Unbeast: Symbolic Bounded Synthesis*

Rüdiger Ehlers

Reactive Systems Group
Saarland University

Abstract. We present Unbeast v.0.6, a tool for synthesising finite-state systems from specifications written in linear-time temporal logic (LTL). We combine bounded synthesis, specification splitting and symbolic game solving with binary decision diagrams (BDDs), which allows tackling specifications that previous tools were typically unable to handle. In case of realizability of a given specification, our tool computes a prototype implementation in a fully symbolic way, which is especially beneficial for settings with many input and output bits.

1 Introduction

Specification engineering is known to be a tedious and error-prone task. During the development of complex systems, the early versions of the specifications for the individual parts of the system often turn out to be incomplete or unrealisable. In order to effectively detect such problems early in the development cycle, specification debugging tools have been developed.

One particularly well-known representative of this class are synthesis tools for reactive systems. These take lists of input and output bits and a temporal logic formula in order to check whether there exists a reactive system reading from the specified input signals and writing to the given output signals that satisfies the specification. In case of a positive answer, they also generate a finite-state representation of such a system. Then, by simulating the resulting system and analysing its behaviour, missing constraints in the specification can often be found.

In this work, we report on the Unbeast tool, which performs this task for specifications written in linear-time temporal logic (LTL). By combining the merits of the bounded synthesis approach [7] with using binary decision diagrams (BDDs) as the symbolic reasoning backbone and the idea of splitting the specification into safety and non-safety parts, we achieve competitive computation times in the synthesis process. Our approach extracts implementations for realisable specifications in a fully symbolic manner, which is especially fruitful for systems with many input and output bits. As a consequence, even in the development of comparably complex systems that typically fall into this class, our tool is applicable to at least the initial versions of a successively built specification, despite the fact that the synthesis problem is 2EXPTIME-complete.

* This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

2 Tool description

Input language: The tool takes as input XML files that provide all information necessary for the synthesis process. We assume that the specification of the system has the form $(a_1 \wedge \ldots \wedge a_n) \rightarrow (g_1 \wedge \ldots \wedge g_m)$, where each of $a_1, \ldots, a_n, g_1, \ldots, g_m$ is an LTL formula. Such formulas are typical for cases in which a component of a larger system is to be synthesised: the *assumptions* $a_1, \ldots, a_n$ represent the behaviour of the environment that the system can assume, whereas the guarantees $g_1, \ldots, g_n$ reflect the obligations that the system to be synthesised needs to fulfil.

Tool output: In case of a realisable specification, a system implementation in form of a NuSMV [2] model is produced (if wanted). Alternatively, the user has the possibility to run a simulation of a system satisfying the specification, where the user provides the input to the system. If the specification is unrealisable, the roles in the simulation are swapped – the tool then demonstrates interactively which environment behaviour leads to a violation of the specification.

2.1 Technology

The Unbeast v.0.6 tool implements the synthesis techniques presented in [7, 3]. We use the library Cudd v.2.4.2 [8] for constructing and manipulating BDDs during the synthesis process. We constrain its automatic variable reordering feature by enforcing that predecessor and successor variables in the transition relation are pairwise coupled in the ordering.

The first step is to determine which of the given assumptions and guarantees are safety formulas. In order to detect also simple cases of *pathological safety* [6], this is done by computing an equivalent Büchi automaton using the external LTL-to-Büchi converter ltl2ba v.1.1 [5] and examining whether all maximal strongly connected components in the computed automaton do not have infinite non-accepting paths. We take special care of bounded look-ahead safety formulas: these are of the form $\mathsf{G}(\psi)$ for the LTL globally operator G and some formula ψ in which the only temporal operator allowed is the next-time operator. They are immediately identified as being safety formulas.

In a second step, for the set of bounded look-ahead assumptions and the set of such guarantees, we build two safety automata for their respective conjunctions. Both of them are represented in a symbolic way, i.e., we allocate predecessor and successor state variables that encode the last few input/output bit valuations and compute a transition relation for this automaton in BDD form. For the remaining safety assumptions and guarantees, safety automata are built by taking the Büchi automata computed in the previous step and applying a subset construction for determinisation in a symbolic manner. For the remaining non-safety parts of the specification, a combined universal co-Büchi automaton is computed by calling ltl2ba again.

In the next phase, the given specification is checked for realisability. Here, for a successively increasing so-called *bound value*, the bounded synthesis approach [7] is performed by building a safety automaton from the co-Büchi automaton

Table 1. Running times (in seconds) of UNBEAST v.0.6 and ACACIA v.0.3.9.9 on the load balancing case study on a Sun XFire computer with 2.6 Ghz AMD Opteron processors running an x64-version of Linux. All tools considered are single-threaded. We restricted the memory usage to 2 GB and set a timeout of 3600 seconds.

Tool	Setting / # Clients	2	3	4	5	6	7	8	9
A		+ 0.4	+ 0.5	+ 0.7	+ 0.9	+ 1.4	+ 2.7	+ 5.5	+ 12.7
U+S	1	+ 0.0	+ 0.1	+ 0.0	+ 0.0	+ 0.1	+ 0.1	+ 0.1	+ 0.1
U−S		+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.1	+ 0.1
A		+ 0.4	+ 0.4	+ 0.4	+ 0.5	+ 0.6	+ 0.9	+ 1.6	+ 3.0
U+S	$1 \wedge 2$	+ 0.0	+ 0.1	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.1	+ 0.1
U−S		+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.0	+ 0.1
A	$1 \wedge 2 \wedge 3$	- 21.8	- 484.3	timeout	timeout	timeout	memout	memout	timeout
U		- 0.1	- 0.1	- 0.1	- 0.1	- 0.3	- 1.0	- 6.8	- 73.2
A		+ 0.7	+ 1.4	+ 8.5	memout	memout	memout	memout	timeout
U+S	$1 \wedge 2 \wedge 4$	+ 0.2	+ 0.3	+ 1.0	+ 35.5	+ 214.1	timeout	timeout	timeout
U−S		+ 0.1	+ 0.2	+ 0.3	+ 2.5	+ 3.1	+ 11.2	+ 48.3	+ 386.4
A	$1 \wedge 2 \wedge 4 \wedge 5$	- 148.7	timeout	timeout	timeout	memout	memout	memout	timeout
U		- 0.2	- 0.5	- 909.4	timeout	timeout	timeout	timeout	timeout
A	$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 179.1	timeout	timeout	timeout	memout	memout	timeout	timeout
U		- 0.1	- 0.7	- 585.5	timeout	timeout	timeout	timeout	timeout
A	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 182.7	memout	timeout	timeout	timeout	timeout	timeout	timeout
U		- 0.2	- 1.5	- 787.9	timeout	timeout	timeout	timeout	timeout
A		+ 11.6	+ 68.8	+ 406.6	memout	timeout	timeout	timeout	timeout
U+S	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 0.1	+ 0.4	+ 1.4	+ 86.6	+ 1460.4	timeout	timeout	timeout
U−S		+ 0.1	+ 0.2	+ 0.4	+ 3.7	+ 3.9	+ 17.7	+ 84.1	+ 1414.3
A	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 41.0	- 1498.9	timeout	memout	timeout	timeout	timeout	timeout
U		- 0.1	- 0.1	- 0.2	- 0.9	- 15.8	- 427.9	timeout	timeout
A		+ 67.5	+ 660.1	memout	timeout	timeout	timeout	timeout	timeout
U+S	$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 0.3	+ 2.2	+ 36.0	+ 899.3	timeout	timeout	timeout	timeout
U−S		+ 0.2	+ 0.6	+ 11.6	+ 16.9	+ 1222.2	timeout	timeout	timeout

for the non-safety part of the specification and solving the safety games induced by a special product of the automata involved [3].

Finally, if the specification is found to be realisable (i.e., the game computed in the previous phase is winning for the player representing the system to be synthesised), the symbolic representation of the winning states of the system is used to compute a prototype implementation satisfying the specification. Intuitively, this is done by constructing a circuit that keeps track of the current position in the game and computes a transition to another winning position whose input matches the one observed after each computation cycle. At the same time, the output labelling along the transition is used as the output for the system. For this computation, only the BDD representation of the winning positions is used.

3 Experimental results

We use a load balancing system [3] as our case study. The specification is supposed to be successively built by an engineer who applies a synthesis tool after each modification of the specification. The setting is parametrised by the number of servers the balancer is meant to work for. For some number of servers $n \in \mathbb{N}$, the load balancer has $n + 1$ input bits and n output bits.

Table 1 surveys the results. The individual assumptions and guarantees in the specification are numbered as in [3]. For example, the setting $6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$

corresponds to a specification with the assumption no. 5 and the guarantees no. 2, 4, and 6. For every setting, the table describes whether the specification was found to be realisable ("+") or not ("−") and the running times for the ACACIA v.0.3.9.9 [4] tool (abbreviated by "A") and UNBEAST (abbreviated by "U"). Both tools can only check for either realisability or unrealisabilty at a time. Thus, we ran them for both cases concurrently and only report the running times of the instances that terminated with a positive answer. For realisable specifications, for our tool, we distinguish between the cases that a system is to be synthesised ("$+S$") or just realisability checking is to be performed ("$-S$"). We did not configure ACACIA to construct an implementation.

4 Conclusion

We presented UNBEAST, a tool for the synthesis of reactive systems from LTL specifications. In the experimental evaluation, we compared our tool against ACACIA on a case study from [3] and found it to be faster in all cases, sometimes even orders of magnitude. For academic purposes, the tool can freely be downloaded from `http://react.cs.uni-saarland.de/tools/unbeast`.

Especially when only realisability of a specification is to be checked, the BDD-based bounded synthesis approach turns out to work well. However, it can be observed that extracting a prototype implementation significantly increases the computation time. This is in line with the findings in [1], where the same observation is made in the context of synthesis from a subset of LTL. We see this as a strong indication that the problem of extracting winning strategies from symbolically represented games requires further research.

References

1. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. ENTCS **190**(4) (2007) 3–16
2. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In Brinksma, E., Larsen, K.G., eds.: CAV. Volume 2404 of LNCS., Springer (2002) 359–364
3. Ehlers, R.: Symbolic bounded synthesis. In Touili, T., Cook, B., Jackson, P., eds.: CAV. Volume 6174 of LNCS., Springer (2010) 365–379
4. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In Bouajjani, A., Maler, O., eds.: CAV. Volume 5643 of LNCS., Springer (2009) 263–277
5. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In Berry, G., Comon, H., Finkel, A., eds.: CAV. Volume 2102 of LNCS., Springer (2001) 53–65
6. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3) (2001) 291–314
7. Schewe, S., Finkbeiner, B.: Bounded synthesis. In Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y., eds.: ATVA. Volume 4762 of LNCS., Springer (2007) 474–488
8. Somenzi, F.: CUDD: CU Decision Diagram package release 2.4.2 (2009)

A Tool information

UNBEAST v.0.6 was written in C++ and is available for download from the URL given in the paper in an aggregated source code form. The tool can freely be used for academic purposes, but does not have an open source license. UNBEAST comes with comprehensive documentation that gets one easily started.

B Testing the output of the tool

In order to gain confidence in the correctness of the results produced by our tool, we ran some tests. First of all, the realisability/unrealisability results for the case study in the paper always match between the ACACIA tool and UNBEAST.

Additionally, we used the 23 mutex variations from [11, 4] as test cases. For usage with our tool, we adapted these examples to the Mealy-type computation model used in this work by prefixing all references to input variables with a next-time operator. The realisability/unrealisability results obtained always matched the one obtained using Lily v.1.0.2 [11] and ACACIA v.0.3.9.9 [4]. For the specifications found to be realisable, we used NUSMV for verifying the generated models. NUSMV never found a counter-example.

C Some notes on the content of this tool paper

Operator precedences: Other synthesis tools like ACACIA and LILY use a subset of PSL as their input language. We decided not to do so as we are not aware of a formal description document for PSL that is publicly available. The book typically referred to in this context [10] does not provide a truly formal semantics and clear operator precedences. As we are also aware of differences in the operator precedences between different LTL-to-Büchi tools, we decided to circumvent all these problems by requiring the user to make everything explicit in the XML file. Even if we stated the precedences in the documentation of the tool, users would still sometimes accidentally assume other precedence rules that they are familiar with from other tools, which leads to problems that are very hard to find.

Assumptions→Guarantees specification form: We assumed that the specifications given consist of sets of assumptions and guarantees. This is a common choice in the literature, see, e.g., [1, 9, 12].

On comparing against* Acacia*: The tool ACACIA has a lot of options, many of them optimised for specifications which are big conjunctions of LTL formulas parts (so there are no assumptions, or these are made local to the guarantees). As our specifications in this paper mostly have assumptions (and comparing against all parameter settings is infeasible for a short tool paper), we only used the default settings.

In the experimental evaluation, we used Acacia with parameters that made the tool only check for realisability/unrealisability. This is justified by the fact

that for the techniques involved there, implementation extraction is very fast and thus no big deal anyway.

The computation model used: The tool UNBEAST uses a Mealy-type computation model, i.e., we assume that in every computation cycle, first the input to the system arrives and then the system can choose the output. Details and a justification are provided in [3].

Syntactic safety detection: In the main part of the paper, we describe a technique to detect some cases of pathological safety. Note that the LTL-to-Büchi converter ensures that maximal strongly connected components without accepting states are already pruned away. Otherwise using the algorithm described would sometimes lead to incorrect results.

Comparison against version 0.5 of* Unbeast*: In [3], we reported benchmark results on the same case study as here. The experimental evaluation in that paper was based on an earlier version of our tool (which was not mentioned at all in [3]). The following new features have been added in the meantime:

- A simulator for unrealisable and realisable specifications
- Variable grouping in the BDD order (which has an impact on the performance)

Differences in the experimental evaluation in comparison to [3]: In [3], we did not consider that case that only realisability checking is to be performed. In order to give the reader a better impression of the difficulty of actually extracting an implementation, we decided to do so in this tool paper.

D Example input file

The following example specification describes a two-process mutex:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE SynthesisProblem SYSTEM "SynSpec.dtd">
<SynthesisProblem>
  <Title>A mutex</Title>
  <Description>A not quite complex example</Description>
  <PathToLTLCompiler>ltl2ba-1.1/ltl2ba</PathToLTLCompiler>
  <GlobalInputs>
    <Bit>Request0</Bit>
    <Bit>Request1</Bit>
  </GlobalInputs>
  <GlobalOutputs>
    <Bit>Grant0</Bit>
    <Bit>Grant1</Bit>
  </GlobalOutputs>
```

```
  <Assumptions>
    <LTL><G><F><Not><Var>Request0</Var></Not></F></G></LTL>
    <LTL><G><F><Not><Var>Request1</Var></Not></F></G></LTL>
  </Assumptions>
  <Specification>
    <LTL><G><Or><Not><Var>Grant0</Var></Not>
       <Not><Var>Grant1</Var></Not></Or></G></LTL>
    <LTL><G><Or><Not><Var>Request0</Var></Not><F>
       <Var>Grant0</Var></F></Or></G></LTL>
    <LTL><G><Or><Not><Var>Request1</Var></Not><F>
       <Var>Grant1</Var></F></Or></G></LTL>
  </Specification>
</SynthesisProblem>
```

References

9. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In: DATE. (2007) 1188–1193
10. Eisner, C., Fisman, D.: A Practical Introduction to PSL (Series on Integrated Circuits and Systems). Springer (2006)
11. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD. (2006) 117–124
12. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: VMCAI. (2006) 364–380