Slugs: Extensible GR(1) Synthesis

Rüdiger Ehlers[1(✉)] and Vasumathi Raman[2]

[1] University of Bremen and DFKI GmbH, Bremen, Germany
ruediger.ehlers@uni-bremen.de
[2] United Technologies Research Center, Berkeley, CA, USA

Abstract. Applying reactive synthesis in practice often requires modifications of the synthesis algorithm in order to obtain useful implementations. We present `slugs`, a generalized reactivity(1) synthesis tool that has a powerful plugin architecture for modifying any aspect of the synthesis process to fit the application. `Slugs` comes pre-equipped with a variety of plugins that improve the quality of the synthesized solutions along criteria such as quick response, cost-optimality, and error-resilience. We demonstrate the utility and scalability of the tool on an example from robotics.

1 Introduction

Reactive synthesis automates the task of developing correct-by-construction finite-state machines: rather than writing an implementation and a specification for verifying the system, the engineer need only devise the specification, and the implementation is computed automatically. Of the many synthesis approaches available to the practitioner, *generalized reactivity(1) synthesis* [1], which is commonly abbreviated as *GR(1) synthesis*, has found widespread use for applications in robotics and control. Reasons for this success include its comparatively low, singly-exponential time complexity, and its amenability to symbolic computation using binary decision diagrams (BDDs).

The basic idea behind reactive synthesis is to capture *all* of the requirements of the desired implementation in the specification, and to then accept *any* implementation that satisfies the requirements. On a theoretical level, this is a compelling premise: if the obtained implementation is not good enough, the engineer can simply add additional requirements until it is. However, on a practical level, this approach is problematic: in many cases, system properties such as "quick response" or "few states" cannot be captured precisely in the specification without resorting to synthesis with a cost or payoff function. Introducing costs leads to a higher computational complexity, loss of ability to efficiently use BDDs as a computational data structure, and unfavourable theoretical properties, such as suboptimality of finite memory solutions. Having several optimization criteria in the specification can also create undecidable synthesis problems. Finally, some optimization criteria cannot be expressed quantitatively. Examples include minimizing the time spent waiting for environment fairness conditions

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 333–339, 2016.
DOI: 10.1007/978-3-319-41540-6_18

(i.e. environment actions that can be assumed to be performed infinitely often) to hold, and cooperation with the environment on preserving the environment assumptions.

All these arguments advocate for a different approach to practical reactive synthesis: rather than encoding every qualitative requirement for the synthesized controller into the specification, why not adapt the synthesis algorithm itself to compute implementations that have the properties needed for practical applications? The simplicity of generalized reactivity(1) synthesis makes it particularly suitable as a starting point for demonstrating this approach to synthesis. Modifications of the standard GR(1) algorithm for synthesizing eager and cost-optimal implementations [2] or cooperative implementations [3] that still support symbolic computation have already been proposed in the past, along with semantic modifications for robotics applications [4] and techniques for debugging support [5].

The presented tool `slugs` offers a framework for GR(1) synthesis and its modifications. It has a small simple core implementation of the GR(1) synthesis algorithm that can be extended by user-written plugins. The architecture of `slugs` allows one to use multiple plugins at the same time, where each plugin only modifies a part of the synthesis process. A focus of the tool lies on *conciseness* and *readability* of the code, to make it easier for algorithms to be adapted for specific application domains. For example the realizability check on a specification, which amounts to evaluating the main fixpoint formula from [1], takes only 23 lines of code, and yet is very readable. The `slugs` synthesis tool comes with a specification debugger, using which the cause for realizability and unrealizability of a specification can be determined in an interactive fashion. The tool is written in C++ and is available under the permissive MIT open source license.

This paper is structured as follows: in the next section, we describe the particular view of GR(1) synthesis that `slugs` takes. Section 3 then provides an overview of `slugs`'s architecture and the available plugins. Finally, Sect. 4 demonstrates `slugs`'s performance on an example specification.

2 GR(1) Synthesis

Synthesis of reactive systems has been identified to have a high computational complexity for many specification logics. For generalized reactivity(1) specifications, the synthesis problem has a complexity that is only exponential in the number of atomic propositions in the specification (or polynomial in the size of the the state space of the *game structure* built in the synthesis process). Given sets of input positions $\mathcal{I}$ and $\mathcal{O}$, specifications in this fragment of linear temporal logic (LTL) are of the form

$$(\varphi_i^a \wedge \varphi_s^a \wedge \varphi_l^a) \rightarrow (\varphi_i^g \wedge \varphi_s^g \wedge \varphi_l^g),$$

where φ_i^a, φ_s^a, and φ_l^a are called the *assumptions*, and φ_i^g, φ_s^g, and φ_l^g are called the *guarantees* of the specification. The assumptions are used to state what we

know about the behavior of the environment in which the synthesized system is intended to operate, whereas the guarantees contain the properties that the synthesized system needs to satisfy if the environment behaves as expected. In GR(1) specifications, all assumptions and guarantees must have a certain shape. The *initialization assumptions* φ_i^a and φ_i^g must be free of temporal operators and state valid variable valuations for $\mathcal{I}$ and $\mathcal{O}$ when the synthesized controller starts to operate. The safety properties φ_s^a and φ_s^g state how the proposition valuations for $\mathcal{I}$ and $\mathcal{O}$ can evolve during a step of the synthesized controller's execution. The *liveness properties* φ_l^a and φ_l^g state which transitions of $(\mathcal{I} \cup \mathcal{O})$'s valuations are supposed to happen infinitely often.

Synthesis from generalized reactivity(1) specifications is often reduced to solving a *fixpoint* equation on a *game structure* that is built from the specification. The transitions in the game structure are given by the safety assumptions and guarantees, and the liveness properties are translated to environment and system *goals*, which the system and environment player try to satisfy infinitely often in a play of the game, respectively. In contrast to [1], we use a modified fixpoint equation with only a single occurrence of the *enforceable predecessor operator* EnfPre to compute from which positions the *system player* can win the game:

$$W = \nu Z. \bigwedge_{j=1}^{n} \mu Y. \bigvee_{i=1}^{m} \nu X. \mathsf{EnfPre}((\varphi_{l,j}^g \wedge Z') \vee Y' \vee (\neg \varphi_{l,i}^a \wedge X'))$$

Here, ν is the *greatest fixpoint operator* whereas μ is the *least fixpoint operator*, while the number of liveness assumptions and guarantees are m and n, respectively. The EnfPre operator takes as input a set of *transitions* (the corresponding operator in [1] takes a set of *states*), and computes the set of positions of the game from which the system player can ensure that, after the next valuations to $\mathcal{I}$ and $\mathcal{O}$ have been selected by the environment and system players, respectively, the resulting transition is in the set of given transitions. The specification formula only explicitly mentions the liveness assumptions and guarantees of the specification, as the safety constraints are encoded in the game structure, and the initialization constraints only need to be considered after computing the set of winning positions in the game, W. If for every first environment player move, the system player can ensure that the resulting position satisfies $\varphi_i^a \rightarrow \varphi_i^g$ and is in W, then there exists an implementation for the specification, and it can be extracted from the sequence of transitions given to EnfPre during the evaluation of the least fixpoint, after all the greatest fixpoint operators have been fully evaluated.

The modified fixpoint formula makes it easier to alter the synthesis algorithm, as it channels the possible actions of the implementation to be synthesized through a single invocation of the EnfPre operator. Restricting or extending this set of actions thus amounts to simply adding or removing transitions from the operand of EnfPre. Also, the modified fixpoint formula makes the aims of the system player more explicit: in every step of the system's execution, the system should either reach a *system goal* (which need to be reached infinitely often for

the liveness guarantees to hold), get closer to the system goal, or wait for some environment goal to be reached: this last option is only available until the current environment goal has been reached. Except in very simple specifications, it is commonly not under the control of the system which of these cases holds. The system must, however, ensure that at least one of them holds at every point in time. The conjunctions and disjunctions over i and j make sure that all liveness assumptions and guarantees are considered in order.

The presented tool `slugs` does not build the game structure explicitly, but rather uses binary decision diagrams (BDDs) as symbolic data structure. As the synthesis games have all valuations of $\mathcal{I} \cup \mathcal{O}$ as positions, position sets and transitions relations can be represented efficiently as BDDs with $|\mathcal{I}|+|\mathcal{O}|$ many or twice as many variables (one for each 'end' of a transition). All BDD operations are performed by the `CUDD` library [6].

3 Modifying GR(1) Synthesis

Reactive synthesis has many applications, but most of them require the adaptation of the synthesis approach in order to yield useful implementations **and** to scale to problems of relevant size at the same time.

As one example, when performing automated high-level planning in robotics, liveness assumptions are often used to model that *doors* in some workspace must be open infinitely often. For a robot that needs to perform a certain task, this allows the robot to wait for a door to open, for example if it has to pass through the door in order to perform its task. Yet, high-level robot controllers are often observed to wait needlessly for doors to open when alternative paths exist; this is considered to decrease the quality of the controller, even if the specification is satisfied. While lifting the specification to a *weighted* one could solve the problem, solving (synthesis) games symbolically with costs is substantially harder and leads to unfavourable theoretical properties (e.g. optimal strategies require infinitely many states for mean-payoff games with liveness objectives). As an alternative, `slugs` contains a simple modification to the GR(1) fixpoint formula that penalizes such "waiting": the implementation in slugs takes just 7 lines of C++ code.

As another example, in high-level robotics applications [7], the position of a robot in a workspace is commonly under the control of the robot. Safety guarantees constrain the motion of the robot such that it can only move to adjacent regions of a workspace. However, the robot does not have control over where in the workspace it is deployed. A synthesized controller should thus be able to deal with *any* initial robot position. This makes the initial position an input to the controller that is used exactly once, namely when the controller starts. Integrating this additional input into the specification would lead to many more variables in the BDDs during synthesis, which decreases performance. As an alternative, we can confirm that all possible initial locations for the robot are winning by testing for membership in W. This change in the synthesis process needs a single line of code.

The `slugs` tool offers many plugins that implement other such modifications to the synthesis process. It was designed exactly with such modifications in mind and is optimized towards being easily extensible. In particular, all core parts of the synthesis process have been kept short and easily readable, to enable modifying them with the least effort possible. Slugs uses C++ features such as operator overloading to make all BDD operations concise and easily readable. Furthermore, the input language of `slugs` is very simple to parse. This facilitates modifications of the synthesis process that are based on *preprocessing* the specification. An additional script to translate from a richer input language to `slugs`' simpler language is also provided for convenience.

The `slugs` distribution can be obtained from https://github.com/VerifiableRobotics/slugs and comes equipped with a few plugins that implement techniques that can be found in the literature on GR(1) synthesis:

- The two plugins mentioned above (producing implementations that wait less for the environment and system initialization robotics semantics).
- A plugin to compute implementations that cooperate with the environment to satisfy the environment assumptions [3].
- A plugin to compute a counterstrategy from an unrealizable specification.
- Plugins to compute symbolic and explicit implementations, the former being represented as BDDs.
- A plugin to compute *estimators* for incomplete information synthesis [8].
- A plugin that lets `slugs` execute a controller in an interactive way, such that it can be used as a tool for simulating the controller called from other tools.
- Various plugins to compute *specification reports*, which help with debugging formal specifications as in [5].
- A plugin that allows output variables to be divided into two classes, "fast" and "slow", and ensures that each transition in the solution is safe even if the faster actions complete first [9].
- A plugin that computes implementations with *recovery transitions*, which allow the system to continue operating after safety assumption failures that can be compensated, similarly to the approach in [10].
- A plugin that computes the permissive implementations from [11].
- A plugin implementing the two-dimensional cost notion from [2].
- A plugin that computes a weakening of the assumptions in case of a realizable specification such that the specification stays realizable under the weakening.

In addition, the `slugs` distribution comes with several `Python` scripts that augment its functionality:

- A script to modify a specification such that it encodes the *error-resilient* synthesis problem for the original specification [12].
- A debugger to simulate implementations for realizable specifications and to simulate a falsifying environment for unrealizable specifications.
- A compiler that converts specifications with integer variables and constraints to purely boolean specifications.
- A specification report generator similar to the one described in [5].

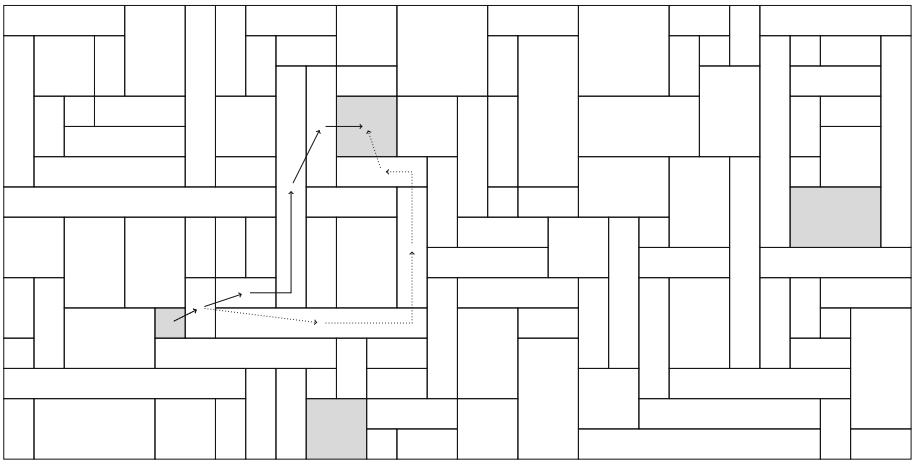

Fig. 1. A robot workspace and two paths to reach the respective next goal. The naive one is dotted, whereas the optimized one is not.

The `slugs` tool also allows to combine multiple plugins, provided that they have been marked as being compatible.

4 Slugs in Action

We now describe a concrete application made possible by a `slugs` plugin.

Consider the synthesis problem for a high level robot controller in which the robot operates on the workspace depicted in Fig. 1. The workspace is partitioned into regions, which all have different sizes. The position of the robot is under its control, but it can only move to adjacent cells in each step. There are four goals for the robot that each have to be visited infinitely often. For two of the goal regions, we have additional input bits. Those regions do not have to be reached if their respective input bit is **false**. The standard GR(1) synthesis algorithm does not use the relative sizes of the regions, and therefore the strategy that we synthesize in this setting is not optimal with respect to the physical grounding of the scenario. The figure shows one relatively complex path for getting from one gray region to another one that is part of the synthesized strategy.

To ensure that the physically more efficient strategy is obtained, the synthesis algorithm must be modified to incorporate worst-case costs on transitions between rooms. For this example, we use the difference between region centers as cost measure. `Slugs` comes with a plugin that implements GR(1) synthesis with a cost function, similar to the modifications described in [2]. The alternative path is shown in Fig. 1 as well. Note that the notion of optimality here is somewhat specific to the application domain, as we always optimize the cost for reaching the *next goal* of the robot, rather than using an optimization criterion such as mean-payoff.

To solve the realizability problem of the scenario, `slugs` needs 0.03 s on a i5 1.6 GHz computer. Extracting an explicit-state implementation takes 0.1 s in addition (1015 states), and synthesizing the optimal strategy takes another 14.9 s (resulting in 3315 states).

Acknowledgements. This work was supported by NSF ExCAPE and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

References

1. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
2. Jing, G., Ehlers, R., Kress-Gazit, H.: Shortcut through an evil door: optimality of correct-by-construction controllers in adversarial environments. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4796–4802 (2013)
3. Ehlers, R., Könighofer, R., Bloem, R.: Synthesizing cooperative reactive mission plans. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3478–3485 (2015)
4. Raman, V., Piterman, N., Finucane, C., Kress-Gazit, H.: Timing semantics for abstraction and execution of synthesized high-level robot control. IEEE Trans. Robot. **31**(3), 591–604 (2015)
5. Ehlers, R., Raman, V.: Low-effort specification debugging and analysis. In: 3rd Workshop on Synthesis (SYNT), pp. 117–133 (2014)
6. Somenzi, F.: CUDD: CU Decision Diagram package release 3.0.0 (2015)
7. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal-logic-based reactive mission and motion planning. IEEE Trans. Robot. **25**(6), 1370–1381 (2009)
8. Ehlers, R., Topcu, U.: Estimator-based reactive synthesis under incomplete information. In: 18th International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 249–258 (2015)
9. Raman, V., Finucane, C., Kress-Gazit, H.: Temporal logic robot mission planning for slow and fast actions. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 251–256 (2012)
10. Wong, K.W., Ehlers, R., Kress-Gazit, H.: Correct high-level robot behavior in environments with unexpected events. In: Robotics: Science and Systems (RSS) (2014)
11. Wen, M., Ehlers, R., Topcu, U.: Correct-by-synthesis reinforcement learning with temporal logic constraints. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4983–4990 (2015)
12. Ehlers, R., Topcu, U.: Resilience to intermittent assumption violations in reactive synthesis. In: 17th International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 203–212 (2014)