Robot-on-a-grid Moving Obstacle Evasion Benchmarks

Rüdiger Ehlers
University of Bremen & DFKI GmbH
Germany

I. Basic Setup

This benchmark set is concerned with a robot moving on an $m_x \times m_y$-cell two-dimensional grid with the objective to avoid colliding with a 2×2-cell moving obstacle. In every step, the robot can move by ± 1 cell in the x direction, and/or by ± 1 cell in the y direction. Collisions with the boundaries of the workspace do not need to be avoided. However, the workspace does not have a torus shape, so that its boundaries cannot be crossed.

Motion is triggered by the robot using two atomic propositions for the x direction, and two propositions for the y direction. The robot has no inertia. Providing the updated x- and y-coordinates is the task of the environment.

In order to allow the robot to evade the moving obstacle, the obstacle can only move in every second computation step, while the robot can move in every computation step.

The overall setting is very simple and abstracts from many aspects of interest in the robotics application domain. However, it is already challenging for large workspace sizes and the abstraction from the details present in robotics applications makes it easy to understand.

A. Input and output signals

We have the following signals:

- Inputs to the system to be synthesized:
  1) signals $rx_0, \ldots, rx_{|rx|}$ that encode the binary representation of the current x position of the robot
  2) signals $ry_0, \ldots, ry_{|ry|}$ that encode the binary representation of the current y position of the robot
  3) signals $ox_0, \ldots, ox_{|ox|}$ that encode the binary representation of the current x position of the obstacle
  4) signals $oy_0, \ldots, oy_{|oy|}$ that encode the binary representation of the current y position of the obstacle
- Outputs of the system to be synthesized:
  1) signals mx_0, mx_1 to encode whether the robot wants to move left, right, or does not want to change its x position
  2) signals my_0, my_1 to encode whether the robot wants to move up, down, or does not want to change its y position
  3) signal $obsmove$ with which the system must track whether the obstacle has moved in the previous step.

B. Specification

The specification consists of *assumptions* and *guarantees*, which are connected by a *strong implication* as common in GR(1) synthesis, i.e., it is the aim of the system to make sure that no guarantee is violated before some assumption is violated (see, e.g., [1], section 3.3). If some assumption and some guarantee are violated in the same step, the system also satisfies its overall specification.

- Assumptions:
  1) At every step, the obstacle and the robot can only move by at most one step in x and y directions each.
  2) The robot position is updated according to the moves chosen by the system.
  3) Initially, the x position and y position of the robot are both 0.
  4) The binary encoding of the robot's x position is never $\geq m_x$.
  5) The binary encoding of the robot's y position is never $\geq m_y$.
  6) The binary encoding of the x position of the obstacle's upper left corner is never $\geq m_x - 1$.
  7) The binary encoding of the y position of the obstacle's upper left corner is never $\geq m_y - 1$.
  8) The obstacle can only change its position if before the change, $obsmove$ is true.
  9) Initially, the obstacle is in the workspace corner that is most distant from the robot.
- Guarantees:
  1) The robot is never within the boundaries of the moving obstacle
  2) $obsmove$ is set to false precisely after a change of the obstacle position
  3) mx_0 and mx_1 together represent a valid move
  4) my_0 and my_1 together represent a valid move

II. Error-resilience

In the basic setup, the robot can move twice as fast as the obstacle, which allows the robot to evade the obstacle for workspace sizes of 5×5 cells or larger.

To make the benchmark a bit more challenging, we add a little twist: during the run of the system, the environment may temporarily violate Assumption 8 a few times, so that the obstacle can sometimes move in successive steps. Such temporary assumption violations are also called *glitches*.

To implement this idea, a *glitch counter* is introduced that is under the control of the environment, i.e., is an input to the system. The counter has a maximum value of c for some $c \in \mathbb{N}$ and is binary-encoded into a set of additional input signals of sufficient number.

Assumption 8 of the basic setting is removed and the following assumptions are added:

1) The glitch counter value can only increase over time
2) Whenever the obstacle moves while $obsmove$ is false, the glitch counter value strictly increases
3) The glitch counter value is never $> c$

Note that the error-resilience model here is relatively simple: a finite, fixed number of glitches must be tolerated during the execution of the system (which is infinitely long). This is in contrast to more advanced error-resilience schemes such as k-resilience [2], [3], where an infinite number of blocks of the system's execution with up to k (not necessarily consecutive) glitches each need to be tolerated, provided that there is enough time for the system to recover from the glitches in between the blocks. In contrast to GR(1) synthesis, without using liveness properties, a second counter for measuring the length of the recovery period would have been needed here, which is why the simpler error-resilience notion was used for this benchmark set.

III. Compilation workflow

The benchmarks have been formulated as *structured specifications* for the generalized reactivity(1) game solver SLUGS [4]. The term *structured* in this context refers to support for constraints over (non-negative) integer numbers, which are automatically translated to Boolean constraints when compiling the structured SLUGS specification into a purely boolean form.

The purely boolean GR(1) safety specification is then translated to an and-inverter-graph (AIG) representation of a monitor automaton that checks the satisfaction of the specification. The AIG is finally optimized using the ABC toolset [5] by applying the `rewrite` command.

IV. Configurations

Table I lists the configurations used as benchmarks. To allow a comparison with the later outcomes of the SyntComp, computation times of the SLUGS GR(1) synthesis tool on the benchmarks (before translation to the AIG monitor automaton form) are given. They are wall-clock times and have been obtained on a computer with an AMD E-450 processor running an x86 Linux at 1.6 GHz with 4GB of memory.

The tool SLUGS has been used in its version from the 21st of Februrary 2014. The parameter `--onlyRealizability` has been supplied to the tool in order to switch off extracting an explicit-state implementation in case the specification is found to be realizable.

References

[1] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.

[2] C.-H. Huang, D. Peled, S. Schewe, and F. Wang, "Rapid recovery for systems with scarce faults," in *GandALF* (M. Faella and A. Murano, eds.), vol. 96 of *EPTCS*, pp. 15–28, 2012.

[3] R. Ehlers and U. Topcu, "Resilience to intermittent assumption violations in reactive synthesis," in *HSCC*, 2014.

[4] R. Ehlers, C. Finucane, and V. Raman, "Slugs GR(1) synthesizer." http://github.com/ltlmop/slugs, 2013.

[5] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification, release 140221, http://www.eecs.berkeley.edu/~alanmi/abc/."

m_x	m_y	c	is realizable	SLUGS computation time
8	8	0	yes	0.3s
8	8	1	no	0.4s
16	16	3	yes	1.7s
16	16	4	no	4.5s
24	24	7	yes	27.6s
24	24	8	no	20s
32	32	11	yes	53s
32	32	12	no	53s
48	48	19	yes	5m59s
48	48	20	no	4m54s
64	64	27	yes	36m12s
64	64	28	no	54m40s
96	96	43	yes	62m44s
96	96	44	no	95m44.829s
128	128	59	unknown	$>$ 4h
128	128	60	unknown	$>$ 4h

TABLE I
PARAMETER COMBINATIONS USED AS BENCHMARKS