Slugs: Extensible GR(1) Synthesis

Rüdiger Ehlers[1] and Vasumathi Raman[2]

[1] University of Bremen and DFKI GmbH, Germany
[2] United Technologies Research Center, United States of America

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 (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, Section 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].

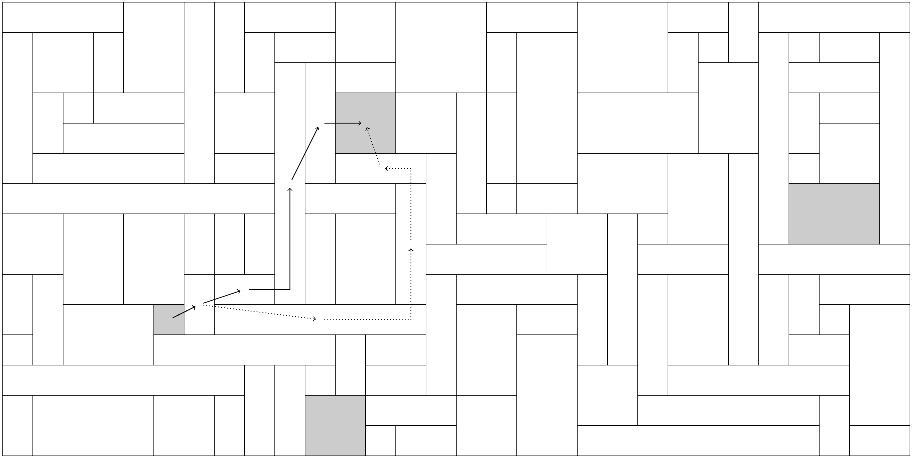

Fig. 1. A robot workspace and two paths to reach the respective next goal. The naive one is dotted, whereas the optimized one is not.

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 Figure 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 Figure 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 seconds on a i5 1.6GHz computer. Extracting an explicit-state implementation takes 0.1 seconds in addition (1015 states), and synthesizing the optimal strategy takes another 14.9 seconds (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) (2012) 911–938
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). (2013) 4796–4802
3. Ehlers, R., Könighofer, R., Bloem, R.: Synthesizing cooperative reactive mission plans. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). (2015) 3478–3485
4. Raman, V., Piterman, N., Finucane, C., Kress-Gazit, H.: Timing semantics for abstraction and execution of synthesized high-level robot control. IEEE Transactions on Robotics **31**(3) (2015) 591–604
5. Ehlers, R., Raman, V.: Low-effort specification debugging and analysis. In: 3rd Workshop on Synthesis (SYNT). (2014) 117–133
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. Robotics **25**(6) (2009) 1370–1381
8. Ehlers, R., Topcu, U.: Estimator-based reactive synthesis under incomplete information. In: 18th International Conference on Hybrid Systems: Computation and Control (HSCC). (2015) 249–258
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). (2012) 251–256
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). (2015) 4983–4990
12. Ehlers, R., Topcu, U.: Resilience to intermittent assumption violations in reactive synthesis. In: 17th International Conference on Hybrid Systems: Computation and Control (HSCC). (2014) 203–212

A Installing Slugs

The reactive synthesis tool `slugs` can be obtained by checking out a copy using the version control system `git`. On systems with a `git` installation that is usable from the command line interface, this can be done with the following command:

```
git clone https://github.com/VerifiableRobotics/slugs
```

While `slugs` uses the CUDD binary decision diagram library by Fabio Somenzi, it is already included in the cloned folder. The `slugs` main executable can then be compiled with the following commands:

```
cd slugs/src
make
```

The resulting executable will be put into the `src` directory. As a prerequisite, the following items must be installed:

- a moderately modern default compiler (clang or gcc),
- The C++ library `boost`

The library must be installed in a way such that it can be used by the compiler without setting additional paths. This is typically the case on Linux and MacOS machines.

In order to use the Python scripts that come with `slugs`, Python 2.x with $x \geq 6$ must be installed. The interactive specification debugger furthermore requires that the `python-libcurses` Python library is installed.

B Writing Slugs Specification

The synthesis tool `slugs` computes implementations from their specifications. A specification is given as a text file in one of two formats:

- The very basic `slugsin` format, or
- The more simple to use `structuredslugs` format.

The latter format is a strict extension of the former. In both cases, the input files are text files.

B.1 The basic input format

A `slugsin` file consists of multiple sections, which are started by *section headers*. These are:

- `[INPUT]`: The lines after this section header denote the atomic input propositions.
- `[OUTPUT]`: The lines after this section header denote the atomic output propositions.

- [ENV_TRANS]: After this section header, the *environment safety assumptions* are given.
- [SYS_TRANS]: After this section header, the *system safety guarantees* are given.
- [ENV_INIT]: After this section header, the *environment initialization assumptions* are given.
- [SYS_INIT]: After this section header, the *system initialization guarantees* are given.
- [ENV_LIVENESS]: After this section header, the *environment liveness assumptions* are given.
- [SYS_LIVENESS]: After this section header, the *system liveness guarantees* are given.

Any line in a `slugsin` file starting with a "#" symbol is a comment line and is ignored by the tool. Empty lines are also ignored. Before the first section header in an input file, there may only be empty or comment lines.

All non-comment and non-empty lines in the assumption and guarantee sections denote `constraints`. These are always given on individual lines. Constraints are given as Boolean formulas in Polish prefix notation, i.e., in which all non-unary operators are assumed to be binary, and the operators are written before the operands. There is also always a space between operators and operands. There is no *next* operator. If a constraint refers to the value of a variable at the end of a transition, this is denoted by adding a `'` to the variable name. The boolean operators available for specifying constraints are `!` (not), `|` (or), `&` (and), and `^` (exclusive or). Let us consider the following specification as an example:

```
[INPUT]
a
b

[OUTPUT]
x
y

[ENV_INIT]
& ! a ! b

[SYS_INIT]
& ! x ! y

[ENV_TRANS]
| a' ! | x y

[SYS_TRANS]
| ! x' ! y'

[SYS_LIVENESS]
```

```
& a y
```

In this specification, there are two input propositions, namely `a` and `b`, and there are two output propositions, called `x` and `y`. Initially, all of these have a value of **false**, as specified by the initialization assumption and the initialization guarantee. For example, the initialization assumption `& ! a ! b` represents the boolean formula $\neg a \wedge \neg b$. The only environment transition constraint states that $(a \rightarrow \neg(x \vee y))$ must always hold. So the environment may only set a to **true** during a transition if x and y are not both **false** in the last output signal valuation. Other than that, the next values for a and b may be arbitrary. The safety guarantees state that every output signal valuation chosen by the system *after* the system has started must not be $(x', y') = (\textbf{true}, \textbf{true})$.

There is one liveness guarantee for the system and no liveness assumption. The guarantee states that infinitely often, a and y need to be true in the same step of the system's execution. Since the system cannot control a, it needs to drive the environment into setting a to **true**. The environment assumptions allow this: since by them, a needs to be **true** in the next computation cycle after (x, y) have been set to $(\textbf{false}, \textbf{true})$, the system can just choose $(\textbf{false}, \textbf{true})$ as output for two cycles in a row to make progress towards satisfying the liveness guarantees. As this behavior does not contradict the safety guarantees, `slugs` can compute a controller.

There are no designated operators for implication ($\rightarrow$) and equivalence ($\leftrightarrow$) in the simple `slugsin` input language. However they can be simulated by the two-operator sequences "`| !`" and "`! ^`".

B.2 The structured slugs input format

The basic structure of a structured `slugs` specification file is the same as for a non-structured one. There are two extensions:

- The possibility to use infix notation in the constraints.
- Support for non-negative integer variables of bounded domains, including comparisons and additions of them.

The latter of these extensions allows to write expressions in a more human-readable form. E.g., instead of writing `| a' ! | x y` as in the example from the previous subsection, the user can simply write `a' | !(x | y)`. Implications, biimplication, and braces are supported, so `(! a' -> !(x | y))` is also a valid formula. It is allowed to mix constraints in infix form and in Polish prefix form, provided that they are all on their own lines. Operator precedence is as expected, i.e., unary operations bind strongest, then **and** and then **or**.

The support for integer variables allows to specify scenarios in which the progress of some quantity is described. In order to specify an integer variable, in the `[INPUT]` or `[OUTPUT]` block, the variable is declared in the form *variableName* : *minimumValue* ... *maximumValue*.

For example, when synthesizing a vehicle controller that obtains the current speed from some sensor, we can declare the input as follows:

```
[INPUT]
speed:0...127
```

This declaration adds 7 binary variables to the instance, which are enough to encode all 128 possible speed values. The only allowed arithmetic operation on integer variables is addition, and in order to obtain boolean constraints from expressions, the values of integer variables have to be compared against other values. For example, to constrain the system to not let the speed exceed 120 units, we could add the following constraint to the specification:

```
[SYS_TRANS]
speed' <= 120
```

In order to make this specification realizable, we need to give the system control over the speed. We do so by giving it an *acceleration output*:

```
[OUTPUT]
acc:0...5
```

The acceleration should always be an element of $\{-2, -1, 0, 1, 2\}$. Because the `structuredslugs` format does not allow non-negative numbers, we have to store the acceleration $+2$ instead. We can now describe the effect of acceleration by an environment safety assumption:

```
[ENV_TRANS]
speed'+2 = speed+acc
```

Normally, one would want to write `speed' = speed+acc-2` as the constraint, but as subtraction is not supported, the subtraction has to be shifted to the other side of the equation.

In the integer operations in `structuredslugs` specification files, there is no overflow semantics in the computations. Thus, by accelerating indefinitely, the system can force the environment to violate this constraint. Thus, we have to bound the speed from below and above:

```
[ENV_TRANS]
speed'+2 = speed+acc | speed'=127 & speed+acc>=129
                     | speed'=0 & speed+acc<=2
```

Note that the constraint needs to be written on a single line as line endings terminate constraints in `slugs` and `structuredslugs` files. If we want to add a noise of ± 1 per computation cycle to the speed update, we can do so by using the following requirements instead:

```
[ENV_TRANS]
speed'+2 <= speed+acc+1 | speed'=127 & speed+acc>=129
                        | speed'=0 & speed+acc<=2
speed'+2+1 >= speed+acc | speed'=127 & speed+acc>=129
                        | speed'=0 & speed+acc<=2
```

C Using Slugs

After a specification is written, slugs can be applied to it. From the slugs main directory, this can be done with the command

```
src/slugs <Options> [InputFileName.slugs]
```

By default, slugs only checks if the specification is realizable, i.e., if there exists an implementation. To obtain an implementation, a couple of output plugins are available:

- ✓ --explicitStrategy: lets slugs extract an explicit-state strategy. The output is compatible with the one produced by the GR(1) synthesis tool included in the jtlv formal methods framework distribution.
- ✓ --jsonOutput: this option can be given in addition to the previous one. It reformats the explicit-state output such that it can easily be read with a JSON parser.
- ✓ --symbolicStrategy: lets slugs extract a BDD-based symbolic strategy. There is one positional strategy per system goal in the output file.
- ✓ --simpleSymbolicStrategy: lets slugs extract a BDD-based symbolic strategy in which the positional strategies for the system goals have been merged into one strategy with a larger state space.

In case a structuredslugs file is the starting point, it first has to be translated to a standard slugsin file before it can be used. This can be done with the following command:

```
tools/StructuredSlugsParser/compiler.py [InputFile.
                                structuredslugs] > [OutputFile.slugsin]
```

In case there there is a syntax error in the structuredslugs file, a parser error will be printed.

C.1 Debugging a specification

Getting specifications right is difficult. Oftentimes, it makes sense to write a specification step-by-step and then check after each step if whether it is realizable or not is as expected. Whenever this is not the case, this indicates an unexpected implication of the specification or a bug in it. slugs comes with some scripts to help with debugging a specification.

The *interactive* debugger allows to examine the behavior of a controller for a realizable specification, and the behavior of an environment that falsifies an unrealizable specification. The debugger is based on the curses library for text-based user interfaces. The debugger can be started with the following command:

```
tools/cursesSimulator.py [inputFile.slugsin]
```

Figure 2 shows the interface of the debugger. It always shows the input and output values in the last 5 computation cycles. They can be changed by inputting values into some cells of the table. The up/down cursor keys allow the user to change the currently selected input/output variable, whereas the left/right key allow to change the computation cycles.

Variable values that have been enforced by the user are underlined. A color coding shows which variable values are enforced by safety assumptions/guarantees or by the winning strategy. Pressing the `h` button toggles the help screens. The help screens explain the color codes and and meaning of the flags that indicate which user-forced values are illegal. Along the input and output values, the debugger also shows the number of the current environment and system player goals. The text-based interface allows to run the debugger overnight on a computation server in combination with the Linux `screen` utility. This is useful for specifications whose realizability computation requires a lot of computation time.

```
SLUGS Simulator (stepmothers.slugsin)

 +-----------------+-------+-------+-------+
 | Round           | 0     | 1     | 2     |
 +=================+=======+=======+=======+
 | in1             | 0     | 0     | 0     |
 | in2             | 0     | 0     | 0     |
 | in3             | 0     | 4     | 4     |
 | in4             | 0     | 0     | 0     |
 | in5             | 0     | 3     | 3     |
 +=================+=======+=======+=======+
 | level1          | 0     | 0     | 0     |
 | level2          | 0     | 0     | 0     |
 | level3          | 0     | 4     | 8     |
 | level4          | 0     | 0     | 0     |
 | level5          | 0     | 3     | 6     |
 | choice          | 1     | 1     | 1     |
 +=================+=======+=======+=======+
 | Env. Goal. Num. | 0     | 0     | 0     |
 | Sys. Goal. Num. | 0     | 0     | 0     |
 | Flags           |       |       |       |
 +-----------------+-------+-------+-------+


Press (h) for help.
```

Fig. 2. The user interface of the interactive strategy debugger for a problem with the input signals $\{in1, \ldots, in4\}$ and the output signals $\{level1, \ldots, level5, choice\}$.

In addition to the specification debugger, there is also a Python script that computes a *specification report* from a specification. This functionality can be used as follows:

```
tools/createSpecificationReport.py [inputFile] > [outputFile.html]
```

The input file can either be in `slugsin` form or in `structuredslugs` form. Some of the analysis steps take a long time, but the report can be already be viewed with a web browser before all steps have been performed. The specification report generation process is an implementation of the concepts from [5].

C.2 Other plugins

The `slugs` tool comes with a variety of other plugins that alter the functionality of the tool. Most of them can be combined with any of the strategy extraction options given above. A complete list of parameters and their meaning can be optioned by running `slugs --help`. Some frequently used plugins are:

- ✓ `--sysInitRoboticsSemantics`: this lets `slugs` treat the initialization constraints in a way such that a specification is only called realizable if *all* initial valuations that satisfy the initialization assumptions and guarantees are winning for the system. This semantics is useful to tackle high-level robot specifications in which the initial position of the robot in a workspace is not under its control.
- ✓ `--biasForAction`: extracts controllers that rely on the liveness assumptions being satisfied as little as possible.
- ✓ `--counterStrategy`: for unrealizable specifications, this plugin computes a counter-strategy for the environment to force the violation of the specification. This plugin is currently not available in combination with a symbolic output strategy.
- ✓ `--simpleRecovery`: adds transitions to the system implementation that allow it to recover from sparse environment safety assumption faults in case this can done greedily.
- ✓ `--cooperativeGR1Strategy`: computes a controller that is cooperative with its environment, as explained in [3].

In addition, there is also a Python script that implements *error-resilient synthesis*, as defined in [12]. It can be used by executing

```
tools/kResilientRealizabilityChecker.py [inputFile.slugsin]
```

The output will be the error resilience levels that are realizable for the particular specification. Running the script with the parameter `--writeOptimalCaseSpecifications` will furthermore let the script create specification files that implement the maximal possible error-resilience levels. The new specification files will be named by the *inputFile*`(`*resilienceLevel*`).slugsin` scheme.

D Developing Plugins for Slugs

The main advantage of `slugs` over other stand-alone reactive synthesis tools is that its plugin architecture makes it easy to alter the synthesis process. All plugins can be found in the `src` directory of the `slugs` distribution and have a name of the form `extension....hpp`. So essentially, they are C++ header files.

The reason for the plugins being in header files is that `slugs` uses the C++ template mechanism in order to encapsulate the plugins in a way that makes them *easy to read* and pluggable in many ways at the same time. A basic plugin starts with the following source code:

```
#ifndef __X_YOUR_EXTENSION_HPP__
#define __X_YOUR_EXTENSION_HPP__

template<class T> class XYourExtension : public T {
protected:

    // Constructor
    XYourExtension<T>(std::list<std::string> &filenames)
     : T(filenames) {}

public:

    static GR1Context* makeInstance(std::list<std::string>
     &filenames) {
        return new XYourExtension<T>(filenames);
    }
};
#endif
```

In the code, the `YourExtension` string must be replaced by the actual name of the plugin. A plugin inherits the functions from some basetype T, which must inherit from the class `GR1Context` declared in `gr1context.hpp`. However, it may also inherit from a version of the class that has already been decorated by another plugin. The ways in which plugins are combinable is stated in `main.cpp`, including the inheritance chains used in each case. In order to ensure that a member field or function of the base class can be used, by the language semantics of C++, it must be declared. For example, in order to import the element `initEnv` from `GR1Context` (which contains the set of all positions in the synthesis game that satisfy the initialization assumptions), one would add the line

```
using T::initEnv;
```

below the `protected:` line of the new extension template class. A plugin mainly works by overriding functions from the `GR1Context` base class. It can also add new functions that can then be used in other plugins that *depend* on this one.

There are four main functions in the `GR1Context` that can be overriden:

- `void init(std::list<std::string> &filenames)`: This function is called immediately after constructing the `GR1Context`. It allocates the BDD variables, reads the specification file and parses it into BDDs.
- `void execute()`: This function orchestrates the main functionality of the tool. Plugins that alter `slugs`' behavior in a way such that it does not actually perform synthesis any more overwrite this function.

– `void checkRealizability()`: If the `execute` function is not overwritten, this function is called to determine if the specification is realizable. It (normally) first computes the set of winning positions in the synthesis game and then analyses it together with the initialization constraints to determine realizability.
– `void computeWinningPositions()`: This function computes the set of winning positions in the synthesis game built from the specification. The result is stored in the `GR1Context` field `winningPositions`. At the same time, the transitions leading closer to system goals, which are used for strategy extraction, are stored in the `GR1Context` field `strategyDumpingData`.

The `slugs` tool uses a couple of other custom classes in addition. The `BF` class represents a binary decision diagram (BDD). The term `BF` stands for *boolean function*. Most of the plugins do not perform operations that are specific to BDDs, so other data structures for representing boolean functions could also be used. In addition to the `BF` class, there are also the `BFVarVector` and `BFVarCube` classes, which represent variable lists and variable sets, respectively. The `BFManager` represents the general manager class for boolean functions. Currently, the class holds a reference to a CUDD manager object.

If during the development of a plugin, a BDD is to be printed at runtime, this can be performed by the `BF_newDumpDot` function. It gets four parameters:

– A reference to the variable info container,
– The BF/BDD to be printed,
– A variable group order, and
– A file name (with a `.dot`) ending.

The variable info container is used by the `BF_newDumpDot` function to obtain the names of the variables in the BDD. The variable group order can either be `NULL` or be a space-separated string of variable groups. The `GR1Context` container implements a BDD/BF variable manager, which in turn implements a variable info container, so when using `BF_newDumpDot` from a function in a plugin, the first parameter is typically `*this`.

The variable manager maintains a list of BF/BDD variables and the *types* of the variables. The types are all defined in `variableTypes.hpp`. The header file furthermore defines run-time strings for all variable types and a hierarchy between them. The hierarchy allows to split a variable type into several subtypes in some plugins while considering only more coarse-grained variable type sets in other plugins. An important feature of the `slugs` variable manager is that it comes with the two classes `SlugsVarVector` and `SlugsVarCube`. They inherit from `BFVarVector` and `BFVarCube` respectively, but add the possibility to compute them automatically from the data in the variable manager when instantiating them as fields in a descendent of `GR1Context`. As an example, the declaration of the `GR1Context` class contains the following field definition:

```
SlugsVarVector varVectorPre{PreInput, PreOutput, this};
```

It declares the variable varVectorPre, which consists of all variables of the type PreInput and PreOutput, and for the SlugsVariableManager implemented by the object (hence the this). A list of all variable types can be found in the header file variableTypes.hpp. Whenever a new variable is added, all SlugsVarVector and SlugsVarCube objects need to be updated. This is done by calling the computeVariableInformation function of the SlugsVariableManager. This is done after a specification has been read. When new BF/BDD variables are allocated in plugins, the function must be called manually afterwards.

After a new plugin has been implemented, it needs to be connected to the main program. This is done in three steps:

– An #include directive for the new extension needs to be added to main.cpp
– A command line parameter and its explanation needs to be added to the commandLineArguments list.
– All desireed parameter combinations with the new plugin need to be added to optionCombinations list in main.cpp. The command line arguments leading to the plugin combination must be given in lexicographically ordered form.

Afterwards, the plugin can be used in the specified parameter combinations. The feasible combinations of the plugins that are already included in the slugs distribution are managed by the plugin_combination_enumerator.py script. It contains a list of plugins, requirements on their combinations, and fills the said lists in the main.cpp file with the allowed combinations.