Modular Synthesis of Reactive Programs

Kedar S. Namjoshi
Nokia Bell Labs
kedar.namjoshi@nokia-bell-labs.com

Aalok Thakkar
University of Pennsylvania
athakkar@cis.upenn.edu

Richard J. Trefler
University of Waterloo
trefler@cs.uwaterloo.ca

This paper formulates new methods for reactive program synthesis that are fundamentally modular in nature. That helps overcome state explosion which otherwise severely limits scalability.

The goal is to synthesize a multi-process reactive program from a temporal specification of its behavior. A key result by Pnueli and Rosner [11] shows that whether such a program exists is undecidable even if the individual processes are finite-state. On the other hand, the question is semi-decidable: one can systematically enumerate programs and use model checking to analyze each candidate. The drawback, however, is an exponential blowup in enumeration (a process with k states has $O(2^{k^2})$ choices for its transition relation), coupled with state explosion during model checking. Finkbeiner and Schewe [13] avoid explicit enumeration by encoding the entire synthesis question as a Boolean satisfiability query. This encoding is based on the global state space and therefore results in queries with *exponentially many* variables and constraints in N, the number of processes, severely limiting scalability. Our experiments show that this state explosion limits synthesis to a 7-process instance of a mutual exclusion protocol.

In a classic paper [5], Dijkstra shows how to 'invert' Hoare's proof system for program verification into a systematic method for program construction. This insight can be generalized to the principle that *any proof system for verification can be inverted into a method for program synthesis*. We propose to invert proof systems for modular reasoning and symmetry reduction into synthesis procedures which, by design, avoid state explosion. In the modular synthesis method, the number of variables and constraints in the SAT query grows only *polynomially* in N. Combined with symmetry restrictions, the method synthesizes a 42-process instance of a token-passing mutual exclusion protocol about 17 minutes.

There is a rich collection of proof systems in the literature, covering a variety of program models and specification logics. We focus here on synthesizing finite state programs in a standard shared memory model. Specifications are given as linear-time temporal logic properties.

The intuition is that one is progressively tightening the space of candidate programs and their correctness proofs. Non-modular synthesis searches over all programs and all proofs. Modular synthesis limits the search to modular proofs. Symmetry constraints restrict the space of candidate programs, requiring them to be isomorphic. Beyond improvements in scalability, the restrictions also result in programs that are closer to those constructed by hand, as hand-crafted protocols are typically symmetric and loosely coupled.

1 BACKGROUND

We begin by outlining general principles that turn a verification proof system into a synthesis method. Consider the problem of verifying that a transition system $M = (S, I, T, AP, \lambda)$ (state space S, initial states I, transition relation $T \subseteq S \times S$, atomic proposition set AP, labeling function $\lambda : S \to 2^{AP}$) satisfies a temporal property whose *negation* is defined as a Büchi automaton $A = (Q, \hat{q}, 2^{AP}, \delta, F)$ (automaton states Q, initial state $\hat{q}$, alphabet 2^{AP}, transition relation δ and accepting set F).

As is well known (cf. [7]), one needs an invariant $\theta \subseteq S \times Q$ and a partial function $\mathsf{rank} : S \times Q \to (W, \prec)$ (where the range is a well-founded set such as $(\mathsf{Nat}, <)$), with the following properties:

- (definedness) rank is defined for all pairs in θ,
- (initiality) $\theta(s, \hat{q})$ holds for all initial states s,
- (inductiveness) if $\theta(s, q)$ and $T(s, s')$ and $\delta(q, \lambda(s), q')$ then $\theta(s', q')$ holds,
- (rank decrease) if $\theta(s, q)$ and $T(s, s')$ and $\delta(q, \lambda(s), q')$ then $\mathsf{rank}(s', q') \preceq_q \mathsf{rank}(s, q)$. Here, $\preceq_q$ is $\prec$ if $q \in F$ and is $\preceq$ otherwise.

It is easy to see that if these conditions hold, there is no computation of M that is accepted by A. Consider any computation of M on which there is an accepting run of A. Every pair (s, q) along the run satisfies θ by initiality and inductiveness and thus has a rank value. By acceptance, a state in F occurs on the run infinitely often; by rank decrease, the induced sequence of ranks is an infinite strictly decreasing chain, which contradicts well-foundedness.

Non-Modular Synthesis. In [13] essentially this proof rule is turned into a finite-state synthesis method by limiting S to a finite set of states (with a fixed initial state), and the rank domain to a finite range of natural numbers. With S and W fixed, the proof conditions turn into Boolean constraints. The unknowns are the structural components (the transition relation and the labeling) and the proof components (the invariant and the rank relation). For instance, the transition relation is represented as a set of $|S|^2$ Boolean variables $\{T(s, s')\}$, and the invariant as a set of $|S| \times |Q|$ Boolean variables $\{\theta(s, q)\}$. From a solution to this set of constraints (if one exists), one can read off both the synthesized program (T, I, λ) *and* its correctness proof (θ, ρ).

Shared-State Concurrency. We apply this method to synthesize multiprocess reactive programs. The shared state space is denoted X. Each process M_i has a local state space L_i, and the structure $M_i = (S_i, I_i, T_i, AP_i, \lambda_i)$, where the state space S_i is $X \times L_i$. Concurrency is represented in the standard way as interleaving. The combined state space for N processes is $X \times S_0 \times \ldots \times S_{N-1}$, which has size exponential in N. This causes the non-modular SAT query to have an exponential number of variables and constraints.

We use a simple running example of a protocol for mutually exclusive access to a shared resource. The following requirements are based on per-process atomic propositions H ('hungry', no access to resource) and E ('eating', access to resource).

(1) (Labeling) In every state, a process is either hungry or eating, but not both.

$$\forall i : G((\neg \mathsf{H}_i \wedge \mathsf{E}_i) \vee (\mathsf{H}_i \wedge \neg \mathsf{E}_i)) \tag{1}$$

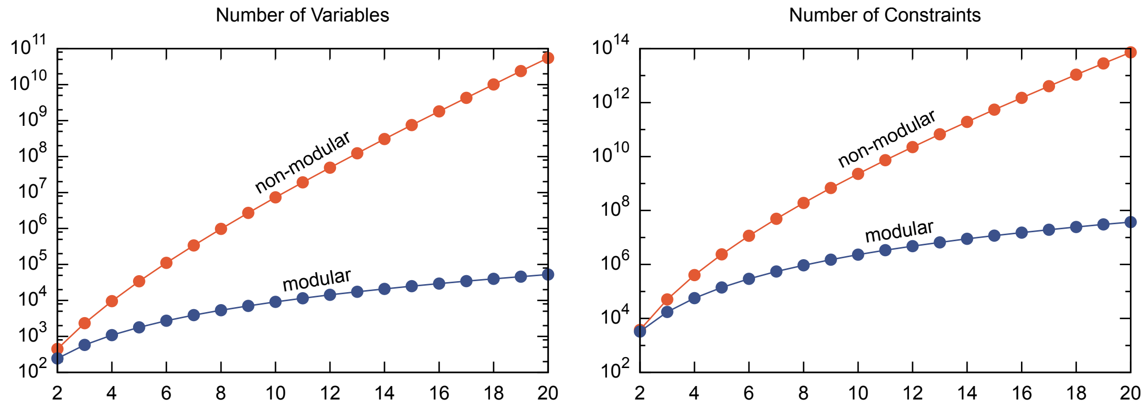

Figure 1: Growth of SAT variables and constraints with increasing N. Note the log-linear scale.

(2) (Mutual Exclusion) There is no global state where more than one process is eating.

$$\forall i, j : i \neq j : G(\neg \mathsf{E}_i \vee \neg \mathsf{E}_j) \quad (2)$$

(3) (Starvation Freedom) Every hungry process is eventually eating.

$$\forall i : G(\mathsf{H}_i \rightarrow F(\mathsf{E}_i)) \quad (3)$$

In our experiments, a protocol with N processes with 2 internal states requires a rank function bounded by at least $2N$. Therefore the number of variables grows as $O(N^3 2^N)$, limiting non-modular synthesis to a 7-process protocol instance.

2 MODULAR SYNTHESIS

The modular synthesis procedure relies on a modular proof system. Intuitively, it restricts the search space of protocols only to those that have a modular correctness proof.

The Modular Proof System. The proof system relies on a collection of per-process invariants $\{\gamma_i\}$ rather than a single global invariant. The Owicki-Gries-style conditions (cf. [8]) defined below ensure that (1) each assertion is inductive within its own process, and (2) it is unaffected by 'interference' (i.e., changes to shared state) resulting from the actions of other processes.

- (Initiality) $\gamma_i(x, l)$ for each $(x, l) \in I_i$,
- (Local Invariance) If $\gamma_i(x, l)$ and $T_i((x, l), (x', l'))$ then $\gamma_i(x', l')$ holds, and
- (Non-Interference) If $\gamma_i(x, l)$ and $\gamma_j(x, m)$ (for $j \neq i$) and $T_j((x, m), (x', m'))$ then $\gamma_i(x', l)$ holds.

It can be shown that the conjunction of the local invariants, $\gamma = (\wedge\, i : \gamma_i)$, is a global inductive invariant.

A local temporal property for a single process M_i is represented by a Büchi automaton A_i for its negation. The modular proof method follows the standard pattern defined previously, with one key change: a new non-interference rule checks that the invariant and rank function are unaffected by the actions of other processes.

- rank_i is defined for all pairs $((x, l), q)$ in θ_i,
- (Initiality) $\theta_i((x, l), \hat{q})$ holds for all $(x, l) \in I_i$,
- (Local Inductiveness and Rank Relation) if $\theta_i((x, l), q)$ and $T_i((x, l), (x', l'))$ and $\delta_i(q, \lambda_i(x, l), q')$ then $\theta_i((x', l'), q')$ holds, and $\mathsf{rank}_i((x', l'), q') \preceq_q \mathsf{rank}_i((x, l), q)$,
- (Non-Interference) if $\theta_i((x, l), q)$ and $\gamma_j(x, m)$ (for $j \neq i$) and $T_j((x, m), (x', m'))$ and $\delta_i(q, \lambda_i(x, l), q')$ then $\theta_i((x', l), q')$ holds and $\mathsf{rank}_i((x', l), q') \preceq_q \mathsf{rank}_i((x, l), q)$.

Soundness is established along the lines sketched for the general case. It establishes that all computations of the multiprocess system satisfy the local temporal property represented by A_i.

Verification to Synthesis. We now follow the bounded synthesis approach and limit X and each L_i to a finite space and similarly limit the rank domain W to be finite. As before, the proof constraints turn into propositional constraints on Boolean variables representing the components T_i and λ_i and the proof assertions γ_i, θ_i, and rank_i, for each process index i. Although the constraints may seem more complex, the number of variables and constraints in the query is *polynomial* in N.

There is, of course, a catch – in fact, two. Both arise from known limitations of modular methods. A modular method is limited to proving local temporal properties. We give an example of manually refining a non-local specification to a stronger localized specification. Second, not all correct programs may have purely modular proofs. This is remedied by adding auxiliary state and synthesizing auxiliary transitions; we omit details. The running example does not use them.

Refinement. We observe from the non-modular synthesized instances for small N that in the synthesized protocol, the shared variable acts as a token that cycles through the processes, ensuring mutual exclusion and starvation freedom. We now refine the specification, making it more localized and directing the search towards discovering such a protocol. In the refinement, we require the shared variable to take on values that are process indexes, and introduce a circular clockwise permutation π defined by $\pi(i) = (i+1) \bmod N$. The starvation freedom property (3) is refined to:

- A hungry process eats only if it has the token.

$$\forall i : G((x = i \wedge \mathsf{H}_i) \rightarrow F(\mathsf{E}_i)) \quad (4)$$

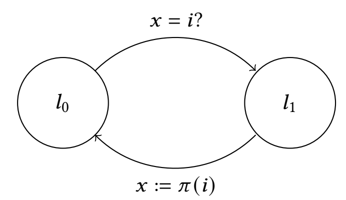

Figure 2: The internal states of process i in the synthesized mutual exclusion protocol expressed as a finite state system with guards. In this process, l_0 has labels $\{\texttt{hungry}_i, \neg\texttt{eating}_i\}$ and l_1 has labels $\{\neg\texttt{hungry}_i, \texttt{eating}_i\}$ and l_1.

- A process always releases its token to its successor.

$$\forall i : G(x = i \rightarrow F(x = \pi(i))) \tag{5}$$

By the second property, the token must eventually reach every hungry process, ensuring by the first property that it gets to eat. With the refined specification, the number of variables grows only as $O(N^3)$, a polynomial, as each local inductiveness constraint is defined only for pairs of states. This growth is improvement is evident Figure 1. Surprisingly, though, this improvement does not translate into decreased run time. The modular method can synthesize only up to 5 processes.

Symmetry. So far, we have restricted the shape of proofs by requiring modularity and strengthened the specifications to make them localized. We now add a structural requirement, requiring the synthesized processes to be symmetric.

$$\forall i : T_i((x, l), (x', l')) \rightarrow T_{\pi(i)}((\pi(x), l), (\pi(x'), l')) \tag{6}$$

Additionally, we prove that if there exists a symmetric solution, it must admit a symmetric proof, that is, it should satisfy constraints of the form:

$$\forall i : \theta_i((x, l), q) \rightarrow \theta_{\pi(i)}((\pi(x), l), q) \tag{7}$$

$$\forall i : \mathsf{rank}_i((x, l), q) = \mathsf{rank}_{\pi(i)}((\pi(x), l), q) \tag{8}$$

The addition of constraints (6)-(8) has a dramatic effect on scalability: the SAT solver synthesizes a protocol with 42 processes in about 17 minutes. Intuitively, this is because the symmetry constraint collapses the search: a decision to add a T_0 transition forces the addition of symmetric transitions in all other processes.

Although symmetry reduction allows checks to be limited to a single representative process, we observe that this does not significantly improve performance. Notably, one needs the combination of modularity and symmetry; symmetry alone does not work well, as observed in Figure 3.

3 RELATED WORK AND CONCLUSIONS

The central concept behind this work is simple and general: turn deductive proof rules into synthesis procedures. Modularity and symmetry have been used in synthesis, but in different settings. In [1], modularity is used to synthesize *synchronous, decoupled* multi-agent systems. Our asynchronous shared-memory model is

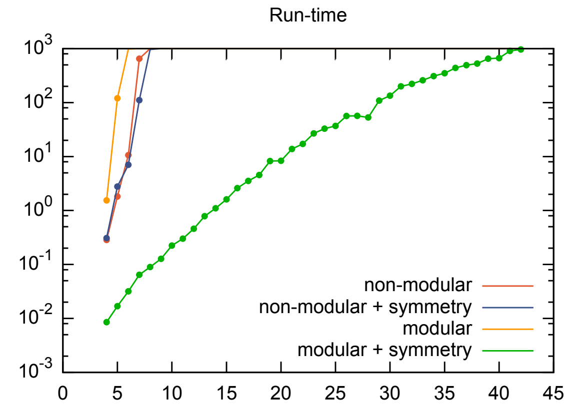

Figure 3: Figure showing run-times of the four methods.

different, as are the algorithms (SAT vs. game solving). Symmetry is used to speed up completion of partial process skeletons [2] and example-based synthesis of sequential programs [3]. To the best of our knowledge, prior work has not attempted to simply invert existing modular proof rules into a synthesis procedure.

Program synthesis is *not* a push-button procedure. As shown here, a protocol designer has to play an active part to convert specifications into localized forms, and to specify the right kind of symmetry. However, this is all at the meta-level of specifications, raising the level at which protocol design is carried out. Ongoing research is on whether one can build on the modular methods to directly synthesize parametrically correct protocols.

These results build upon much prior work. We use Manna and Pnueli's elegant formulation [7] of automaton-based deductive verification, and are inspired by the Dijkstra's approach to systematic program construction [5]. We build on the ideas in [6, 13] on bounded reactive synthesis and in particular the reduction to Boolean satisfiability. That turns out to be a very flexible notion, easily adapated to incorporate a variety of structural and proof constraints. The classical approach to reactive synthesis is via games or tree automata [4, 10, 12] which are mathematically elegant but appear to be quite difficult to adapt in a like manner. That is also the case for GR(1) synthesis [9], which is based on fixed-point constructions.

The results in this paper are only an initial exploration of the possibilities of proof system inversion. Given the rich variety of program models, specification logics, and accompanying deductive proof systems, this promises to be a particularly fertile topic.

Acknowledgement: The authors were supported, in part, by NSF awards CCF-1563393 and CCF-1703791 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2016. Compositional Synthesis of Reactive Controllers for Multi-agent Systems. In *CAV*. 251–269.

[2] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2015. Automatic Completion of Distributed Protocols with Symmetry. In *CAV*. 395–412.

[3] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2020. Augmented example-based synthesis using relational perturbation properties. *PACMPL* 4, POPL (2020), 56:1–56:24.

[4] J. Richard Büchi and L.H. Landweber. 1969. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.* 138 (1969), 367–378.

[5] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

[6] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2009. An Antichain Algorithm for LTL Realizability. In *Proc. of CAV*. 263–277.

[7] Zohar Manna and Amir Pnueli. 1987. Specification and Verification of Concurrent Programs By Forall-Automata. In *POPL*. 1–12.

[8] Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285.

[9] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of reactive (1) designs. In *International Conference on VMCAI*, Vol. 3855. Springer, Springer, 364–380.

[10] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *Prof. of POPL*. 179–190.

[11] Amir Pnueli and Roni Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *FOCS*. 746–757.

[12] M.O. Rabin. 1969. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.* 141 (1969), 1–35.

[13] Sven Schewe and Bernd Finkbeiner. 2007. Bounded synthesis. *Proc. of ATVA* (2007), 474–488.