PARTY
Parameterized Synthesis of Token Rings*

Ayrat Khalimov, Swen Jacobs, and Roderick Bloem

Graz University of Technology, Austria

Abstract. Synthesis is the process of automatically constructing an implementation from a specification. In parameterized synthesis, we construct a single process such that the distributed system consisting of an arbitratry number of copies of the process satisfies a parameterized specification. In this paper, we present Party, a tool for parameterized synthesis from specifications in indexed linear temporal logic. Our approach extends SMT-based bounded synthesis, a flexible method for distributed synthesis, to parameterized specifications. In the current version, Party can be used to solve the parameterized synthesis problem for token-ring architectures. The tool can also synthesize monolithic systems, for which we provide a comparison to other state-of-the-art synthesis tools.

1 Introduction

Synthesis methods and tools have received increased attention in recent years, as suitable solutions have been found for several synthesis tasks that have long been considered intractable. Although current tools have made large strides in efficiency, the run time of synthesis tools and the size of the resulting system still depend strongly on the size of the specification. In the case of parameterized specifications, this is particularly noticeable, and often unnecessary.

Bounded synthesis [13] is a method for solving the LTL synthesis problem by considering finite-state implementations with bounded resources. The space of all possible implementations is explored by iteratively increasing the bound. While several tools for LTL synthesis are based on variants of this approach [7,3], none of the available tools support distributed or parameterized synthesis.

In this paper, we introduce Party, the first tool that implements parameterized synthesis [14], based on the original SMT-based approach to bounded synthesis. Using cutoff results from parameterized verification [10], parameterized synthesis problems are reduced to distributed synthesis problems, and solved by bounded synthesis. While mainly intended to solve parameterized synthesis problems, Party can also be used for standard (monolithic) synthesis tasks.

Party is available at `https://github.com/5nizza/Party`. In the following sections, we present the background and implementation details of Party, and experimental results comparing Party to existing synthesis tools.

* This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) under the RiSE National Research Network (S11406)

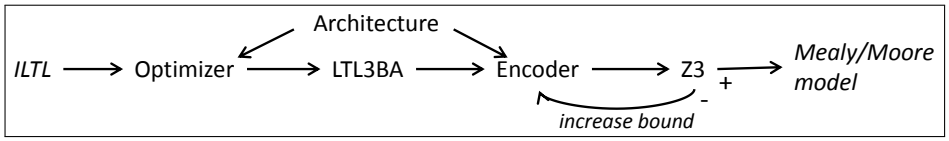

Fig. 1: The PARTY high-level control flow.

2 Background: Parameterized Synthesis

We use an approach that reduces the problem of LTL synthesis to a sequence of first-order satisfiability problems. This reduction, called *Bounded Synthesis*, has been introduced by Schewe and Finkbeiner [13] for the case of distributed systems with a fixed number of finite-state components. It is based on a translation of the specification into a universal co-Büchi automaton, followed by the generation of a first-order constraint that quantifies over an unbounded number of states. The existence of a solution to this constraint is equivalent to the realizability of a system accepted by the automaton, and thus synthesis is reduced to a satisfiability problem. To make this problem decidable, an additional bound on the size of the implementation is asserted, resulting in a semi-decision procedure for the problem by considering increasing bounds.

Based on results by Emerson and Namjoshi [10] on model checking parameterized token rings, Jacobs and Bloem [14] extended the bounded synthesis method to parameterized systems, consisting of an arbitrary number of isomorphic processes. Depending on the syntactic form of the specification, synthesis of parameterized token rings can be reduced to synthesis of isomorphic processes in small rings, and the resulting implementations are guaranteed to satisfy the specification in rings of arbitrary size. The sufficient size of the ring is called the *cutoff* and should not be confused with the *bound* on size of individual processes.

In previous work [16], we used results by Clarke et al. [5] to extend the applicability of this approach, and generalized optimizations from parameterized or distributed verification to synthesis methods, to improve its efficiency in practice.

While the approaches above have been implemented in a prototype before [16], in PARTY we have added several features such as the synthesis of Mealy machines, as well as the synthesis of non-parameterized, monolithic, systems. Also, we have refactored our implementation for usability, including an input language that is derived from the language used by Acacia+.

3 Tool Description

The high-level control flow of PARTY is given in Fig. 1.

Input. Specifications consist of four parts: inputs, outputs, assumptions and guarantees. Assumptions and guarantees are in ILTL and may contain universal quantifiers at the beginning of the expression. This simple structure of properties, $\forall i.\ A_i \rightarrow \forall j.\ G_j$, is enough to model all examples we have considered

thus far. The format of the language is very similar to that of Acacia+; an example specification is given in Listing 1. Note that, although specifications of parameterized token rings are in ILTL\X, PARTY supports a X operator whose semantics is local to a given process [16].

```
[INPUT_VARIABLES]
r;
[OUTPUT_VARIABLES]
g;
[ASSUMPTIONS]
Forall (i) r_i=0;
Forall (i) G(((r_i=1)*(g_i=0)->X(r_i=1)) *
             ((r_i=0)*(g_i=1)->X(r_i=0)));
Forall (i) G(F((r_i=0)+(g_i=0)));
[GUARANTEES]
Forall (i) g_i=0;
Forall (i,j) G(!((g_i=1) * (g_j=1)));
Forall (i) G((((r_i=0)*(g_i=0))->X(g_i=0)) *
             (((r_i=1)*(g_i=1))->X(g_i=1)));
Forall (i) G(F(((r_i=1)*(g_i=1)) +
               ((r_i=0)*(g_i=0))));
```

Listing 1: Specification of parameterized Pnueli arbiter.

Optimizer. *Optimizer* takes as input an ILTL specification and an architecture (currently *token-ring* or *monolithic*). It adds domain-specific environment assumptions, such as fair scheduling, to the specification and optimizes it according to user provided option `-opt (no, strength, async_hub, sync_hub)`. Then, Optimizer identifies the cutoff of the specification and instantiates it accordingly. The instantiated specification is in LTL without quantifiers, and can be synthesized with an adapted version of the bounded synthesis approach.

Encoder, Z3 SMT Solver. The LTL specification obtained from Optimizer is translated into an automaton by LTL3BA [1]. The result is passed to *Encoder*, together with the architecture. Currently Encoder supports monolithic architectures and parameterized token rings, but it can be extended to other parameterized architectures and general distributed synthesis.

Given a bound on the implementation, Encoder generates an SMT query in AUFLIA logic. This query is fed to solver Z3 [6] for a satisfiability check. If the SMT query for a given bound is unsatisfiable, control is returned to the Encoder who increases the bound, encodes a new query and feeds it to the solver. If the query is satisfiable, the model is converted into a Mealy or a Moore machine.

Output. For realizable inputs, PARTY outputs a Mealy or Moore machine in dot or nusmv format. NuSMV v2.5.4 [4] can be used for model checking.

Implementation. PARTY is written in python3 (5k lines of code) and tested on Ubuntu 12.04. Python does not introduce a significant overhead since the most computationally expensive parts are done in LTL3BA and Z3.

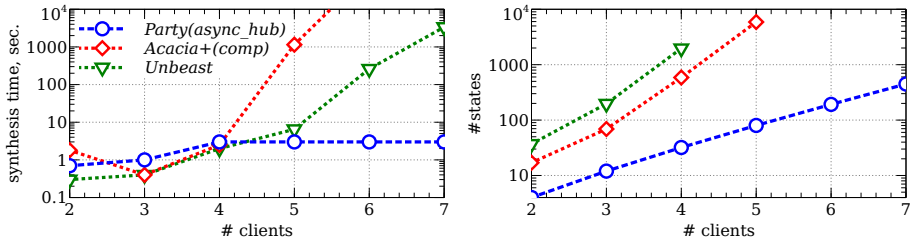

Fig. 2: Synthesis time and model size on a parameterized full arbiter example[16]. The tools demonstrate similar behavior on 'x_arbiter' example and on 'Pnueli' arbiter (but in the latter case async_hub optimization is not complete).

4 Experiments

We compare PARTY with Acacia+(v2.1) and UNBEAST (v0.6b) on parameterized and several monolithic examples.

Parameterized benchmarks. To test parameterized part of PARTY, we use several arbiter specifications: *full* arbiter [16], *'Pnueli'* arbiter in GR(1) style, *x_arbiter* that uses the local next operator, and variants of these.

Fig. 2 shows the efficiency of parameterized synthesis compared to the standard approach, for 'full' arbiter with a parametric number of clients. Starting with 6 clients, PARTY outperforms the other tools by orders of magnitude. The other tools can only generate arbiters with 5 or 6 clients within one hour, whereas the parameterized approach can stop after synthesizing a token ring of cutoff size 4, and clone the resulting process model to form a ring of any larger size.

The right-hand side of the figure shows implementation size with increasing number of clients. Models generated by PARTY are several orders of magnitude smaller than others and grows less steeply with increasing the number of clients.

Monolithic benchmarks. For the comparison we use realizable benchmarks from tool Lily (3,5-10,12-23) [15], load balancer(*lb2-lb4*) [7], and genbuf benchmark(*gb2*)[2]. Table 1 compares times of PARTY with times of Acacia+/UNBEAST. All models synthesized by PARTY were model checked with NuSMV [4].

It is difficult to provide a fair comparison due to different semantics of system models. For example, Acacia+ outputs Moore machines, UNBEAST Mealy machines as NuSMV models, and PARTY supports both. Also, UNBEAST has its own xml-based input format and does not provide a converter from other formats. Therefore, we could not run UNBEAST on *gb2* benchmark.

The difficulty of fair comparison is reflected by Table 1. The tools were run with their default parameters. UNBEAST was run with extracting models option. Acacia+ and PARTY were provided with two different specifications: in Moore semantics (the system moves first), and in Mealy semantics (the environment moves first). In the second case PARTY generated Mealy models, while Acacia+ still generated Moore-like models. The table shows that UNBEAST outperforms other tools in terms of synthesis time, and models in PARTY are the smallest.

Table 1: Comparison of PARTY monolithic with Acacia+ and UNBEAST (t/o=1h). Numbers in parenthesis mean the size of implementation[1]

	lily	lily16	lb2	lb3	lb4	gb2
UNBEAST(Mealy)	4(540/160)	0.1(55/54)	0.2(25/11)	1(576/33)	13(m/o)	-
Acacia+(Mealy)	12(80)	1(15)	2(10)	1(42)	54(145)	-
PARTY(Mealy)	22(33)	35(6)	1(1)	3(2)	139(3)	-
Acacia+(Moore)	7(61)	1(15)	1(7)	4(14)	1639(41)	1(49)
PARTY(Moore)	12(40)	1526(8)	1(3)	364(6)	t/o	t/o

A detailed analysis shows that PARTY spends most of the time in SMT solving, where in turn proving unsatisfiability for bounds that are too small takes most of the time. For example, the synthesis time for *lily16* is 25 minutes if we explore all model sizes starting with 1. But if we force PARTY to search the model of exact size 8, the solution is found in 2 minutes. This means that we need to explore incremental solving and other methods to avoid long unsatisfiability checks.

Incremental solving. PARTY supports two incremental approaches: for increasing size of rings (parameterized architectures only), and for increasing bounds.

Instead of searching for a model in a token ring of the cutoff size, we can use even smaller rings and then check if the result satisfies the specification in a ring of sufficient size. Preliminary experiments demonstrate the effectiveness of this approach: the 'Pnueli' arbiter can be synthesized in 50 seconds using the usual approach vs. 15 seconds using the incremental approach.

To handle increasing bounds, we can use incrementality of SMT solvers when unsatisfiability for the current bound is detected. When the bound is increased, we pull constraints directly relating to the old bound, and push new ones. On most benchmarks, this approach is comparable to non-incremental one, but on some examples it is much faster: full3: 506 seconds (orig) vs. 140 seconds (incr).

5 Conclusions

We presented PARTY, a tool for parameterized synthesis. In the current version, the tool can synthesize Mealy and Moore machines as process implementations in monolithic architectures or parameterized token-ring architectures. For the latter case, it implements optimizations that speed up synthesis by several orders of magnitude. The input language is derived from languages of existing tools, supports full LTL for monolithic and ILTL\X for parameterized architectures.

Besides the fact that this is the first implementation of parameterized synthesis, an experimental comparison to other synthesis tools has been difficult

[1] UNBEAST model sizes are calculated by NuSMV with `-r` option as suggested by Rüdiger Ehlers. Two sizes are given: for default model extraction method, and for a learning based method [9]. On *lb4* NuSMV crashed with a memory allocation error.

because tools are often specialized to a subclass of problems, and use different input languages (see also Ehlers [8]). It may be worthwhile to discuss standards for languages and subclasses of synthesis problems, much like in other automated reasoning communities, e.g., represented by the SMT-LIB initiative.

PARTY is designed modularly, and we are working on several extensions. Most importantly, parameterized synthesis can be extended to other architectures that allow automatic detection of cutoffs [5,11]. Also, we plan to further increase efficiency of the synthesis process, either by more high-level optimizations or by integration of verification techniques, like the lazy synthesis approach [12].

References

1. Babiak, T., Kretínský, M., Rehák, V., Strejcek, J.: LTL to büchi automata translation: Fast and more deterministic. In: Flanagan, C., König, B. (eds.) TACAS. LNCS, vol. 7214, pp. 95–109. Springer (2012)
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. ENTCS 190(4), 3–16 (2007)
3. Bohy, A., Bruyre, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S. (eds.) CAV. LNCS, vol. 7358, pp. 652–657. Springer (2012)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking (July 2002)
5. Clarke, E.M., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR. LNCS, vol. 3170, pp. 276–291. Springer (2004)
6. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
7. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P., Leino, K. (eds.) TACAS, LNCS, vol. 6605, pp. 272–275. Springer (2011)
8. Ehlers, R.: Experimental aspects of synthesis. In: Proceedings of the International Workshop on Interactions, Games and Protocols. EPTCS, vol. 50 (2011)
9. Ehlers, R., Könighofer, R., Hofferek, G.: Symbolically synthesizing small circuits. In: Cabodi, G., Singh, S. (eds.) FMCAD. pp. 91–100. IEEE (2012)
10. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
11. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D.A. (ed.) CADE. LNCS, vol. 1831, pp. 236–254. Springer (2000)
12. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI. LNCS, vol. 7148, pp. 219–234. Springer (2012)
13. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer pp. 1–21 (2012)
14. Jacobs, S., Bloem, R.: Parameterized synthesis. In: Flanagan, C., König, B. (eds.) TACAS. LNCS, vol. 7214, pp. 362–276. Springer (2012)
15. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD. pp. 117–124. IEEE Computer Society (2006)
16. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI. LNCS, vol. 7737. Springer (2013)