# BoSy: An Experimentation Framework for Bounded Synthesis[⋆]

Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup

Saarland University, Saarbrücken, Germany
`lastname@react.uni-saarland.de`

**Abstract.** We present BoSy, a reactive synthesis tool based on the bounded synthesis approach. Bounded synthesis ensures the minimality of the synthesized implementation by incrementally increasing a bound on the size of the solutions it considers. For each bound, the existence of a solution is encoded as a logical constraint solving problem that is solved by an appropriate solver. BoSy constructs bounded synthesis encodings into SAT, QBF, DQBF, EPR, and SMT, and interfaces to solvers of the corresponding type. When supported by the solver, BoSy extracts solutions as circuits, which can, if desired, be verified with standard hardware model checkers. BoSy won the LTL synthesis track at SYNTCOMP 2016. In addition to its use as a synthesis tool, BoSy can also be used as an experimentation and performance evaluation framework for various types of satisfiability solvers.

## 1 Introduction

The reactive synthesis problem is to check whether a given $\omega$-regular specification, usually presented as an LTL formula, has an implementation, and, if the answer is yes, to construct such an implementation. As a theoretical problem, reactive synthesis dates back all the way to Alonzo Church's solvability question [6] in the 1950s; as a practical engineering challenge, the problem is fairly new. Tools for reactive synthesis started to come out around 2007 [5, 9, 11, 17]. The first SYNTCOMP tool competition took place at CAV 2014 and was originally restricted to safety specifications, and only later, starting with CAV 2016, extended with an LTL synthesis track [14].

In this paper, we present BoSy, the winner of the 2016 LTL synthesis track. BoSy is based on the bounded synthesis approach [12]. Bounded synthesis ensures the minimality of the synthesized implementation by incrementally increasing a bound on the size of the solutions it considers. For each bound, the existence of a solution is encoded as a logical constraint solving problem that is solved by an appropriate solver. If the solver returns "unsat", the bound is increased and a new constraint system is constructed; if the solver returns a satisfying assignment, an implementation is constructed.

⋆ Supported by the European Research Council (ERC) Grant OSARES (No. 683300).

From an engineering perspective, an interesting feature of the bounded synthesis approach is that it is highly modular. The construction of the constraint system involves a translation of the specification into an $\omega$-automaton. Because the same type of translation is used in model checking, a lot of research has gone into optimizing this construction; well-known tools include ltl3ba [1] and spot [8]. On the solver side, the synthesis problem can be encoded in a range of logics, including boolean formulas (SAT), quantified boolean formulas (QBF), dependency quantified boolean formulas (DQBF), the effective propositional fragment of first-order logic (EPR), and logical formulas with background theories (SMT) (cf. [10]). For each of these encodings, there are again multiple competing solvers.

BoSy leverages the best tools for the LTL-to-automaton translation and the best tools for solving the resulting constraint systems. In addition to its main purpose, which is the highly effective synthesis of reactive implementations from LTL specifications, BoSy is therefore also an experimentation framework, which can be used to compare individual tools for each problem, and even to compare tools across different logical encodings. For example, the QBF encoding is more compact than the SAT encoding, because it treats the inputs of the synthesized system symbolically. BoSy can be used to validate, experimentally, whether QBF solvers translate this compactness into better performance (spoiler alert: in our experiments, they do). Likewise, the DQBF/EPR encoding is more compact than the QBF encoding, because this encoding treats the states of the synthesized system symbolically. In our experiments, the QBF solvers nevertheless outperform the currently available DQBF solvers.

In the remainder of this paper, we present the tool architecture, including the interfaces to other tools, and report on experimental results[1].

## 2 Tool Architecture

An overview of the architecture of BoSy is given in Figure 1. For a given bounded synthesis instance, BoSy accepts a JSON-based input format that contains a specification $\varphi$ given in LTL, the signature of the implementation given as a partition of the set of atomic propositions into inputs and outputs, and the target semantics as a Mealy or a Moore implementation. In the preprocessing component, the tool starts to search for a system *strategy* with $\varphi$ and an environment *counter-strategy* with $\neg\varphi$ in parallel.[2]

After parsing, the LTL formula is translated into an equivalent universal co-Büchi automaton using an external automata translation tool. Currently, we support ltl3ba [1] and spot [8] for this conversion, but further translation tools can be integrated easily. Tools that output SPIN never-claims or the HOA format are supported.

[1] BoSy is available online at `https://react.uni-saarland.de/tools/bosy/`.

[2] The LTL reactive synthesis problem is not dual with respect to dualizing the LTL formula only, but the target semantics has to be adapted as well. If one searches for a transition-labeled (Mealy) implementation, a counterexample is state-labeled (Moore) and vice versa.

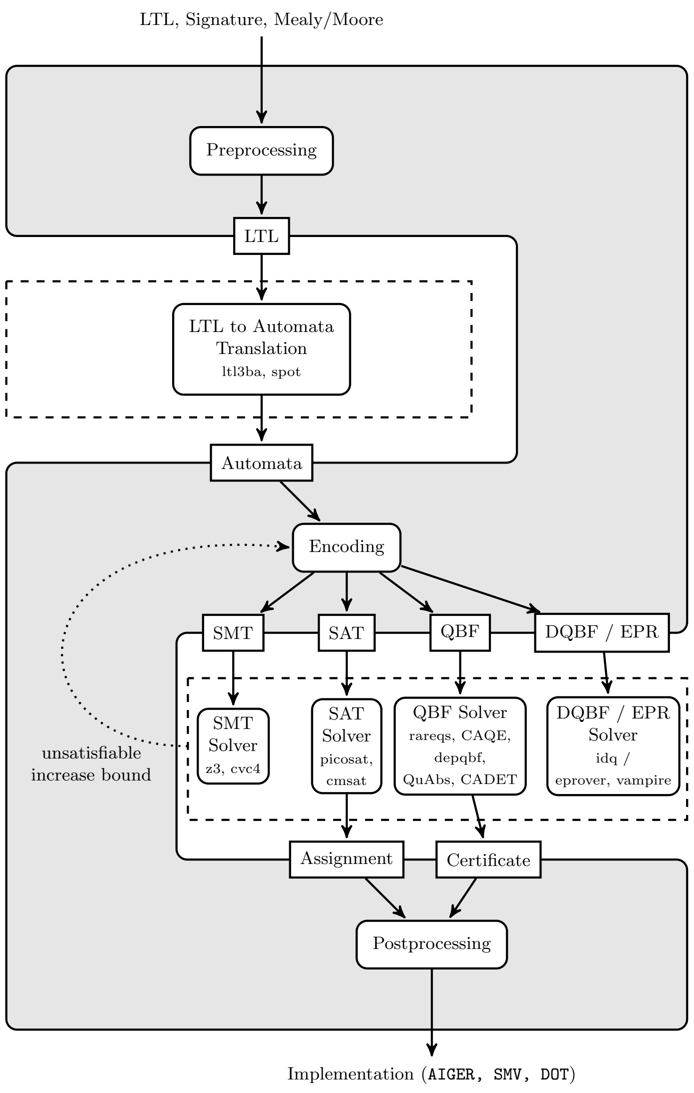

Fig. 1: Tool Architecture of BoSy

To the resulting automaton, we apply some basic optimization steps like replacing rejecting terminal states with safety conditions and an analysis of strongly connected components to reduce the size of the constraint system [12].

The encoding component is responsible for creating the constraint system based on the selected encoding, the specification automaton, and the current bound on the number of states of the system. The component constructs a constraint system using a logic representation which supports propositional logic, different kinds of quantification, and comparison operators (between natural numbers). Our implementation contains the following encoding options. These encodings differ in their ability to support the symbolic encoding of the existence of functions. We refer the reader to [10] for details.

- A *propositional* backend for SAT, where all functions are unrolled to conjunctions over their domain.
- An *input-symbolic* encoding employing QBF solvers, where functions with one application context are symbolically represented.
- Two encodings (*state-symbolic*, *symbolic*) using DQBF/EPR solvers, where functions, which are used in multiple contexts, are encoded symbolically.
- An *SMT* encoding resembling the original bounded synthesis encoding [12].

This constraint system is then translated to a format that the selected solver understands, and the solver is called as an external tool. We support SAT solvers that accept the DIMACS input format and that can output satisfying assignments, currently PicoSAT [3] and CryptoMiniSat [24]. We have three categories of QBF solving tools: QDIMACS/QCIR solver that can output top-level assignments (RAReQS [16], CAQE [21], and DepQBF [18]), QDIMACS preprocessors (Bloqqer [4]), and certifying QDIMACS/QCIR solver that can provide boolean functions witnessing satisfiable queries (QuAbS [25], CADET [20], and CAQE [21]). For the remaining formats, i.e., DQDIMACS (iDQ [13]), TPTP3, and SMT (Z3 [19], CVC4 [2]), we only require format conformance as witness extraction is not supported, yet.

After the selected solver with corresponding encoding has finished processing the query and reports *unsatisfiable*, the *search strategy* determines how the bound for the next constraint encoding is increased. Currently, we have implemented a linear and an exponential search strategy. In case the solver reports *satisfiable*, the implementation will be extracted in the postprocessing component. The extraction depends on the encoding and solver support, we currently support it for SAT and QBF.

In case of the encoding to SAT, the solver delivers an assignment, which is then translated to our representation of the synthesized implementation. The transition function and the functions computing the outputs are represented as circuits.

In case of the QBF-encoding, we take a two-step approach for synthesis. The QBF query has the quantifier prefix $\exists\forall\exists$ [10]. In synthesis mode, the query is solved by a combination of QBF preprocessor and QBF solver. From a satisfiable query, the assignment of the top-level existential quantification is extracted [23] and then used to reduce the original query by eliminating the top level existential quantifier. The resulting query, now with a $\forall\exists$ prefix, is then solved using a certifying QBF solver that returns a certificate, that is a circuit representing the witnessing boolean functions. This certificate is then translated into the same functional representation of the synthesized implementation as in the SAT case.

This common representation of the implementations allows the translation into different output formats. From our representation, it is possible to translate the implementation into an AIGER circuit as required by the SYNTCOMP rules, to a SMV model for model checking, or to a graphical representation using the DOT format.

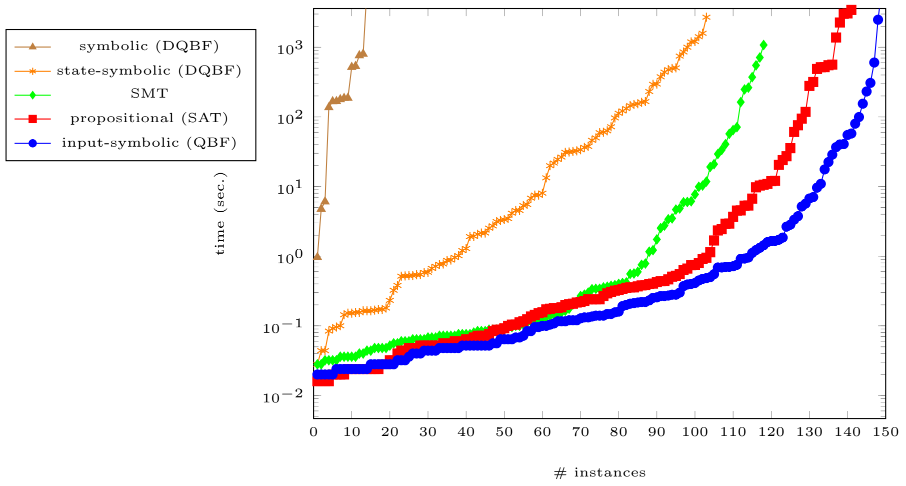

Fig. 2: Number of solved instances within 1 hour among the 195 instances from SYNTCOMP 2016. The time axis has logarithmic scale. The experiment was run on a machine with a 3.6 GHz quad-core Intel Xeon processor.

Further encodings can be integrated as extra components in the tool. Such an encoder component has access to the automaton, the semantics, and the input and output signals. The encoder must provide a method `solve` that takes as its parameter a bound and returns whether there is an implementation with the given bound that realizes the specification. The method is implemented by building the constraint system and solving it using a theory solver. One can either use our logic representation or build the textual solver representation directly. If the component supports synthesis, it implements a second method `extractSolution` that is called if `solve` returns true. It returns a representation of the realizing implementation as described above. In order to integrate new solver formats, one has to provide a translator from our logic representation to this format.

## 3 Experimentation

The reactive synthesis competition has a library of LTL benchmarks that can be transformed into the BoSy file format using the organizers' conversion tool [15]. The tool runsolver [22] is used in our experiments to get predictable timing results and to set appropriate time and memory limits. Figure 2 compares the performance of the different encodings for determining realizability on the SYNTCOMP benchmark set. Notably, the encoding employing QBF solving performed better than the SAT-based one and both solve more instances than the original SMT encoding. The two encodings using DQBF are not yet competitive due to limited availability of DQBF solvers.

Different configurations of BoSy may not only result in varying running times, but may also effect the *quality* of the synthesized implementation. A measure-

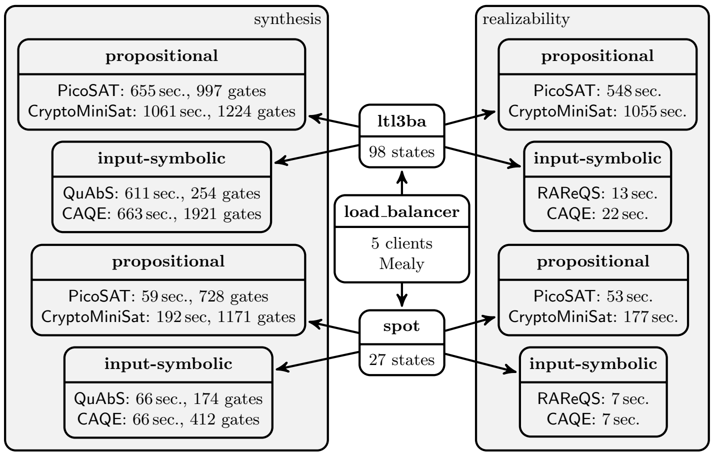

Fig. 3: The diagram shows the result on solving time and implementation quality for different configurations of BoSy on a sample specification.

ment of the quality of the implementation that has also been used in the reactive synthesis competition is the size of the implementation, measured in the number of gates in the circuit representation. The correctness of an implementation, i.e., whether the synthesized solution actually satisfies the original specification can be verified by model checking. One can either encode the solution as a circuit and use an AIGER model checker, or one can use the SMV representation and model check it with NuSMV [7].

For a sample benchmark, a parametric load-balancer [9] instantiated with 5 clients, we provide experimental results for different configurations of BoSy in Fig. 3. The two automaton conversion tools produce significantly different automata, where the state space of the automaton produced by spot is only one third of the one produced by ltl3ba. Consequently, the constraint system generated for the ltl3ba version is larger and thereby the running time worse compared to the spot version. This impact is stronger on the propositional encoding than on the input-symbolic one for realizability. An observation that also translates to other benchmarks is that the size of the implementation is usually smaller using the input-symbolic encoding. On the other hand, extracting solutions is cheaper in the propositional case as only assignments are extracted.

## References

1. Babiak, T., Kretínský, M., Rehák, V., Strejcek, J.: LTL to Büchi automata translation: Fast and more deterministic. In: Proceedings of TACAS. LNCS, vol. 7214, pp. 95–109. Springer (2012)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of CAV. LNCS, vol. 6806, pp. 171–177. Springer (2011)
3. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)

4. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Proceedings of CADE-23. LNCS, vol. 6803, pp. 101–115. Springer (2011)
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 652–657. Springer (2012)
6. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Journal of Symbolic Logic 28(4), 289–290 (1963)
7. 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: Proceedings of CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002)
8. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and $\omega$-automata manipulation. In: Proceedings of ATVA. LNCS, vol. 9938, pp. 122–129 (2016)
9. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: TACAS. Lecture Notes in Computer Science, vol. 6605, pp. 272–275. Springer (2011)
10. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Proceedings of TACAS. LNCS, vol. 10205, pp. 354–370. Springer (2017)
11. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: CAV. Lecture Notes in Computer Science, vol. 5643, pp. 263–277. Springer (2009)
12. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT 15(5-6), 519–539 (2013)
13. Fröhlich, A., Kovásznai, G., Biere, A., Veith, H.: iDQ: Instantiation-based DQBF solving. In: Proceedings of POS@SAT. EPiC Series in Computing, vol. 27, pp. 103–116. EasyChair (2014)
14. Jacobs, S., Bloem, R., Brenguier, R., Khalimov, A., Klein, F., Könighofer, R., Kreber, J., Legg, A., Narodytska, N., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The 3rd reactive synthesis competition (SYNTCOMP 2016): Benchmarks, participants & results. In: Proceedings of SYNT@CAV. EPTCS, vol. 229, pp. 149–177 (2016)
15. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Proceedings of SYNT@CAV. EPTCS, vol. 229, pp. 112–132 (2016)
16. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. Artif. Intell. 234, 1–25 (2016)
17. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD. pp. 117–124. IEEE Computer Society (2006)
18. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. JSAT 7(2-3), 71–76 (2010)
19. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
20. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Proceedings of SAT. LNCS, vol. 9710, pp. 375–392. Springer (2016)
21. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Proceedings of FMCAD. pp. 136–143. IEEE (2015)
22. Roussel, O.: Controlling a solver execution with the runsolver tool. JSAT 7(4), 139–144 (2011)
23. Seidl, M., Könighofer, R.: Partial witnesses from preprocessed quantified boolean formulas. In: Proceedings of DATE. pp. 1–6. European Design and Automation Association (2014)
24. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Proceedings of SAT. LNCS, vol. 5584, pp. 244–257. Springer (2009)
25. Tentrup, L.: Solving QBF by abstraction. CoRR abs/1604.06752 (2016)