synthesis and designed our tasks accordingly until we observed saturation in findings and then moved to more tasks that examined more aspects.

External validity. (how well can the outcome of a study be expected to apply and be generalized to other settings) We have two concerns that fall under this category:

- **Target population.** We chose a population that represents future users of reactive synthesis well. Therefore, we believe that the results generalize well in this context. While using students may introduce some potential threats, other than the fact that they are all from the same university, participants took different classes, and many already have a strong industrial background in diverse companies; this creates a diverse background of participants, even more than an industrial setting within a small team in a single company. The participants are not formal methods experts but more representative of an average professional software engineer. Our experiment design decision to use students who represent professional software engineers and not formal methods experts was intentional.
- **Main study platform.** The exploratory study was conducted using Spectra, state-of-the-art in reactive synthesis environment. It is possible that using Spectra and not another reactive synthesizer affected the results somehow. However, while analyzing the data, we put special emphasis on themes that generalize well to other existing and future reactive synthesis platforms.

Other threats. The class participants worked in pairs so our analysis was done in team granularity, treating each team as one entity. As a result, we have no feedback from individuals on their own overall experience. Yet, we observed saturation in our findings from different teams, which reduces the threat. In addition, the study was carried out as part of a university class, and students were graded for their work. To encourage students to be active and provide us feedback, our course policies emphasized the importance of active participation and detailed documentation of the process as the main grading criteria for the class. That said, all the students who signed the consent form received the 4 bonus points on their grade, regardless of their performance in the class tasks.

VII. Conclusion

Reactive synthesis is a promising direction towards integrating correct-by-construction methods into the software development process. However, most previous research efforts on reactive synthesis had focused on technical contributions. **This work is the first to qualitatively explore an end-to-end reactive synthesis development process from the perspective of developers, collect rich data of different kinds, and use it to identify current strengths, challenges, and opportunities for improvement**. Based on these, we have proposed **action items** towards making reactive synthesis more applicable for software engineers, from the **specification language** and **analysis tools** to the **development process methodologies and activities**.

It is important to note that our findings and action items might not directly generalize beyond reactive synthesis to other formal methods applications. That said, the questions that emerged from our findings may inspire corresponding research questions in similar domains. For example, the limited usefulness of cores for debugging and explaining problems in specifications may inspire similar studies on the use of cores in tools such as Alloy [70]. As another example, the need for better mechanisms to present the accurate meaning of patterns to users may be applicable to model-checking tools. Every such potential generalization requires a specific study in the relevant context.

Finally, following our present study, future research efforts include a focus on specification quality and comprehension, on user-centered solutions for problems in specifications, and, more broadly, on the identification and development of the methodologies, activities, and tools that can serve as building blocks for an effective end-to-end reactive synthesis development process.

Acknowledgements

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

References

[1] Table of contents. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 5–10, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

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

[3] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.

[4] G. Amram, D. Ma'ayan, S. Maoz, O. Pistiner, and J. O. Ringert. Triggers for reactive synthesis specifications. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[5] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY – A New Requirements Analysis Tool with Synthesis. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, pages 425–429, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012. In Commemoration of Amir Pnueli.

[7] V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel. Controller synthesis: From modelling to enactment. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 1347–1350. IEEE Press, 2013.

[8] D. G. Cavezza, D. Alrajeh, and A. György. Minimal assumptions refinement for realizable specifications. In K. Bae, D. Bianculli, S. Gnesi, and N. Plat, editors, *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020*, pages 66–76. ACM, 2020.

[9] S. E. Chasins, E. L. Glassman, and J. Sunshine. PL and HCI: Better Together. *Commun. ACM*, 64(8):98–106, jul 2021.

[10] K. Chatterjee, R. Ehlers, and S. Jha, editors. *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014*, volume 157 of *EPTCS*, 2014.

[11] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 52–67. Springer, 2008.

[12] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers. Pliers: A process that integrates user-centered methods into programming language design. *ACM Trans. Comput.-Hum. Interact.*, 28(4), jul 2021.

[13] J. Corbin and A. Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[14] A. Cypher. Eager: Programming repetitive tasks by example. In *Readings in human–computer interaction*, pages 804–810. Elsevier, 1995.

[15] N. Danas, T. Nelson, L. Harrison, S. Krishnamurthi, and D. J. Dougherty. User studies of principled model finder output. In A. Cimatti and M. Sirjani, editors, *Software Engineering and Formal Methods*, pages 168–184, Cham, 2017. Springer International Publishing.

[16] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models for fallible domains. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 211–220, New York, NY, USA, 2011. Association for Computing Machinery.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[18] R. Ehlers and V. Raman. Slugs: Extensible GR(1) Synthesis. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 333–339. Springer International Publishing, 2016.

[19] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006.

[20] D. G. Feitelson. Considerations and pitfalls in controlled experiments on code comprehension. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 106–117. IEEE, 2021.

[21] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 229–239, New York, NY, USA, 2015. Association for Computing Machinery.

[22] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray. Control design for hybrid systems with TuLiP: The Temporal Logic Planning toolbox. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*, pages 1030–1041, 2016.

[23] B. Finkbeiner and S. Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5–6):519–539, oct 2013.

[24] C. Finucane, G. Jing, and H. Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1988–1993, 2010.

[25] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for GR(1) synthesis and related algorithms. *Acta informatica*, 57(1):37—79, 2020.

[26] D. Gabbay. The declarative past and imperative future. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 409–448, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[27] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, jan 2011.

[28] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[29] S. Gulwani and P. Jain. Programming by examples: PL meets ML. In B. E. Chang, editor, *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, volume 10695 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

[30] R. Hoda. Socio-technical grounded theory for software engineering. *IEEE Trans. Software Eng.*, 48(10):3808–3832, 2022.

[31] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on software engineering and methodology (TOSEM)*, 11(2):256–290, 2002.

[32] S. Jacobs, G. A. Perez, R. Abraham, V. Bruyere, M. Cadilhac, M. Colange, C. Delfosse, T. van Dijk, A. Duret-Lutz, P. Faymonville, et al. The Reactive Synthesis Competition (SYNTCOMP): 2018-2021. *arXiv preprint arXiv:2206.00251*, 2022.

[33] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[34] U. Klein and A. Pnueli. Revisiting synthesis of GR(1) specifications. In S. Barner, I. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing*, pages 161–181, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[35] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *International journal on software tools for technology transfer*, 15(5):563–583, 2013.

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

[37] S. Krishnamurthi and T. Nelson. The human in formal methods. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 3–10, Cham, 2019. Springer International Publishing.

[38] A. Kuvent, S. Maoz, and J. O. Ringert. A symbolic justice violations transition system for unrealizable GR(1) specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 362–372, New York, NY, USA, 2017. Association for Computing Machinery.

[39] T. Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Mag.*, 30(4):65–67, 2009.

[40] T. Y. Lee, C. Dugan, and B. B. Bederson. Towards understanding human mistakes of programming by example: An online user study. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*, IUI '17, page 257–261, New York, NY, USA, 2017. Association for Computing Machinery.

[41] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, 1987.

[42] D. Ma'ayan and S. Maoz. Supporting artifact, 2023. https://doi.org/10.5281/zenodo.7566397.

[43] D. Ma'ayan and S. Maoz. Supporting materials website, 2023. https://smlab.cs.tau.ac.il/syntech/exploratory/.

[44] D. Ma'ayan, S. Maoz, and J. O. Ringert. Anti-patterns (smells) in temporal specifications. In *IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2023.

[45] D. Ma'ayan, S. Maoz, and R. Rozi. Validating the correctness of reactive systems specifications through systematic exploration. In E. Syriani, H. A. Sahraoui, N. Bencomo, and M. Wimmer, editors, *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*, pages 132–142. ACM, 2022.

[46] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. Reactive high-level behavior synthesis for an atlas humanoid robot. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4192–4199, 2016.

[47] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 96–106, New York, NY, USA, 2015. Association for Computing Machinery.

[48] S. Maoz and J. O. Ringert. On well-separation of GR(1) specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 362–372, New York, NY, USA, 2016. Association for Computing Machinery.

[49] S. Maoz and J. O. Ringert. Reactive Synthesis with Spectra: A Tutorial. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 320–321, 2021.

[50] S. Maoz and J. O. Ringert. Spectra: a specification language for reactive systems. *Softw. Syst. Model.*, 20(5):1553–1586, 2021.

[51] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for GR(1) specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1016–1026, 2019.

[52] S. Maoz and R. Shalom. Inherent vacuity for GR(1) specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 99–110, New York, NY, USA, 2020. Association for Computing Machinery.

[53] S. Maoz and R. Shalom. Unrealizable cores for reactive systems specifications. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, page 25–36. IEEE Press, 2021.

[54] S. Maoz and I. Shevrin. Just-in-time reactive synthesis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, page 635–646, New York, NY, USA, 2020. Association for Computing Machinery.

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

[56] B. A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.*, 12(2):143–177, apr 1990.

[57] B. A. Myers and R. McDaniel. Chapter 3 - demonstrational interfaces: Sometimes you need a little intelligence, sometimes you need a lot. In H. Lieberman, editor, *Your Wish is My Command*, Interactive Technologies, pages 45–III. Morgan Kaufmann, San Francisco, 2001.

[58] B. A. Myers, J. F. Pane, and A. J. Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, sep 2004.

[59] H. Peleg and N. Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. *Leibniz international proceedings in informatics*, 166, 2020.

[60] H. Peleg, S. Shoham, and E. Yahav. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1114–1124, 2018.

[61] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[62] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA, 1989. Association for Computing Machinery.

[63] P. Ralph, N. b. Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, et al. Empirical standards for software engineering research. *arXiv:2010.03525*, 2020.

[64] A. Reid, L. Church, S. Flur, S. de Haas, M. Johnson, and B. Laurie. Towards making formal methods normal: meeting developers where they are. *arXiv preprint arXiv:2010.16345*, 2020.

[65] L. Ryzhyk and A. Walker. Developing a practical reactive synthesis tool: Experience and lessons learned. In R. Piskac and R. Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, volume 229 of *EPTCS*, pages 84–99, 2016.

[66] T. Sedano, P. Ralph, and C. Péraire. Software development waste. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 130–140. IEEE Press, 2017.

[67] A. Siegel, M. Santomauro, T. Dyer, T. Nelson, and S. Krishnamurthi. Prototyping formal methods tools: A protocol analysis case study. In *Protocols, Strands, and Logic*, pages 394–413. Springer, 2021.

[68] K.-J. Stol, P. Ralph, and B. Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery.

[69] M. D. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Softw. Qual. J.*, 14(3):187–208, 2006.

[70] E. Torlak, F. S. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[71] C. Treude and M.-A. Storey. Effective communication of software development knowledge through community portals. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 91–101, New York, NY, USA, 2011. Association for Computing Machinery.

[72] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.

[73] M. Waterman, J. Noble, and G. Allan. How Much Up-Front? A Grounded theory of Agile Architecture. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 347–357, 2015.

[74] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.

[75] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. L. Glassman. Interpretable program synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[76] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman. *Interactive Program Synthesis by Augmented Examples*, page 627–648. Association for Computing Machinery, New York, NY, USA, 2020.