Towards Efficient Parameterized Synthesis*

Ayrat Khalimov, Swen Jacobs, and Roderick Bloem

Graz University of Technology, Austria

Abstract. Parameterized synthesis was recently proposed as a way to circumvent the poor scalability of current synthesis tools. The method uses cut-off results in token rings to reduce the problem to bounded distributed synthesis, and thus ultimately to a sequence of SMT problems. This solves the problem of scalability in the size of the architecture, but experiments show that the size of the specification is still a major issue. In this paper we propose several optimizations of the approach. First, we tailor the SMT encoding to systems with isomorphic processes and token-ring architecture. Second, we extend the cut-off results for token rings and refine the reduction, using modularity and abstraction techniques. Some of our optimizations also apply to isomorphic or distributed synthesis in arbitrary architectures. To evaluate these optimizations, we developed the first completely automatic implementation of parameterized synthesis. Experiments show a speed-up of several orders of magnitude, compared to the original method.

1 Introduction

By automatically generating correct implementations from a temporal logic specification, reactive synthesis tools relieve system developers from manual low-level implementation and debugging. However, existing tools are not very scalable. For instance, Bloem et al. [3] describe the synthesis of an arbiter for the ARM AMBA Advanced High Performance Bus. The results, obtained using RATSY [2], show that both the size of the implementation and the time for synthesis increase steeply with the number of clients that the arbiter can handle. Since an arbiter for $n+1$ clients is very similar to an arbiter for n clients, this is unexpected.

Similar to the AMBA arbiter, many other specifications in verification and synthesis are naturally parameterized in the number of parallel interacting components [15,14]. To address the poor scalability of reactive synthesis tools in the number of components, Jacobs and Bloem [13] introduced a parameterized synthesis approach. A simple example of a parameterized specification is the following LTL specification of a simple arbiter:

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i) \end{array}$$

* 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) through the national research network RiSE (S11406).

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 108–127, 2013.
© Springer-Verlag Berlin Heidelberg 2013

In parameterized synthesis, we synthesize a building block that can be cloned to form a system that satisfies such a specification, for any number of components.

Jacobs and Bloem [13] showed that parameterized synthesis is undecidable in general, but semi-decision procedures can be found for classes of systems with cut-offs, i.e., where parameterized verification can be reduced to verification of a system with a bounded number of components. They presented a semi-decision procedure for token-ring networks, building on results by Emerson and Namjoshi [8], which show that for the verification of parameterized token rings, a cut-off of 5 is sufficient for a certain class of specifications. Following these results, parameterized synthesis reduces to distributed synthesis in token rings of (up to) 5 identical processes. To solve the resulting problem, a modification of the SMT encoding of the distributed bounded synthesis problem [12] was used.

Experiments with the parameterized synthesis method [13] revealed that only very small specifications could be handled with this encoding. For example, the simple arbiter presented in the beginning can be synthesized in a few seconds for a ring of size 4, which is the sufficient cut-off for this specification. However, synthesis does not terminate within 2 hours for a specification that also excludes spurious grants, in a ring of the same size. Furthermore, the previously proposed method uses cut-off results of Emerson and Namjoshi [8] and therefore inherits a restricted language support and cannot handle specifications in assume-guarantee style [3].

Contributions. We propose several optimizations for the parameterized synthesis method, and extend the supported specification language. Some of the optimizations apply to the bounded synthesis approach in general, others only to special cases like isomorphic synthesis, or synthesis in token rings. In particular, we introduce two kinds of optimizations:

1. We revisit the previously proposed SMT encoding and tailor it to systems with isomorphic processes and token ring architecture. We use a *bottom-up approach* that models the global system as a product of isomorphic processes, in contrast to the top-down approach used before. Also, we encode particular properties of token rings more efficiently by *restricting the class of solutions* without losing completeness.
2. We consider optimizations that are independent of the SMT encoding. This includes *incremental solving*, i.e. generating isomorphic processes in small rings and testing the result in bigger rings. Furthermore, we use *modular generation of synthesis constraints* when different classes of properties can be encoded in token rings of different sizes. For local properties (that require a ring of size two), we introduce an *abstraction* that replaces the second process with assumptions on its behavior, thus reducing the ring of size two to a single process with additional assumptions on its environment. Finally, we show how to simplify specifications by *strengthening*, e.g. removing liveness assumptions from parts that specify safety properties.

For some of the examples we consider in this paper, these optimizations show a speedup of at least 3 orders of magnitude, which means that we need seconds where we would otherwise need hours.

To allow the reader to assess the ideas behind our optimizations, and their correctness, we present in some detail the background of distributed synthesis in Sect. 2, and parameterized synthesis with the original SMT encoding in Sect. 3. In Sect. 4 we introduce optimizations of the SMT encoding. In Sect. 5, we consider extensions of our language, which allow us to support a broader class of specifications. Sect. 6 introduces more general optimizations that are independent of the encoding, and finally Sect. 7 gives experimental results.

2 Preliminaries

We consider the synthesis problem for distributed systems, with specifications in (fragments of) LTL. Given a system architecture A and a specification φ, we want to find implementations of all system processes in A, such that their composition satisfies φ.

Architectures. An *architecture* A is a tuple (P, env, V, I, O), where P is a finite set of processes, containing the environment process env and system processes $P^- = P \setminus \{env\}$, V is a set of Boolean system variables, $I = \{I_i \subseteq V \mid i \in P^-\}$ assigns a set I_i of Boolean input variables to each system process, and $O = \{O_i \subseteq V \mid i \in P\}$ assigns a set O_i of Boolean output variables to each process, such that $\dot{\bigcup}_{i \in P} O_i = V$. In contrast to output variables, inputs may be shared between processes. Wlog., we use natural numbers to refer to system processes, and assume $P^- = \{1, \dots, k\}$ for an architecture with k system processes.

Implementations. An *implementation* $\mathcal{T}_i$ of a system process i with inputs I_i and outputs O_i is a labeled transition system (LTS) $\mathcal{T}_i = (T_i, t_i, \delta_i, o_i)$, where T_i is a set of states including the initial state t_i, $\delta_i : T_i \times \mathcal{P}(I_i) \to T_i$ a transition function, and $o_i : T_i \to \mathcal{P}(O_i)$ a labeling function.

The *composition* of a set of implementations $\{\mathcal{T}_1, \dots, \mathcal{T}_k\}$ is the LTS $\mathcal{T}_A = (T_A, t_0, \delta, o)$, with $T_A = T_1 \times \dots \times T_k$, initial state $t_0 = (t_1, \dots, t_k)$, labeling function $o : T_A \to \mathcal{P}(\dot{\bigcup}_{1 \le i \le k} O_i)$ with $o(t_1, \dots, t_k) = o_1(t_1) \cup \dots \cup o_k(t_k)$, and transition function $\delta : T_A \times \mathcal{P}(O_{env}) \to T_A$ with

$$\delta((t_1, \dots, t_k), e) = (\delta_1(t_1, (o(t_1, \dots, t_k) \cup e) \cap I_1), \dots, \delta_k(t_k, (o(t_1, \dots, t_k) \cup e) \cap I_k)),$$

i.e., every process advances according to its own transition function and input variables, where inputs from other system processes are interpreted according to the labeling of the current state.

Asynchronous Systems. An *asynchronous system* is an LTS such that in every transition, only a subset of the system processes changes their state. This is decided by a *scheduler*, which determines for every transition the processes that are allowed to make a step. We assume that the environment is always scheduled, and the scheduler is a part of the environment. Formally, O_{env} contains additional scheduling variables $s_1, \dots, s_k$, and $s_i \in I_i$ for every i. We require $\delta_i(t, I) = t$ for any i and set of inputs I with $s_i \notin I$.

Token Rings. We consider a class of architectures called *token rings*, where processes can only communicate by passing a token. At any time only one process

can possess the token, and in a ring of size k, a process i which has the token can pass it to process $i+1$ mod k by raising an output $\mathsf{send}_i \in O_i \cap I_{i+1}$. We assume that token rings are implemented as asynchronous systems, where in every step only one system process may change its state, except for token-passing steps, in which both of the involved processes change their state.

Distributed Synthesis. The *distributed synthesis problem* for an architecture A and specification φ, is to find implementations for the system processes of A, such that the composition of the implementations $\mathcal{T}_1, \dots, \mathcal{T}_k$ satisfies φ, written $A, (\mathcal{T}_1, \dots, \mathcal{T}_k) \models \varphi$. Specification φ is *realizable* with respect to an architecture A if such implementations exist. Synthesis and checking realizability of LTL specifications have been shown to be undecidable for architectures in which not all processes have the same information wrt. environment outputs [10].

Bounded Synthesis. The *bounded synthesis problem* for given architecture A, specification φ and a family of bounds $\{b_i \in \mathbb{N} \mid i \in P^-\}$ on the size of system processes, as well as a bound b_A for their composition $\mathcal{T}_A$, is to find implementations $\mathcal{T}_i$ for the system processes such that their composition $\mathcal{T}_A$ satisfies φ, with $|\mathcal{T}_i| \leq b_i$ for all process implementations, and $|\mathcal{T}_A| \leq b_A$.

Automata. A *universal co-Büchi tree automaton (UCT)* is a tuple $\mathcal{U} = (\Sigma, \Upsilon, Q, \rho, \alpha)$, where Σ is a finite output alphabet, Υ is a finite input alphabet, $\rho : Q \times \Sigma \times \Upsilon \to \mathcal{P}(Q)$ is the transition relation, and α is the set of rejecting states. We call $\mathcal{U}$ a *one-letter* UCT if $|\Sigma| = |\Upsilon| = 1$. A one-letter UCT is *accepting* if all its paths visit α only finitely often. The *run graph* of a UCT $\mathcal{U}$ on an implementation $\mathcal{T}$ is a one-letter UCT obtained by taking the usual synchronous product and replacing all labels by an arbitrary one. An implementation is accepted if its run graph is accepting. The language of $\mathcal{U}$ consists of all accepted implementations. In the graph defined by Q and δ, we call a strongly connected component (SCC) *accepting* (*rejecting*) if it does not (does) contain states in α.

3 Parameterized Synthesis

In this section we recapitulate the method for parameterized synthesis introduced by Jacobs and Bloem [13].

Parameterized Architectures and Specifications. Let $\mathcal{A}$ be the set of all architectures. A *parameterized architecture* is a function $\Pi : \mathbb{N} \to \mathcal{A}$. A *parameterized token ring* is a parameterized architecture R with
$R(n) = (P^n, env, V^n, I^n, O^n)$, where

- $P^n = \{env, 1, \dots, n\}$,
- I^n assigns to each process a set I_i of isomorphic inputs. That is, for some I, I_i consists of the inputs in I subscripted with i. Additionally, I_i contains the token-passing input send_{i-1} from process $i-1$ (mod n).
- Similarly, O^n assigns isomorphic, indexed sets of outputs to all system processes, with $\mathsf{send}_i \in O_i$, and every output of env is indexed with all values from 1 to n.

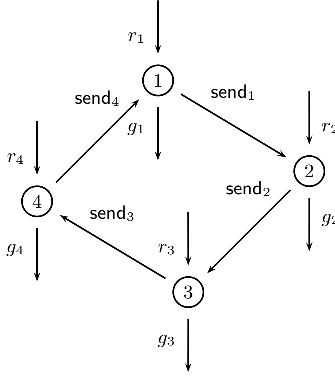

Fig. 1. Token ring with 4 processes

A *parameterized specification* φ is an LTL specification with indexed variables, and universal quantification over indices. We say that a parameterized architecture Π and a process implementation $\mathcal{T}$ *satisfy* a parameterized specification (written $\Pi, \mathcal{T} \models \varphi$) if for any n, $\Pi(n), (\mathcal{T}, \ldots, \mathcal{T}) \models \varphi$.

Example 1. Consider the parameterized token ring R_{arb} with $R_{arb}(n) = (P^n, env, V^n, I^n, O^n)$, where

$$\begin{aligned} P^n &= \{env, 1, \ldots, n\} \\ V^n &= \{r_1, \ldots, r_n, g_1 \ldots, g_n, \mathsf{send}_1, \ldots, \mathsf{send}_n\} \\ I_i &= \{r_i, \mathsf{send}_{i-1}\} \\ O_{env} &= \{r_1, \ldots, r_n\} \\ O_i &= \{g_i, \mathsf{send}_i\} \end{aligned}$$

The architecture $R_{arb}(n)$ defines a token ring with n system processes, with each process i receiving an input r_i from the environment and another input send_{i-1} from the previous process in the ring, and an output send_i to the next process, as well as an output g_i to the environment.

An instance of this parameterized architecture for $n = 4$ is depicted in Fig. 1, and the following is the parameterized specification from the introduction:

$$\begin{aligned} &\forall i \neq j. \quad \mathsf{G}\,\neg(g_i \wedge g_j) \\ &\forall i. \qquad \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i). \end{aligned}$$

Isomorphic and Parameterized Synthesis. The *isomorphic synthesis problem* for an architecture A and a specification φ is to find an implementation $\mathcal{T}$ for all system processes $(1, \ldots, k)$ such that $A, (\mathcal{T}, \ldots, \mathcal{T}) \models \varphi$, also written $A, \mathcal{T} \models \varphi$. The *parameterized synthesis problem* for a parameterized architecture Π and a parameterized specification φ is to find an implementation $\mathcal{T}$ for all system processes such that $\Pi, \mathcal{T} \models \varphi$. The *parameterized (isomorphic) realizability problem* is the question whether such an implementation exists.

A *cut-off* for Π and φ is a number $k \in \mathbb{N}$ such that

$$\Pi(k), \mathcal{T} \models \varphi \quad \Rightarrow \quad \Pi(n), \mathcal{T} \models \varphi \text{ for all } n \geq k.$$

3.1 Reduction of Parameterized to Isomorphic Synthesis

Emerson and Namjoshi [8] have shown that verification of LTL\X properties for implementations of parameterized token rings can be reduced to verification of a small ring with up to five processes, depending on the form of the specification.

Theorem 1 ([8]). *Let R be a parameterized token ring, $\mathcal{T}$ an implementation of the isomorphic system processes that ensures fair token passing, and φ a parameterized specification. For a sequence $\bar{t}$ of index variables and terms in arithmetic modulo n, let $f(\bar{t})$ be a formula that only refers to system variables indexed by terms in $\bar{t}$. Then,*

$$R, \mathcal{T} \models \varphi \iff R(k), \mathcal{T} \models \varphi \text{ for } 1 \leq k \leq n,$$

where n is a cut-off depending on the specification: (a) if $\varphi = \forall i.\ f(i)$, then $n = 2$; (b) if $\varphi = \forall i.\ f(i, i+1)$, then $n = 3$, (c) if $\varphi = \forall i \neq j.\ f(i, j)$, then $n = 4$, and (d) if $\varphi = \forall i \neq j.\ f(i, i+1, j)$, then $n = 5$. [1]

Thus, verification of such structures is decidable. For synthesis, we obtain the following corollary:

Corollary 1 ([13]). *For a given parameterized token ring R and parametric specification φ, parameterized synthesis can be reduced to isomorphic synthesis in rings of size* 2 *(3, 4, 5) for specifications of type a) (b, c, d, resp.).*

Using a modification of undecidability proofs for the distributed synthesis problem [16,10], Jacobs and Bloem [13] showed undecidability of isomorphic synthesis in token rings, which implies undecidability of parameterized synthesis.

3.2 Bounded Isomorphic Synthesis

The reduction from Sect. 3.1 allows us to reduce parameterized synthesis to isomorphic synthesis with a fixed number of processes. To solve the resulting problem, bounded synthesis is adapted for isomorphic synthesis in token rings.

Bounded Synthesis. Following [12], the bounded synthesis procedure consists of three steps:

1. **Automata translation.** The LTL specification φ (including fairness assumptions like fair scheduling) is translated into a UCT $\mathcal{U}$ which accepts an LTS $\mathcal{T}$ iff $\mathcal{T}$ satisfies φ.
2. **SMT Encoding.** Existence of an LTS which satisfies φ is encoded into a set of SMT constraints over the theory of integers and free function symbols. States of the LTS are represented by natural numbers, state labels as free

[1] The results of [8] allow to fix one of the indices in the specification. For $\forall i.\ f(i)$ it is enough to verify the property $f(0)$ under the assumption that initially the token is given to a randomly chosen process. For $\forall i \neq j.\ f(i, j)$ it is enough to verify $\forall j \neq 0.\ f(0, j)$. See Lemma 3 in [8].

functions of type $\mathbb{N} \to \mathbb{B}$, and the global transition function as a free function of type $\mathbb{N} \times \mathbb{B}^{|O_{env}|} \to \mathbb{N}$. To obtain an interpretation of these symbols that satisfies the specification φ, we introduce labels $\lambda_q^{\mathbb{B}} : \mathbb{N} \to \mathbb{B}$ and free functions $\lambda_q^{\#} : \mathbb{N} \to \mathbb{N}$, which are defined such that (i) $\lambda_q^{\mathbb{B}}(t)$ is true iff the product of $\mathcal{T}$ and $\mathcal{U}$ contains a path from an initial state to a state (t, q) with $q \in Q$ and (ii) valuations of the $\lambda_q^{\#}$ must be non-decreasing along paths of $\mathcal{U}$, and strictly increasing for transitions that visit a rejecting state of $\mathcal{U}$. This ensures that an LTS satisfying these constraints cannot have runs which enter rejecting states infinitely often. The corresponding constraint for an UCT $(\Sigma, \Upsilon, Q, \rho, \alpha)$ and an implementation (T, t, δ, o) is

$$\bigwedge_{t} \bigwedge_{I} \bigwedge_{q,q'} \lambda_q^{\mathbb{B}}(t) \wedge q' \in \rho(q, o(t), I) \to \lambda_{q'}^{\mathbb{B}}(\delta(t, I)) \wedge \lambda_{q'}^{\#}(\delta(t, I)) \rhd_q \lambda_q^{\#}(t), \quad (1)$$

where $\rhd_q$ equals $>$ if $q \in \alpha$, and $\rhd_q$ equals $\geq$ otherwise. Furthermore, we add a constraint that $\lambda^{\mathbb{B}}$ holds in the initial state of the run graph.
Finally, transition functions of individual processes are defined indirectly by introducing projections $d_i : \mathbb{N} \to \mathbb{N}$, mapping global to local states. To ensure that local transitions of process i only depend on inputs in I_i, we add a constraint

$$\bigwedge_{i} \bigwedge_{t,t'} \bigwedge_{I,I'} d_i(t) = d_i(t') \wedge I \cap I_i = I' \cap I_i \to d_i(\delta(t, I)) = d_i(\delta(t', I')). \quad (2)$$

3. **Iteration for Increasing Bounds.** To obtain a decidable problem, the number of states in the LTS that we are looking for is bounded, which allows us to instantiate all quantifiers over state variables t, t' explicitly with all values in the given range. If the constraints are unsatisfiable for a given bound, we increase it and try again. If they are satisfiable, we obtain a model, giving us an implementation for the system processes such that φ is satisfied.

Adaption to Token Rings. The bounded synthesis approach is adapted for synthesis in token rings, along with some first optimizations for a better performance of the synthesis method.[2]

– We want to obtain an asynchronous system in which the environment is always scheduled, along with exactly one system process. In general, we could add a constraint $\bigwedge_i \bigwedge_I s_i \notin I \to d_i(\delta(t, I)) = d_i(t)$ (where I is a set of inputs and s_i is the scheduling variable for process i). For our case, we do not need $|P|$ scheduling variables, but can encode the index of the scheduled process into a binary representation with $log_2(|P^-|)$ inputs.
– We use the semantic variation where environment inputs are not stored in system states, but are directly used in the transition term that computes the following state (cp. [12], Sect. 8). This results in an implementation which is a factor of $|O_{env}|$ smaller.

[2] This includes modifications and optimizations mentioned in [12], as well as [13].

- We encode the special features of token rings: i) Exactly one process should have the token at any time; ii) Only a process which has the token can send it; iii) If process i is scheduled, currently has the token, and wants to send it, then in the next state process $i+1$ has the token and process i does not; iv) If process i has the token and does not send it (or is not scheduled), it also has the token in the next state, and v) if process i does not have the token and does not receive it from process $i-1$, then it will also not have the token in the next step. Properties ii) – v) are encoded in the following constraints, where $\mathsf{tok}_i(d_i(t))$ is true in state t iff process i has the token, $\mathsf{send}(d_i(t))$ is true iff i is ready to send the token, and $\mathsf{sched}_i(I)$ is true iff the scheduling variables in I are such that process i is scheduled:

$$\begin{array}{ll} \bigwedge_i \bigwedge_t \bigwedge_I & \mathsf{tok}(d_i(t)) \rightarrow (\mathsf{send}(d_i(t)) \wedge \mathsf{sched}_i(I)) \vee \mathsf{tok}(d_i(\delta(t,I))) \\ \bigwedge_i \bigwedge_t & \neg\mathsf{tok}(d_i(t)) \rightarrow \neg\mathsf{send}(d_i(t)) \\ \bigwedge_i \bigwedge_t \bigwedge_I & \mathsf{send}(d_i(t)) \wedge \mathsf{sched}_i(I) \rightarrow \neg\mathsf{tok}(d_i(\delta(t,I))) \\ \bigwedge_i \bigwedge_t \bigwedge_I & \mathsf{send}(d_{i-1}(t)) \wedge \mathsf{sched}_{i-1}(I) \rightarrow \mathsf{tok}(d_i(\delta(t,I))) \\ \bigwedge_i \bigwedge_t \bigwedge_I & \neg\mathsf{tok}(d_i(t)) \wedge \neg(\mathsf{send}(d_{i-1}(t)) \wedge \mathsf{sched}_{i-1}(I)) \rightarrow \neg\mathsf{tok}(d_i(\delta(t,I))). \end{array} \quad (3)$$

  We do not encode property i) directly, because it is implied by the remaining constraints whenever we start in a state where only one process has the token.
- Token passing is an exception to the rule that only the scheduled process changes its state: if process i is scheduled in state t, and both $\mathsf{tok}(d_i(t))$ and $\mathsf{send}(d_i(t))$ hold, then in the following transition both processes i and $i+1$ will change their state. The constraint which ensures that only scheduled processes may change their state is modified into

$$\begin{array}{ll} \bigwedge_i \bigwedge_t \bigwedge_I & \neg\mathsf{sched}_i(I) \wedge \neg(\mathsf{sched}_{i-1}(I) \wedge \mathsf{tok}(d_{i-1}(t)) \wedge \mathsf{send}(d_{i-1}(t))) \\ & \rightarrow d_i(\delta(t,I)) = d_i(t). \end{array} \quad (4)$$

- We use isomorphism constraints to encode that the processes are identical. To this end, we use the same function symbols for state labels of all system processes, and restrict local transitions such that they result in the same local state whenever the previous local states and the visible inputs are equivalent. Since our definition allows processes that are not scheduled to receive the token, we add a rule for this special case. The resulting constraints for local transitions are:

$$\begin{array}{ll} \bigwedge_{i>1} \bigwedge_{t,t'} \bigwedge_{I,I'} & d_1(t) = d_i(t') \wedge \mathsf{sched}_1(I) \wedge \mathsf{sched}_i(I') \wedge I \cap I_1 = I' \cap I_i \\ & \rightarrow d_1(\delta(t,I)) = d_i(\delta(t',I')) \\ \bigwedge_{i>1} \bigwedge_{t,t'} \bigwedge_{I,I'} & d_1(t) = d_i(t') \wedge \mathsf{send}(d_n(t)) \wedge \mathsf{send}(d_{i-1}(t')) \\ & \wedge\ \mathsf{sched}_n(I) \wedge \mathsf{sched}_{i-1}(I') \wedge I \cap I_1 = I' \cap I_i \\ & \rightarrow d_1(\delta(t,I)) = d_i(\delta(t',I')). \end{array} \quad (5)$$

- Finally, a precondition of Thm. 1 is that the implementation needs to ensure fair token-passing. Let $\mathsf{fair_scheduling}$ stand for $\bigwedge_j \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_j$. Then, we always add

$$\bigwedge_i (\mathsf{fair_scheduling} \rightarrow (\mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{send}_i))) \quad (6)$$

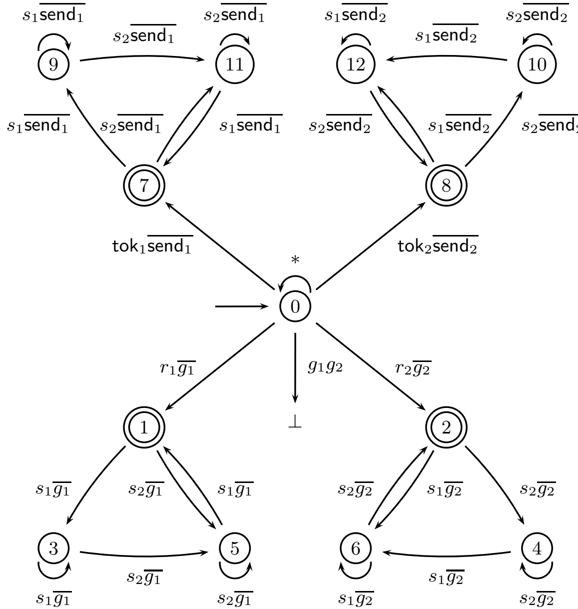

Fig. 2. Universal co-Büchi automaton for Example 2 for two processes

to φ. Similarly, the $\mathsf{fair_scheduling}$ assumption needs to be added to any liveness conditions of the specification, as without fair scheduling in general liveness conditions cannot be guaranteed.

Note that these additional formulas need not be taken into account when choosing which case of Thm. 1 needs to be applied.

Example 2. To synthesize an implementation of the simple arbiter from Example 1, we first add constraints for fair scheduling and fair token-passing. The resulting specification is

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{fair_scheduling} \rightarrow (\mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i)) \\ \forall i. & \mathsf{fair_scheduling} \rightarrow (\mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{send}_i)). \end{array}$$

For a ring of two processes, this specification translates to the co-Büchi automaton shown in Fig. 2. This automaton is encoded into a set of SMT constraints, part of which is shown in Fig. 3 (only constraints for states $0, 1, 3, 5$ of the automaton are shown). These constraints, together with general constraints for asynchronous systems, isomorphic processes, token rings, and size bounds, are handed to the SMT solver.

Correctness and Completeness of Bounded Synthesis for Token Rings. For a specification φ that is realizable in a token ring of size n, the given semi-algorithm will eventually find an implementation satisfying φ in token rings of

$$
\begin{array}{ll}
 & \lambda_0^{\mathbb{B}}(0) \\
 & \mathsf{tok}(d_1(0)) \wedge \neg\mathsf{tok}(d_2(0)) \\
\forall t.\ \forall I. & \lambda_0^{\mathbb{B}}(t) \rightarrow \lambda_0^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_0^{\#}(\delta(t,I)) \geq \lambda_0^{\#}(t) \\
\forall t. & \lambda_0^{\mathbb{B}}(t) \rightarrow \neg(g(d_i(t)) \wedge g(d_j(t))) \\
\forall t.\ \forall I. & \lambda_0^{\mathbb{B}}(t) \wedge r_1 \in I \rightarrow \lambda_1^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_1^{\#}(\delta(t,I)) > \lambda_0^{\#}(t) \\
\forall t.\ \forall I. & \lambda_1^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_3^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_3^{\#}(\delta(t,I)) \geq \lambda_1^{\#}(t) \\
\forall t.\ \forall I. & \lambda_1^{\mathbb{B}}(t) \wedge \mathsf{sched}_2(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_5^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_5^{\#}(\delta(t,I)) \geq \lambda_1^{\#}(t) \\
\forall t.\ \forall I. & \lambda_3^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_3^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_3^{\#}(\delta(t,I)) \geq \lambda_3^{\#}(t) \\
\forall t.\ \forall I. & \lambda_5^{\mathbb{B}}(t) \wedge \mathsf{sched}_2(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_5^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_5^{\#}(\delta(t,I)) \geq \lambda_5^{\#}(t) \\
\forall t.\ \forall I. & \lambda_5^{\mathbb{B}}(t) \wedge \mathsf{sched}_1(I) \wedge \neg g(d_1(t)) \rightarrow \lambda_1^{\mathbb{B}}(\delta(t,I)) \wedge \lambda_1^{\#}(\delta(t,I)) > \lambda_5^{\#}(t) \\
\dots & \dots
\end{array}
$$

Fig. 3. Constraints for Example 2 for two processes

size n. If φ furthermore falls into one of the classes described in Theorem 1, then the implementation will satisfy φ in token rings of arbitrary size.

4 Optimizations of the Encoding

In this section, we describe a first set of optimizations that make synthesis significantly more efficient. We consider the encoding of the problem into SMT constraints, and aim to remove as much decision freedom from the SMT solver as possible. All optimizations presented in this section are sound and complete. Optimization of *counters* was introduced in [12,7] and applies to bounded synthesis in general. *Bottom-up* encoding is possible in general (to some extent), but will be most useful for isomorphic systems. Finally, *fixed token function* is an optimization specific to token rings (or token-passing systems in general).

Counters. As mentioned in Sect. 3, labeling functions $\lambda^{\#}$ count the visits to rejecting states, and a satisfying valuation for them exists only if all run paths visit rejecting states only finitely often. In a run path, a repeated visit to the same rejecting state is possible only if the path stays in an SCC of the specification UCT $\mathcal{U}$. Therefore, we can reduce the number of $\lambda^{\#}$ annotations by introducing them only for states of $\mathcal{U}$ that are in a rejecting SCC. This optimization reduces the number of counters as well as their maximum value significantly.

Example 3. In the simple arbiter, this optimization means that we do not need $\lambda^{\#}$ annotations in the initial state. The benefit becomes more visible in a full arbiter, which in addition requires that there are no spurious grants and that every grant is lowered eventually. A simplified UCT for such a specification is given in Fig.4. Besides mutual exclusion of the grants, this presentation of the UCT only shows the constraints for arbiter i. In the figure, s_i stands for sched_i and $a_i = \mathsf{sched}_i \vee \mathsf{send}_{i-1}$ means the process is active and can react to the environment input. Note that the SCCs around states 2 and 3 only reject rings with fair scheduling — they become larger when processes are added. For this UCT, counters $\lambda^{\#}$ are only needed for states 3, 6, 8, 2, 5, and 7.

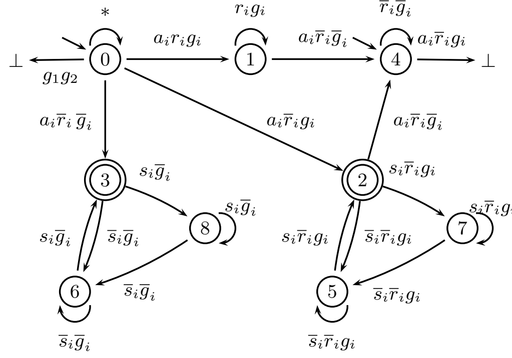

Fig. 4. UCT of the full arbiter

Bottom-Up. In the original approach described in Sect. 3, the SMT solver searches for a global transition function and projection functions d_i that satisfy input dependence, scheduling, and isomorphism constraints (2)(4)(5). Instead, we propose to go bottom-up: to search for a single process transition function and build the global one from local ones. Thus, all the processes share the same transition function symbol – this ensures their isomorphism, and constraints (5) can be removed. Also, process transitions functions now depend only on corresponding to the process inputs, and we can safely remove constraints (2). In addition, we wrap the transition function into an auxiliary function which calls the original if the process is scheduled or receives the token, and otherwise returns the current process state. This obviates the need for constraints (4).

$$
\begin{array}{ll}
 & \lambda_0^{\mathbb{B}}(0,1) \\
 & \mathsf{tok}(0) \wedge \neg\mathsf{tok}(1) \\
\forall t_1.\ \forall t_2.\ \forall I_1.\ \forall I_2. & \lambda_0^{\mathbb{B}}(t_1,t_2) \rightarrow \lambda_0^{\mathbb{B}}(t_1',t_2') \\
\forall t_1.\ \forall t_2. & \lambda_0^{\mathbb{B}}(t_1,t_2) \rightarrow \neg(g(t_1) \wedge g(t_2)) \\
\forall t_1.\ \forall t_2.\ \forall I_1.\ \forall I_2. & \lambda_0^{\mathbb{B}}(t_1,t_2) \wedge r_1 \in I_1 \rightarrow \lambda_1^{\mathbb{B}}(t_1',t_2') \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & \lambda_1^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_2 \wedge \neg g(t_1) \rightarrow \lambda_5^{\mathbb{B}}(t_1',t_2') \wedge \lambda_5^{\#}(t_1',t_2') \geq \lambda_1^{\#}(t_1,t_2) \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & \lambda_5^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_2 \wedge \neg g(t_1) \rightarrow \lambda_5^{\mathbb{B}}(t_1',t_2') \wedge \lambda_5^{\#}(t_1',t_2') \geq \lambda_5^{\#}(t_1,t_2) \\
\forall t_1.\ \forall t_2.\ \forall I_1\ .\forall I_2. & \lambda_5^{\mathbb{B}}(t_1,t_2) \wedge \mathsf{sched}_1 \wedge \neg g(t_1) \rightarrow \lambda_1^{\mathbb{B}}(t_1',t_2') \wedge \lambda_1^{\#}(t_1',t_2') > \lambda_5^{\#}(t_1,t_2)
\end{array}
$$

Fig. 5. Some of constraints for Example 2 for two processes using the Bottom-up encoding and Counters optimizations; $t_i' = \delta(t_i, I_i)$, δ is a wrapped local transition function; labeling functions changed its type: $\lambda_q^{\#/B} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}/\mathbb{B}$.

Similar approach that uses process transition functions explicitly was proposed in [11], but they still keep projection functions and a global transition function to describe the system. We removed these notions at all (see Fig. 5).

On the downside, this optimization does not allow us to bound the number of global states independently of the number of local states as in the original approach [13] or in [11] (and the number of global states is always equal to $|T|^n$).

Fixed Token Function. In the original approach, possession of the token is encoded by an uninterpreted function tok. We fix process states without/with a token $T_{\neg tok}/T_{tok}$ and define $\mathsf{tok}(t) := (t \in T_{tok})$, thus exempting the SMT solver from finding a valuation for tok. Fixing token possession functions has two important consequences.

First, it allows us to precompute global states with exactly one token in a ring. In case of 3 processes, global states are $\{(t_*, t, t) \cup (t, t_*, t) \cup (t, t, t_*)\}$, where $t_* \in T_{tok}, t \in T_{\neg tok}$. Then we build main constraints (1) only for these precomputed global states, and ignore other, invalid, global states. The system cannot move into an invalid global state with number of tokens different from one due to token ring constraints (3). In the original approach, invalid global states constitutes most of the global state space, and ignoring them reduces the state space significantly (exponentially in number of processes), leading to smaller SMT queries.

The second consequence of fixing tok is a possible restriction of generality of solutions. Different separations $T_{\neg tok}/T_{tok}$ may lead to different solutions. In general, systems with larger ratio $p = |T_{\neg tok}|/|T_{tok}|$ have a larger global state space, and processes without a token have more possibilities for transitions. With a maximal p, the system is completely parallel, and a minimal p leads to a sequential processing. The choice of p also affects the synthesis time because it changes the number of global states and, therefore, the size of the query.

A related optimization is to use binary encoding for token possession and sending that would automatically remove bad states from the consideration.

5 Extensions of Supported Language

Before introducing a second set of optimizations, we consider some extensions of our specification language, enabling us to treat more interesting examples. While the results of Emerson and Namjoshi [8] give us cut-offs that allow for parameterized synthesis in principle, a closer inspection of examples from the literature shows that the supported language is not expressive enough to handle many of them. Consider the parameterized arbiter specification introduced by Piterman, Pnueli and Sa'ar [15] with a specification of the form

$$\mathsf{Assume} \rightarrow \mathsf{Guarantee}, \text{ where}$$

$$\mathsf{Assume} \equiv \textstyle\bigwedge_i \left(\overline{r_i} \wedge \mathsf{G}\left((r_i \neq g_i) \rightarrow (r_i = \mathsf{X}\, r_i)\right) \wedge \mathsf{G}\left((r_i \wedge g_i) \rightarrow \mathsf{F}\, \overline{r_i}\right)\right)$$

$$\mathsf{Guarantee} \equiv \textstyle\bigwedge_{i \neq j} \left(\mathsf{G}\, \neg(g_i \wedge g_j)\right) \wedge \bigwedge_i \left(\overline{g_i} \wedge \begin{pmatrix} \mathsf{G}((r_i = g_i) \rightarrow (g_i = \mathsf{X}\, g_i)) \\ \wedge\ \mathsf{G}((r_i \wedge \overline{g_i}) \rightarrow \mathsf{F}\, g_i) \\ \wedge\ \mathsf{G}((\overline{r_i} \wedge g_i) \rightarrow \mathsf{F}\, \overline{g_i}) \end{pmatrix}\right).$$

This specification points to three limitations in the language considered by Emerson and Namjoshi:

First, the conjunction over all processes in the assumption turns into a disjunction when we bring the formula into a prenex form. Disjunctions over processes are however not supported by Emerson and Namjoshi. Second, this formula will quantify over (at least) three independent variables, which is also not supported in their framework. We will show that these two limitations can be effectively overcome in token-rings by using more general results by Clarke et al. [5] on network decomposition.

Finally, the specifications of Assume and $\mathsf{Guarantee}$ contain the X operator, which is completely excluded from the language of both Emerson/Namjoshi and Clarke et al. One may assume that one of the reasons for excluding the X operator is that, in asynchronous distributed systems, the presence of a (fair but otherwise arbitrary) scheduler can invalidate statements about the next state of a given process by simply not scheduling it. We will make some observations about cases where the usual X operator can still be used, and furthermore introduce a local variant of the X operator, that takes scheduling of a process into account.

5.1 Network Decomposition for Token Rings

The results by Clarke et al. allow for specifications with both conjunctions and disjunctions over all processes, and also allow an arbitrary number of index variables. For checking a property ϕ in a token-passing network, the consider decompositions into possible network topologies, where processes that are not represented by an index in ϕ are replaced by so-called *hubs* which simply pass on the token. A *k-indexed property* is a formula with arbitrary quantification that refers to system variables of at most k different processes. Their main result is

Theorem 2 ([5]). *For checking any k-indexed LTL$\backslash$X property ϕ in token-passing networks, it is sufficient to check ϕ on up to $3^{k(k-1)}2^k$ network topologies of size up to $2k$.*

However, in general this result is not constructive, a suitable decomposition into network topologies still needs to be found (for arbitrary network architectures). For the case of token rings, it is easy to find the suitable decomposition: for every number of processes n, there is only one ring of this size. Thus, Thm. 2 directly implies that for any k-indexed property, it is sufficient to check it on rings of all sizes up to $2k$. We can even get a result that is a bit stronger (in that it does not require to check small rings if we only consider rings of size $> 2k$):

Corollary 2. *If ϕ is a k-indexed property and Π a parameterized token-ring architecture, then $2k$ is a cut-off for ϕ.*

Proof. Suppose ϕ holds in a ring of size $2k$, but not in some bigger ring. This means that there is a tuple $(p_1, \ldots, p_k)$, or a set of such tuples, such that for this combination of processes, ϕ is not satisfied. Using the terminology of Clarke et al., each of these tuples defines a network topology, by abstracting processes that are not in the tuple into hubs, where all neighboring processes are abstracted into one hub. By Clarke et al., every network with the same topology will not satisfy

ϕ. Since the size of this network topology is at most $2k$ (hubs and processes p_i alternating), we can find tuples of processes with the same topology in the ring of size $2k$. Thus, ϕ cannot hold in the ring of size $2k$. Contradiction. □

With this result, parameterized synthesis in token rings can be extended to include specifications with an arbitrary number of quantified variables, and arbitrary quantifier alternations.

5.2 Handling the X Operator

In the example given above, all X operators are used in a way that forbids change as long as some conditions hold. We note that this special usage of the X operator is not a problem when we use this specification in asynchronous distributed systems, for two reasons:

1. In this use case, X operators do not make the specification unrealizable because of the scheduling. Indeed, the environment cannot simply invalidate the property by not scheduling the process, since it will trivially hold in the next step if the process controlling the output does not change it.
2. Cut-off results we use holds for LTL\X, but they still hold for this specification, since we can always rewrite such usage of X operators into a form without X: $\mathsf{G}(\varphi \rightarrow p = \mathsf{X}\, p) \Leftrightarrow \mathsf{G}(\varphi \rightarrow p \,\mathsf{W}\, \neg\varphi) \wedge \mathsf{G}(\varphi \rightarrow \neg p \,\mathsf{W}\, \neg\varphi)$.

In addition to this special usage of the usual X operator, we can consider examples with a local variant X_i useful for specifying Globally Asynchronous Locally Synchronous [4] systems. The local X_i specifies the next state from the perspective of process i,

$$\mathsf{X}_i\, p_i \Leftrightarrow \mathsf{F}(\mathsf{sched}_i \wedge \mathsf{X}\, p_i).$$

Obviously, this operator is insensitive to scheduling (in the sense that the environment cannot invalidate properties by not scheduling the process). Furthermore, all our cut-off results still hold for specifications that are conjunctions $\varphi_1 \wedge \ldots \wedge \varphi_n$, if we allow X_i to appear only in local properties φ_i.

6 General Optimizations

In this section we describe high-level optimizations that are not specific to the SMT encoding. The first two optimizations, *incremental solving* and *modular generation of constraints*, are sound and complete. The third, *specification strengthening*, is based on automatic rewriting of the specification and introduces incompleteness. The last optimization, *hub abstraction* is sound and complete.

Modular generation of constraints and specification strengthening apply to bounded synthesis (although the first is particularly useful for parameterized synthesis), while incremental solving only applies to parameterized synthesis. Hub abstraction is specific to token-passing systems.

Incremental Solving. Corollary 2 states that it is sufficient to synthesize a token ring of size $2k$ for k-indexed properties. However, a solution for a smaller number of processes can still be correct in bigger rings. We propose to proceed incrementally, synthesizing first a ring of size 1, then 2, etc., up to $2k$. After synthesizing a process that works in a ring of size n, we check whether it satisfies the specification also in a ring of size $n+1$. Only if the result is negative, we start the computationally much harder task to synthesize a ring of size $n+1$.

Modular Generation of Constraints for Conjunctive Properties. A very useful property of the SMT encoding for parameterized synthesis is that we can separate conjunctive specifications into their parts, generate constraints for the parts separately, and finally search for a solution that satisfies the conjunction of all constraints. In the following, for a parametric specification φ and a number of processes k, let $C(\varphi, k)$ be the set of SMT constraints generated by the bounded synthesis procedure (for a fixed parameterized architecture Π).

We start from the following observation: let $\varphi_1 \wedge \varphi_2$ be a parametric specification, Π a parameterized architecture, $\mathcal{T}$ a process implementation. If $\Pi, \mathcal{T} \models \varphi_1$ and $\Pi, \mathcal{T} \models \varphi_2$, then $\Pi, \mathcal{T} \models \varphi_1 \wedge \varphi_2$. While this may seem trivial, when combined with Thm. 1 and Cor. 1, we obtain the following[3]

Theorem 3. *Let Π be a parameterized architecture and $\varphi_1 \wedge \varphi_2$ a parametric specification, s.t. n_1 is a cut-off for φ_1, and n_2 a cut-off for φ_2 in Π. Then,*

$$\mathcal{T} \models C(\varphi_1, n_1) \wedge C(\varphi_2, n_2) \quad \Rightarrow \quad \Pi(k), \mathcal{T} \models \varphi_1 \wedge \varphi_2 \text{ for } k \geq max(n_1, n_2).$$

In parameterized synthesis, this not only allows us to use separate sets of constraints to ensure different parts of the specification, but also to use different cut-offs for different parts. By conjoining the resulting constraints of all parts, we obtain an SMT problem s.t. any solution will satisfy the complete specification. For a specification like

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i), \end{array}$$

this allows us to separate the global safety condition from the local liveness condition. Then, only for the former we need to generate constraints for a ring of size 4, while for the latter a ring of size 2 is sufficient. This is particularly useful for specifications where the local part is significantly more complex than the global part, like our more complex arbiter examples.

Specification Strengthening. To simplify the specification in assume-guarantee style, we remove some of its assumptions with two rewriting steps. These steps are sound but incomplete, and lead to more robust specifications.

Consider a specification in assume-guarantee style $A_L \wedge A_S \rightarrow G_L \wedge G_S$ with liveness and safety assumptions and guarantees. Our first strengthening is based on the intuition that in practice A_L is not needed to obtain G_S, so we strengthen

[3] Note that, in a slight abuse of notation, we use $\mathcal{T}$ both for the model of the SMT constraints, and for the implementation represented by this model.

the formula to $(A_S \rightarrow G_S) \wedge (A_L \wedge A_S \rightarrow G_L)$. This step is incomplete for specifications where the system can falsify liveness assumptions A_L and therefore ignore guarantees, or if the assumptions $A_S \wedge A_L$ are unrealizable but A_S is realizable. We assume that such cases are not important in practice[4].

Our second strengthening "localizes" assumptions and guarantees. Consider a 2-indexed specification in assume-guarantee style: $\bigwedge_i A_i \rightarrow \bigwedge_j G_j$. Adding assumptions on fairness of scheduling and token ring guarantees, we get:

$$\bigwedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow \bigwedge_j G_j$$
$$\bigwedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow \bigwedge_j TR_j$$

where TR_j are token ring constraints (3), A_i, G_j are assumptions and guarantees referring only to a single process[5]. The truth of $\wedge_j TR_j$ implies the truth of fair token passing $\wedge_j \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_j$, therefore we can add it as an assumption to the first line (logically equivalent to the original specification). After that we "localize" the specification by letting every implication only refer to one process (sound, but incomplete) and get the final strengthened specification:

$$\bigwedge_i (\mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \wedge \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i \rightarrow G_i)$$
$$\bigwedge_i (\mathsf{G}\,\mathsf{F}\,\mathsf{sched}_i \wedge A_i \rightarrow TR_i)$$

The second step, where we add $\wedge_i \mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ as an assumption to the first constraint, is crucial. Otherwise, the final specification becomes too restrictive and we may miss solutions. The reason why $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may prevent this is that $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may work as a local trigger of a violation of an assumption. This is confirmed in the "Pnueli" arbiter experiment, where a violation of one of the assumptions A_i prevents fair token passing in the ring, falsifying $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_j$ for all $j \neq i$.

Filiot et al. in [9] proposed a rewriting heuristic which is essentially a localization step mentioned above. Our version is slightly different since we add $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ assumptions before localization to prevent missing the solutions.

These two strengthenings may change the type, and hence the cut-off, of a specification. For example, after strengthening, the "Pnueli" arbiter specification changes its type from 3-indexed to 2-indexed. Furthermore, most properties become local, and can be efficiently synthesized with the following optimization.

Hub-Abstraction. From Clarke et al. [5] it follows that checking local properties in a token ring is equivalent to checking properties of one process in a ring of size 2, while the second process is a *hub* process which is only required to pass tokens it receives. Instead of introducing an explicit hub process, we model its behavior with environment assumptions A_{hub} for the first process: i) if the process does not have the token, then the environment will finally send the token, ii) if the process has the token, then the environment can not send the token:

$$\mathsf{G}(\neg \mathsf{tok} \rightarrow \mathsf{F}\,\mathsf{send}_{hub})$$
$$\mathsf{G}(\mathsf{tok} \rightarrow \neg\,\mathsf{send}_{hub}).$$

[4] For example, well known class of GR1 specifications [3] used to describe some industrial systems does not use liveness assumptions for safety guarantees. Specifications with unrealizable assumptions likely contain designer errors.

[5] This can be generalized to k-indexed assumptions and guarantees.

We add the hub assumptions A_{hub} above to the original specification and synthesize a single process. Therefore the final property to synthesize becomes:

$$\mathsf{G}\,\mathsf{F}\,\mathsf{sched} \wedge A \wedge A_{hub} \rightarrow G \wedge TR.$$

This property is equivalent to the original one, that is, the abstraction step is sound and complete. The abstracted property is more complex than the original one, but it can be synthesized in a single process setting. Therefore we trade size of a token ring to be synthesized for the size of the specification.

We can go further and replace $\mathsf{G}\,\mathsf{F}\,\mathsf{sched}$ with *true*, which can introduce unsoundness in general. But this step is still sound for the class of specifications mentioned in Sect. 5.2, where the environment cannot violate guarantees by not scheduling the process. This is true for all examples we consider in this paper, and a reasonable assumption for many asynchronous systems.

7 Experiments

For the evaluation of optimizations we developed an automatic parameterized synthesis tool that 1) identifies the cut-off of a given LTL specification 2) adds token ring constraints and fair scheduling assumptions to the specification 3) translates the modified specification into a UCT using LTL3BA [1] 4) for a given cut-off and model size bound encodes the automaton into SMT constraints 5) solves the constraints using SMT solver Z3 v.4.1 [6]. If the solver reports unsatisfiability, then no model for the current bound exists, and the tool goes to step 4 and increases the bound until the user interrupts execution or a model is found. A model synthesized represents a Moore machine that can be cloned and stacked together into a token ring of any size.

We have run our experiments on a single core of a Linux machine with two 4-core 2.66 GHz Intel Xeon processors and 64 GB RAM. Reported times in tables include all the steps of the tool. For long running examples, SMT solving contributes most of the time.

For the evaluation of optimizations we run the tool, with different sets of optimizations enabled, on three examples: a simple arbiter, a full arbiter, and a "Pnueli" arbiter. We show solving times in seconds in Table 1 and Table 2. The horizontal axis of the table has columns for token rings of different sizes – up to a cut-off size – 4 for simple and full arbiters, and 6 for "Pnueli" arbiter.

7.1 Encoding Optimizations

Each successive optimization below includes previous optimizations as well.

Original. The implementation of the original version is described in Sect. 3. It starts with a global transition function and uses projection functions and SMT constraints to specify the underlying architecture and isomorphism of processes.

Counters. We use SCC-based counters for rejecting states, minimizing the necessary annotations of our implementations.

Table 1. Effect of encoding optimizations on synthesis time (in seconds, t/o=2h)

	simple4	full2	full3	full4	pnueli2	pnueli3	pnueli4	pnueli5/6
original	11	t/o	t/o	t/o	52	t/o	t/o	t/o
counters	8	2316	t/o	t/o	19	t/o	t/o	t/o
bottom-up	3	24	934	t/o	23	6737	t/o	t/o
fixed tok function	1	2	28	327	7	252	5691	t/o
total speedup	11	$\geq 10^3$	$\geq 10^2$	≥ 20	7	≥ 30	≥ 1.5	-

Bottom-Up. In this version we use the same local transition and output symbols for all the processes. The significant speedup (two orders of magnitude for full2) is caused by two factors: the number of unknowns gets smaller (we don't need projection functions) and the size of SMT query becomes smaller (no need for constraints (2),(4),(5)).

Fixed Token Function. In this version SMT queries contain constraints only for global states with exactly one token in a ring. It is possible because we hard-code tok by dividing a process state space into two equal sets of states without/with a token (1/2 in case of 3 states). Similar to the previous, this optimization is efficient because it reduces the size of SMT queries and the number of unknowns.

In all experiments, constraints were encoded in AUFLIA logic. We also tried bitvector and real number encodings, but with no considerable speed-up.

7.2 General Optimizations

Each successive optimization below includes previous optimizations except Fixed token function, since it is not clear how to divide process state space in a general way. As a "non-optimized" reference version we use bottom-up implementation.

Incremental Solving. Solving times can be sped up considerably by synthesizing a ring of size 2, and checking whether the solution is correct for a ring of size 4. For instance, for the full arbiter, the general solution is found in 24 seconds when synthesizing a ring of size 2 (time from the "bottom-up" row in Table 1). Checking that the solution is correct for a ring of size 4 takes additional 30 seconds, thus reducing the synthesis time from more than 2 hours to 54 seconds. Times for incremental solving are not given in the table.

Strengthening. This version refers to two optimizations described in Sect. 6 - localizing of assume-guarantee properties and removing liveness assumptions from properties with safety guarantees. Specification rewriting works very well, significantly reducing the size of the specification automaton: for example, the automaton corresponding to the "Pnueli" arbiter in a ring of size 4 after rewriting reduces its size from 1700 to 31 nodes (from 41 to 16 for the full arbiter). Also, this optimization changes the cut-off from 6 to 4 for the "Pnueli" arbiter. We left token rings of size 5/6 in rows below to demonstrate scalability of optimizations.

Table 2. Effect of general optimizations on synthesis time (in seconds, t/o=2h)

	simple4	full2	full3	full4	pnueli2	pnueli3	pnueli4	pnueli5	pnueli6
bottom-up	3	24	934	t/o	23	6737	t/o	t/o	t/o
strengthening	1	6	81	638	2	13	90	620	6375
modular	1	4	8	13	2	4	11	49	262
async hub	1	2	2	5	2	3	9	37	236
sync hub	1	1	2	4	2	3	8	42	191
total speedup	3	20	10^2	$\geq 10^3$	10	10^3	$\geq 10^3$	$\geq 10^2$	≥ 40

Modular. In this version, constraints for specifications of the form $\phi_i \wedge \phi_{i,j}$ are generated separately for local properties ϕ_i and for global properties $\phi_{i,j}$, using the same symbols for transition and output functions. Constraints for ϕ_i are generated for a ring of size 2, and constraints for $\phi_{i,j}$ for a ring of size 4. These sets of constraints are then conjoined in one query and fed to the SMT solver. Such separate generation of constraints leads to smaller automata and queries, resulting in approximately 10x speed up.

Hub Abstractions. By replacing one of the processes in a ring of size 2 with assumptions on its behavior, we reduce the synthesis of a ring of size two to the synthesis of a single process. In row "async hub" the process is synthesized in an asynchronous setting, while in row "sync hub" the process is assumed to be always scheduled. The results do not show a considerable speed up, but this optimization might work in cases of larger specifications.

Remarks. It should be noted that our set of experiments is relatively small, and that SMT solvers are sensitive to small changes in the input. Thus, the experiments would certainly benefit from a larger set of benchmarks, and the individual comparison of any two numbers in the table should be taken with a grain of salt. At the same time, the table shows a clear and significant improvement of the solving time when all optimizations are turned on.

8 Conclusions

We showed how optimizations of the SMT encoding, along with modular application of cut-off results, strengthening and abstraction techniques, leads to a significant speed-up of parameterized synthesis. Experimental results show speed-ups of more than three orders of magnitude for some examples. We also showed that using the X operator does not necessarily break cut-off results or make the specification unrealizable in an asynchronous setting. Finally, we applied cut-off results from verification of general token passing systems [5] to synthesis in token rings, thus extending the specification language that the parameterized synthesis method [13] can handle.

The current bottleneck of SMT-based bounded (and thus, parameterized) synthesis is the construction of the UCT automaton. In our experiments, LTL3BA

could not generate the UCT for an AMBA arbiter with only 1 client within two hours. Therefore, we think that it will be important to develop techniques that help us to avoid construction of the whole automaton (for example by separate tracking of assumptions and guarantees violations, as in [7]).

Acknowledgments. We thank Helmut Veith for inspiring discussions on parameterized systems, Bernd Finkbeiner and Sven Schewe for discussions on distributed and bounded synthesis, and Leonardo de Moura for help with Z3.

References

1. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
2. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A New Requirements Analysis Tool with Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences 78, 911–938 (2012)
4. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford Univ., CA (1984)
5. Clarke, E.M., Talupur, M., Touili, T., Veith, H.: Verification by Network Decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design 40, 232–262 (2012)
8. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. International Journal of Foundations of Computer Science 14, 527–549 (2003)
9. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for LTL synthesis. Form. Methods Syst. Des. 39(3), 261–296 (2011)
10. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Logic in Computer Science (LICS), pp. 321–330. IEEE Computer Society Press (2005)
11. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Proc. Workshop on Automated Formal Methods, pp. 69–76. ACM (2007)
12. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. on Software Tools for Technology Transfer, 1–21 (2012)
13. Jacobs, S., Bloem, R.: Parameterized Synthesis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012)
14. Katz, G., Peled, D.: Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 117–132. Springer, Heidelberg (2011)
15. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
16. Pnueli, A., Rosner, R.: Distributed systems are hard to synthesize. In: Foundations of Computer Science (FOCS), pp. 746–757. IEEE Computer Society Press (1990)