On the Software Engineering Challenges of Applying Reactive Synthesis to Robotics

Shahar Maoz
School of Computer Science, Tel Aviv University, Israel

Jan Oliver Ringert
Department of Informatics, University of Leicester, UK

ABSTRACT

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification. This short paper discusses the software engineering challenges in applying reactive synthesis to robotics, beyond the synthesis algorithms themselves, including the challenge of writing declarative specifications, the challenge of abstraction of data and time, and the challenge of availability of an adequate development process supported by related tools. The identification and description of the challenges are based on our experience in building a development environment for reactive synthesis and applying it to the construction of about 10 different autonomous Lego robots. We describe the challenges using concrete examples from one of the robots built in our lab.

CCS CONCEPTS

• **Software and its engineering** → *Formal methods*; *Software verification*;

KEYWORDS

reactive synthesis, GR(1)

ACM Reference Format:
Shahar Maoz and Jan Oliver Ringert. 2018. On the Software Engineering Challenges of Applying Reactive Synthesis to Robotics. In *RoSE'18: IEEE/ACM 1st International Workshop on Robotics Software Engineering , May 28-June 28 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3196558.3196561

1 INTRODUCTION

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [26]. Rather than manually constructing an implementation and using model checking to verify it against a specification, synthesis offers an approach where a correct implementation of the system is automatically obtained for a given specification, if such an implementation exists. In the case of reactive synthesis, an implementation is typically given as a controller, i.e., an automaton that accepts input from the environment (e.g., from sensors) and produces the system's output (e.g., commands for actuators) to always satisfy the specification.

Reactive synthesis has been studied extensively in the formal methods and software engineering research literature, see, e.g., [1–3, 6, 13, 15, 19, 21, 23]. While synthesis from Linear Temporal Logic (LTL) specifications [26] is generally considered impractical due its high computational complexity (double exponential in the length of the formula), synthesis from fragments of LTL, e.g., General Reactivity of Rank 1 (GR(1)) [3], has shown to allow for efficient sybolic implementations of synthesis algorithms.

One potential application domain for reactive synthesis is robotics. Indeed, in the last few years, a number of research groups have investigated this direction, providing initial results. Kress-Gazit et al. [14, 16, 30] applied reactive synthesis for robotics mission planning, where a synthesized controller guarantees a robot achieves a given task. Maniatopoulos et al. [17] present an approach to automatically generate code implementation of high-level robotic behavior using synthesis. They demonstrated their approach on the Atlas humanoid robot in the context of the 2015 DARPA Robotics Challenge. Wongpiromsarn et al. [31] use GR(1) synthesis in TuLiP, a software toolbox for synthesis of embedded control software with application to robotics mission planning. Gritzner and Greenyer [10] have investigated the use of reactive synthesis to generate PLC code for industrial robots. We have applied reactive synthesis in a case study for synthesizing a forklift controller [20] (used here as our running example, see Sect. 2). Furthermore, various synthesis tools have been developed independent of the robotics domain, e.g., Bloem et al. [2] developed RATSY, a requirement analysis tool with GR(1) synthesis and Ehlers and Raman [6] developed the GR(1) synthesis framework Slugs, which has been extended with a number of plugins for advanced analyses.

While these show some potential for future success, a wide chasm remains to be crossed before reactive synthesis becomes a tool in the hands of robotics software engineers.

In this short paper we discuss the key software engineering challenges in applying reactive synthesis to robotics, beyond the synthesis algorithms themselves. First, the challenge of writing declarative specifications. Second, the challenge of abstraction of data and time. Finally, the challenge of availability of an adequate development process supported by related tools.

We demonstrate these challenges via an example robot, one of many built in our lab at Tel Aviv University, where we have taught two project classes (in 2015 and in 2017), in which small teams of 3rd year Computer Science undergraduate students have used our synthesis environment in a semester long project, to develop about 10 different autonomous Lego robots, which the students actually built and run[1]. From these classes, we have collected over 200 versions of specifications, all written by these students. Our

[1] A short video is available in https://youtu.be/JTTP3L9G6ko

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RoSE'18, May 28-June 28 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5760-9/18/05…$15.00
https://doi.org/10.1145/3196558.3196561

choice of Lego as the underlying robotics technology for the class and for this research was motivated by its relatively low cost and its modularity, allowing us to gain experience from several different robots, e.g., a robot sorting Lego pieces by color, an elevator servicing different floors, a self parking Lego car, performing very different tasks. Furthermore, although different in detail and scale, the use of the Lego robots provided us with concrete examples of some of the challenges one expects to encounter with real-world robotic technologies, such as the inaccuracy of sensor readings, and the limitations in terms of battery, memory, and computation power.

2 RUNNING EXAMPLE

We start off with a running example of a specification for an autonomous forklift. The forklift is an actual Lego robot[2] shown on the left side of Fig. 1. We have constructed and experimented with this robot in our lab (see [20]).

2.1 Forklift Overview and Architecture

The forklift has a sensor to determine whether it is at a station, two distance sensors to detect obstacles and cargo, and an emergency button to stop it. It has two motors to turn the left and right wheels and one motor to lift the fork. Consider an initial set of informal requirements for the behavior of the forklift:

(1) Do not run into obstacles.
(2) Only pick up or drop cargo at stations.
(3) Always keep on delivering cargo.
(4) Never drop cargo at the station where it was picked up.
(5) Stop moving if emergency off is pressed.

The logical software architecture of the forklift is depicted as a component and connector model (see e.g., [28] and [11], for concepts of component and connector models) in Fig. 1 (b). The components on the left side are hardware wrappers that read sensor values and publish them as messages on their output ports. The output ports of the sensor components are connected to input ports of component `Controller` (names and types labeled on ports in Fig. 1). The output ports of component `Controller` (names and types labeled on ports in Fig. 1) are connected to three components on the right that receive commands and encapsulate access to the actuators (here different motors) of the forklift.

The execution of the robot is performed in a control cycle: read sensor data, execute controller, perform actions. The only delays during execution of cycles are introduced by computation times of components (in ranges of milliseconds).

We focus on the development of component `Controller`. This component is the most complicated in terms of reactive behavior as it contains the systems logic of how actuators are controlled to realize desired system behavior.

2.2 Example Specification

We show excerpts of a specification for the forklift robot in Lst. 1. The specification is written in the Spectra language [22]. From the complete specification we can directly synthesize an implementation of component `Controller` (from Fig. 1). The specification starts by declaring environment controlled variables (inputs of `Controller`) and system controlled variables (outputs of `Controller`).

We introduce some short names for expressions in a `define` section in ll. 15-23. As an example, we define `stopping` as an abbreviation for both motors receiving the command `STOP` and we define `deliverUnlessBlocked` as an abbreviation for dropping cargo or being blocked by the emergency off button or a low obstacle.

The main part of the specification are assumptions on the behavior of the environment and guarantees that a system implementation has to satisfy if all the assumptions hold. Assumptions and guarantees that start with the temporal operator **`G`** have to hold on all transitions between states, e.g., the assumption `stationsDontMove` in ll. 26-27 expresses that in all states where the forklift is stopping the value read by the station sensor does not change in the next state. Assumptions and guarantees with the temporal operator **`GF`** must be satisfied repeatedly after any finite number of transitions, e.g., the guarantee `keepDelivering` in ll. 48-49 states that the forklift has to repeatedly deliver cargo unless it is repeatedly blocked.

We have added the five informal requirements from above as comments to guarantees of the specification that most closely resemble each requirement. The complete specification consists of 9 assumptions and 12 guarantees and it takes one second to synthesize an implementation from it.

3 CHALLENGES

We now list the three challenges that we have identified. For each, we first explain the challenge in general terms, then we illustrate it on examples relating to the forklift robot, and finally we mention existing works and how they relate to the presented challenge.

3.1 Writing declarative specifications

A major software engineering challenge for applying reactive synthesis is writing of declarative specifications. In a robotics software engineering process based on synthesis specifications replace code. Software components that are traditionally implemented manually are now synthesized automatically. However, shifting the focus on declarative specifications brings various challenges. First, there is currently little experience and almost no training for writing declarative specifications. Second, specification languages from the formal methods community, e.g., Linear Temporal Logic (LTL) [25] (which can be seen as the assembly language for synthesis), are not friendly to be used by engineers and specifying reasonable constraints on the robot's behavior may lead to very complex formulas. Finally, declarative specifications for non-trivial robot behavior require engineers to make assumptions on the behavior of the environment the robot operates in explicit. Inadequate assumptions and guarantees can lead to no or even undesired implementations of robot behavior.

Example 1. When executing synthesized controllers from earlier versions of the specification shown in Lst. 1 the forklift destroyed the physical lifting mechanism because a synthesized controller issued the lifting command when the robot had already lifted cargo.

[2] Robot based on these building instructions: http://www.nxtprograms.com/NXT2/forklift/steps.html

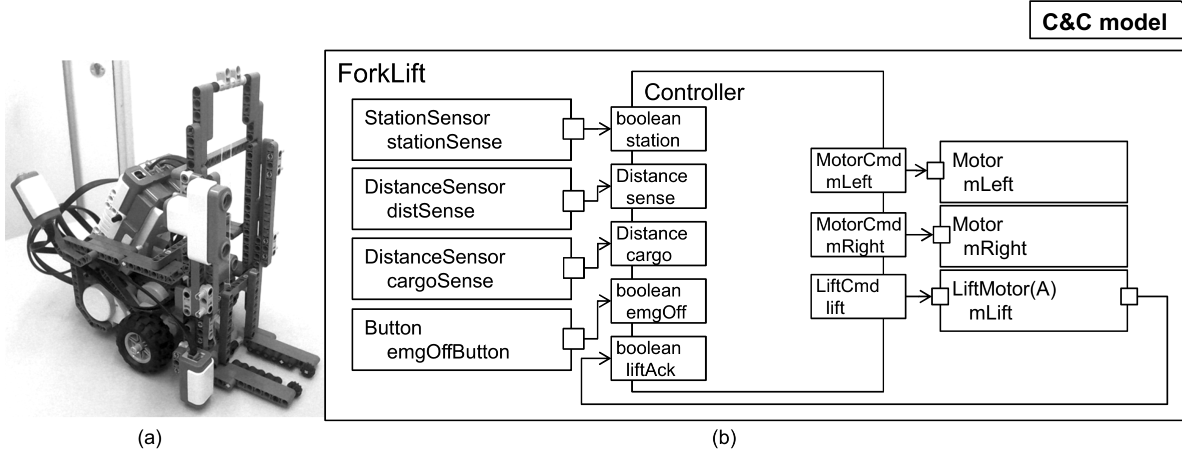

Figure 1: (a) The Lego forklift robot with four sensors and three actuators. (b) The component and connector model of the software architecture of the robot with wrappers for sensors and actuators. An implementation of the main component `Controller` will be synthesized

While an engineer writing imperative code would intuitively understand that over-lifting the fork is dangerous, in the declarative specification we forgot to formulate a constraint expressing this knowledge. The synthesizer thus had no way to know of the physical system's limitations. Later, to formulate the corresponding constraint, we have added the monitor `loaded` to keep track of the state of the fork and the guarantee `noOverlifting` (Lst. 1, ll. 69-70) to prevent commands that can damage the mechanism.

Example 2. Specifications describe temporal relations between system states. These relations easily become complex to describe in a declarative way. As an example, recall requirement (4) to never drop cargo at the same station that it was picked up. We formulate as a the temporal constraint that the between lifting cargo and dropping cargo the forklift has to leave the station. More technically, to express that a predicate p = `!atStation` always has to be satisfied at least once between the satisfaction of predicates q = `lifting` and r = `dropping` we could write the following LTL formula:

$$\mathbf{G}(q \texttt{ \& !}r \texttt{ -> (!}r \ \mathbf{W}\ (p \texttt{ \& !}r\texttt{)))}$$

Intuitively, $\varphi \ \mathbf{W}\ \psi$ expresses that either φ holds forever or φ holds in every state until ψ holds in at least one state. Note that the temporal operator **W** is nested in the scope of the operator **G** and the predicate r is repeated multiple times. These LTL formulas can easily become unreadable, see e.g., the LTL specification patterns of Dwyer et al. [5] collected from existing specifications. In our example specification in Lst. 1 we use pattern names and instantiations [19], e.g., in guarantee `leaveStationForDelivery` (Lst. 1, ll. 52-54), instead of their LTL equivalent.

Example 3. An important part of a specification are assumptions on environment behavior. Note that our initial list of five informal requirements formulated only guarantees. Any non-trivial reactive behavior specification will require assumptions that promise that the environment will react to the behavior of the system, e.g., in order to ensure that packages will be delivered only on stations, the forklift will assume that if it stops on a station the station will not magically disappear (assumption `stationsDontMove`, Lst. 1, ll. 26-27). In addition, we might want to assume that the forklift always eventually finds a station, i.e., **GF** `station`. However, the combination of these two assumptions might lead to a system implementation that forces the environment to violate an assumption: the robot can stop forever, once it is not on a station and the environment will then fail to satisfy the assumption **GF** `station`.

Works Related to this Challenge. Most recent works in the area of applying reactive synthesis are indirectly related to the challenge of writing declarative specifications. In works by Kress-Gazit et al. [14, 16, 30] the specifications are written with primitives specific to path planning on a grid; or can be extracted using natural language processing [16]. In [17] specifications for synthesis are generated from high-level models created by engineers. Fillippidis et al. [8] present a multi-paradigm specification language that mixes declarative and imperative elements. We have recently [19] shown how to support most of the well-known LTL specification patterns of Dwyer et al. [5] as specification elements for GR(1) synthesis.

3.2 Abstraction of data and time

A second major software engineering challenge for applying reactive synthesis to robotics system development is the necessity for abstraction of complex data and time. While this abstraction is a general challenge for robotics software development, properties of the underlying formalisms of reactive synthesis add additional complexity. First, the running time of a synthesis algorithm typically depends on the statespace — including inputs and outputs — of the specified system. As an example, the popular GR(1) synthesis algorithm [3], used in many synthesis tools, has complexity quadratic in the statespace where the statespace is exponential in the numbers of input and output variables. Thus, abstraction of data in the specification is often needed to make synthesis tractable. Second, the temporal relations expressed in specifications refer to

```
type Distance = {FAR, CLOSE};
type MotorCmd = {FWD, STOP, BWD};
type LiftCmd = {LIFT, DROP, NIL};

env boolean station;
env Distance sense;
env Distance cargo;
env boolean emgOff;
env boolean liftAck;

sys MotorCmd mLeft;
sys MotorCmd mRight;
sys LiftCmd lift;

define
  stopping := mLeft = STOP & mRight = STOP;
  backing := mLeft = BWD & mRight = BWD;
  turning := mLeft != mRight;
  forwarding := mLeft = FWD & mRight = FWD;
  dropping := lift = DROP;
  lifting := lift = LIFT;
  lowObstacle := (cargo = CLOSE & !station);
  deliverUnlessBlocked := (dropping | emgOff | lowObstacle);

// station does not change when stopping
asm stationsDontMove:
  G (stopping -> station = next(station));

// driving away from an obstacle will eventually clear
// the sensors unless the forklift stops or goes straight
asm driveAwayFromCargoAndObstacles:
  pRespondsToSUnlessT(backing | turning, // =s, trigger
                      sense=FAR & cargo=FAR, // =p, response
                      forwarding | stopping); // =t, unless

// (1) Do not run into obstacles
gar dontHitObstacles:
  G ((sense = CLOSE | lowObstacle) -> ! forwarding);

gar noTurningCloseToCargo:
  G (cargo=CLOSE -> !turning);

// (2) Only pick up or drop cargo at stations.
gar liftDropAtStationOnly:
  G ((lifting | dropping) -> atStation);

// (3) Always keep on delivering cargo.
gar keepDelivering:
  GF deliverUnlessBlocked;

// (4) Never drop cargo at the station where it was picked up.
gar leaveStationForDelivery:
  // p=!atStation between q=lifting and r=dropping
  pBecomesTrue_betweenQandR(!station, lifting, dropping);

// (5) Stop moving if emergency off is pressed.
gar emergencyOff:
  G (emgOff -> (stopping & lift=NIL));

// monitor loaded is true iff we cargo loaded & acknowledged
monitor boolean loaded {
  !loaded;
  G (liftAck & !loaded -> next(loaded));
  G (liftAck & loaded -> next(!loaded));
  G (!liftAck ->  loaded = next(loaded));
}

// no lifting when loaded and no dropping when not loaded
gar noOverlifting:
  G ((loaded -> !lifting) & (!loaded -> !dropping));
```

Listing 1: Excerpts from a specification of the reactive behavior of component `Controller` of the forklift robot shown in Fig. 1. A complete version of the specification is available from http://smlab.cs.tau.ac.il/syntech/forklift/**. See [22] for a language reference of the Spectra language used in this listing.**

a sequence of states and not to points in the physical time a robot operates in. Thus, an engineer has to find suitable mechanisms and abstractions to relate physical time to a virtual sequence of states.

Example 4. The forklift robot shown in Fig. 1 is equipped with distance sensors that measure distances up to 255cm with a resolution of 1cm. Including many input variables with large ranges in a specification could render traditional synthesis algorithms impractical. When developing the specification shown in Lst. 1 we have introduced an abstraction of values smaller than 20cm to `CLOSE` and values greater or equal to 20cm to `FAR`. The abstraction is performed by component `DistanceSensor` and the environment variables in the specification in Lst. 1 only use the type `Distance = FAR, CLOSE`, e.g., for variable `sense` in guarantee `dontHitObstacles` (Lst. 1, ll. 37-38).

Example 5. Consider the scenario where the forklift delivers cargo. Intuitively we would want to express that the forklift needs to back up before it can turn to drive away from cargo (otherwise the fork might not be clear from cargo). In the forklift specification temporal relations are over states and execution steps but it is unrealistic to assume that backing up for one execution step will clear the fork from cargo. Specifications have to consider physical processes that take time not measured in execution steps. To address this challenge, we added the guarantee `noTurningCloseToCargo` (Lst. 1, ll. 40-41) and the assumption `driveAwayFromCargoAndObstacles` (Lst. 1, ll. 31-34). The assumption expresses that after some finite time of backing up or turning, cargo and obstacles will no longer block the sensors. The synthesizer will determine that backing up to clear cargo is the only viable option to continue.

Works Related to this Challenge. Recent works have started to address challenges of abstraction in different application domains of reactive synthesis. Murray et al. [7, 31] have presented Tulip for synthesizing hybrid reactive systems where piecewise affine dynamics are automatically abstracted. Walker and Ryzhyk [29] presented predicate abstraction and refinement for reactive synthesis of device drivers where large memory registers are automatically abstracted. Raman et al. [27] formalized a pattern for supporting the execution of robot actions with arbitrary timing for reactive synthesis (our solution in Example 5 is similar to their approach).

3.3 Development process and tools

A third major challenge for applying reactive synthesis to robotics concerns the development process and the related tool support. Although the synthesized implementation is correct with regard to the specification, the specification itself may not correctly represent the intentions of the engineer who wrote it. Moreover, specifications, like code, evolve over time, as bugs are corrected and new features are added. Thus, only having a synthesis tool that provides correct-by-construction implementation of the specification does not suffice. Effective use of reactive synthesis implies major changes to traditional development processes and revised developer tasks, which in turn call for new tool support.

Example 6. A specification may be over constrained, in which case it is unrealizable, i.e., there exists an environment that meets the specified assumptions and can force the system to violate some

of the specified guarantees. For example, consider the following guarantee, expressing the requirement to go forward when no obstacle is detected:

```
G (!lowObstacle -> forwarding)
```

This guarantee, together with guarantee `emergencyOff` (Lst. 1, ll. 57-58), overconstrain possible behaviors when there is no obstacle and the emergency off is pressed. The specification would be unrealizable, and so no implementation will be synthesized.

Example 7. The problem described in Example 3 above is a case of non-well-separation [12], where a synthesized controller may satisfy the specification by forcing any environment to violate the assumptions (rather than by satisfying the guarantees). From a development process perspective, this problem may be worse than the above mentioned problem of unrealizability: since the synthesizer is able to output a controller, there is no hint of the problem, which will only appear later, during testing, or worse, in production.

Example 8. An earlier version of the forklift did not include an emergency off feature. To reflect the new requirement in the specification, we added the guarantee `emergencyOff` (Lst. 1, ll. 57-58). Note that this single guarantee affects the behavior of a synthesized controller in almost all of its states. Throughout development of a specification, many similar changes are expected.

Works Related to this Challenge. Some recent works have started to address a few aspects of this challenge. For example, to deal with unrealizable specifications, authors have shown how to compute an unrealizable core [4, 13], i.e., a minimal subset of the guarantees for which the specification is already unrealizable. The core is intended to help the engineer by localizing the problem. Other works suggested also the computation and presentation of a counter-strategy [13, 23, 24], which shows how a synthesized environment can force any system implementation to violate the specification. We have recently presented the JVTS, a symbolic counter-strategy representation, which is typically much smaller, simpler, and can be computed faster than concrete counter-strategies [15].

To deal with non-well-separation, we have recently presented a tool to automatically identify different cases of non-well-separation and provide the engineer with a non-well-separation core, i.e., a minimal subset of the assumptions that is already non-well-separated [21].

To the best of our knowledge, no work has yet addressed other aspects of the development process, such as dealing with the evolution of specifications and testing of synthesized controllers.

4 CONCLUSION

In this short paper we have discussed three major software engineering challenges for the application of reactive synthesis to robotics. We illustrated the challenges using 8 concrete examples, taken from a specification of a Lego forklift built in our lab. We hope this work will encourage discussion and provide directions for future work in this field.

The work is part of a larger project[3] on bridging the gap between the theory and algorithms of reactive synthesis on the one hand and software engineering practice on the other. As part of this project we are building engineer-friendly tools for writing and understanding temporal specifications for reactive synthesis (see, e.g., [9, 15, 18–21]).

[3]SYNTECH: http://smlab.cs.tau.ac.il/syntech/

ACKNOWLEDGEMENTS

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

REFERENCES

[1] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2013. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*. IEEE, 26–33. http://dx.doi.org/10.1109/FMCAD.2013.6679387

[2] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. 2010. RATSY - A New Requirements Analysis Tool with Synthesis. In *CAV (LNCS)*, Vol. 6174. Springer, 425–429. https://doi.org/10.1007/978-3-642-14295-6_37

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

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

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

[6] Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) Synthesis. In *CAV (LNCS)*, Vol. 9780. Springer, 333–339. https://doi.org/10.1007/978-3-319-41540-6_18

[7] Ioannis Filippidis, Sumanth Dathathri, Scott C. Livingston, Necmiye Ozay, and Richard M. Murray. 2016. 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*. IEEE, 1030–1041. https://doi.org/10.1109/CCA.2016.7587949

[8] Ioannis Filippidis, Richard M. Murray, and Gerard J. Holzmann. 2015. A multi-paradigm language for reactive synthesis. In *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015. (EPTCS)*, Pavol Cerný, Viktor Kuncak, and Parthasarathy Madhusudan (Eds.), Vol. 202. 73–97. https://doi.org/10.4204/EPTCS.202.6

[9] Elizabeth Firman, Shahar Maoz, and Jan Oliver Ringert. 2017. Performance Heuristics for GR(1) Synthesis and Related Algorithms. *CoRR* abs/1712.01103 (2017). arXiv:1712.01103 http://arxiv.org/abs/1712.01103

[10] Daniel Gritzner and Joel Greenyer. 2017. Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications. In *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Martina Seidl and Steffen Zschaler (Eds.), Vol. 10748. Springer, 247–262. https://doi.org/10.1007/978-3-319-74730-9_23

[11] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems.* Technical Report AIB-2012-03. RWTH Aachen. http://aib.informatik.rwth-aachen.de/2012/2012-03.pdf

[12] Uri Klein and Amir Pnueli. 2010. Revisiting Synthesis of GR(1) Specifications. In *Haifa Verification Conference (HVC) (LNCS)*, Vol. 6504. Springer, 161–181. https://doi.org/10.1007/978-3-642-19583-9_16

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

[14] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Trans. Robotics* 25, 6 (2009), 1370–1381. https://doi.org/10.1109/TRO.2009.2030225

[15] Aviv Kuvent, Shahar Maoz, and Jan Oliver Ringert. 2017. 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, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 362–372. https://doi.org/10.1145/3106237.3106240

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

[17] Spyros Maniatopoulos, Philipp Schillinger, Vitchyr Pong, David C. Conner, and Hadas Kress-Gazit. 2016. Reactive high-level behavior synthesis for an Atlas humanoid robot. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, Danica Kragic, Antonio Bicchi, and Alessandro De Luca (Eds.). IEEE, 4192–4199. https://doi.org/10.1109/ICRA.2016.7487613

[18] Shahar Maoz, Or Pistiner, and Jan Oliver Ringert. 2016. Symbolic BDD and ADD Algorithms for Energy Games. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. (EPTCS)*, Ruzica Piskac and Rayna Dimitrova (Eds.), Vol. 229. 35–54. https://doi.org/10.4204/EPTCS.229.5

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

[20] Shahar Maoz and Jan Oliver Ringert. 2015. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015 (EPTCS)*, Vol. 202. 58–72. https://doi.org/10.4204/EPTCS.202.5

[21] Shahar Maoz and Jan Oliver Ringert. 2016. On well-separation of GR(1) specifications. In *FSE*. ACM, 362–372. https://doi.org/10.1145/2950290.2950300

[22] Shahar Maoz and Jan Oliver Ringert. 2018. Spectra Language and Spectra Tools User Guide. Available from http://smlab.cs.tau.ac.il/syntech/spectra/. (2018). version: March 2018.

[23] Shahar Maoz and Yaniv Sa'ar. 2013. Counter play-out: executing unrealizable scenario-based specifications. In *ICSE*. IEEE / ACM, 242–251. http://dl.acm.org/citation.cfm?id=2486821

[24] Shahar Maoz and Yaniv Sa'ar. 2013. Two-Way Traceability and Conflict Debugging for AspectLTL Programs. *T. Aspect-Oriented Software Development* 10 (2013), 39–72. https://doi.org/10.1007/978-3-642-36964-3_2

[25] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

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

[27] Vasumathi Raman, Nir Piterman, and Hadas Kress-Gazit. 2013. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*. IEEE, 4075–4081. https://doi.org/10.1109/ICRA.2013.6631152

[28] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice.* Wiley.

[29] Adam Walker and Leonid Ryzhyk. 2014. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 219–226. https://doi.org/10.1109/FMCAD.2014.6987617

[30] Kai Weng Wong, Cameron Finucane, and Hadas Kress-Gazit. 2013. Provably-correct robot control with LTLMoP, OMPL and ROS. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*. IEEE, 2073. https://doi.org/10.1109/IROS.2013.6696636

[31] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M. Murray. 2011. 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)*. ACM, New York, NY, USA, 313–314. https://doi.org/10.1145/1967701.1967747