Simple Factory Assembly Line Benchmarks

Rüdiger Ehlers
University of Bremen & DFKI GmbH
Germany

I. Basic Setup

This benchmark set is concerned with a very simple assembly line. Figure 1 depicts the scenario graphically. We have a conveyor belt with $n = 7$ places that are reachable by two manipulator arms, and on each of the places, there is an object. Before an object is pushed off the belt, $m = 4$ tasks have to be performed on the object. Every k cycles, the belt rotates by one place and delivers a new object. At every step, a manipulator arm can perform one task on the object underneath it. The manipulator arms can also move by one place, but they can never be at the same place and can also not overtake each other. The objects come with one task already having been performed on them. Initially, the belt is filled with objects for which all tasks have been performed already.

The scenarios in this benchmark set have many input and output signals, but only few counters. Thus, they aim at complementing counter-rich other benchmarks in the SyntComp.

A. Input and output signals

We have the following signals:

- Boolean inputs to the system to be synthesized:
  1) signals $ck_0, \ldots, ck_{|ck|}$ that encode the binary representation of a counter for how many steps are left until k steps without belt motion have been witnessed
  2) signals $\{p_{i,j}\}_{0 \leq i < n, 0 \leq j < m}$ that encode whether task j has already been performed on the object at position i
- Boolean outputs of the system to be synthesized:
  1) signals $arm1pos_1, \ldots, arm1pos_{\lceil \log_2 n \rceil}$ to encode the position of the first arm
  2) signals $arm2pos_1, \ldots, arm2pos_{\lceil \log_2 n \rceil}$ to encode the position of the second arm
  3) signals $arm1action_1, \ldots, arm1action_{\lceil \log_2 m \rceil}$ to encode which task the first arm is currently performing
  4) signals $arm2action_2, \ldots, arm1action_{\lceil \log_2 m \rceil}$ to encode which task the first arm is currently performing

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.

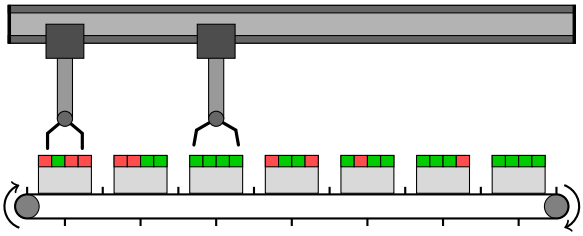

Fig. 1. Schematic drawing of a simple assembly line

- Assumptions:
  1) The ck counter cycles through the values $0, \ldots, k-1$.
  2) Whenever the ck counter becomes 0, all objects are pushed forward by one position, and the new object at the left of the belt has at least one task already been performed. Tasks performed by the robot arms on the objects are applied before the push.
  3) Whenever the ck counter is not 0, the object task signals that represent tasks currently being performed by one of the arms become **true**. All others retain their value.
  4) Initially, the tasks of all objects on the belt have already been performed, and the ck counter starts with a value of 0.
- Guarantees:
  1) Manipulator arm 1 is always left of manipulator arm 2
  2) All arms can move by at most one position per round.
  3) Before a push occurs, all tasks of the right-most objects must have already been performed.

II. Error-resilience

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 2 in the sense that a new object without any task having already been performed can be placed onto the belt. We call such a case a *glitch* for the scope of this benchmark description.

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 2 is replaced by the requirement that whenever the ck counter becomes 0, all objects are pushed forward by one position. Tasks performed on the objects are still applied before the push. We add the following assumptions:

1) The glitch counter value can only increase over time
2) Whenever in some round, a new object is pushed onto the belt for which no task has already been performed, the glitch counter is increased by 1.
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 when using GR(1) synthesis, we cannot use liveness properties in the SYNTCOMP, so a second counter for measuring the length of the recovery period would have been needed here for adding a k-resilience system requirement to the benchmarks, 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 generalized reactivity(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 command sequence `rewrite`.

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	n	k	c	is realizable	SLUGS computation time
5	3	1	0	yes	0.8s
5	4	1	0	no	2.7s
5	5	2	0	yes	55.0s
5	6	2	0	no	9m15.0s
7	3	1	0	yes	2.3s
7	5	2	0	yes	5m36.2s
3	3	1	1	no	0.2s
4	3	1	1	yes	0.5s
5	3	1	4	yes	2.4s
5	3	1	5	no	2.6s
5	5	2	1	yes	5m29s
5	5	2	10	yes	14m25s
5	5	2	11	no	16m15s
7	5	2	10	yes	118m1s
7	5	2	11	unknown	$>$ 4h

TABLE I
PARAMETER COMBINATIONS USED AS BENCHMARKS