VerC3: A Library for Explicit State Synthesis of Concurrent Systems

Marco Elver*
University of Edinburgh
marco.elver@ed.ac.uk

Christopher J. Banks
University of Edinburgh
c.banks@ed.ac.uk

Paul Jackson
University of Edinburgh
paul.jackson@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

***Abstract*—We propose an alternative, explicit state only, approach to concurrent system synthesis. In particular, the focus of this work is on the synthesis of distributed protocols. Given a correctness specification and a protocol skeleton (i.e. incomplete with holes), the goal is to synthesize the holes. At the heart of our technique is a dynamic programming based algorithm that prunes inferred failure candidates. The algorithm exploits the fact that typically only a few transitions are needed to reach an erroneous state in a faulty distributed protocol. Therefore, it is unlikely that every hole to be synthesized is contributing towards the error; thus, faulty protocol candidates where only a subset of holes were used can be used to infer failures of later candidates with a superset of holes.**

We evaluate the tool using a cache coherence protocol synthesis case study. Specifically, we study a directory based MSI protocol, assuming an unordered interconnect which gives rise to numerous race conditions which must be resolved via introducing transient states—a common cause of complexity and bugs in such protocols. In the case study, we therefore focus on synthesizing the transient state actions (we consider up to 12 holes out of possible 35). With the proposed candidate pruning optimization, we report up to 43x improvement over a naïve candidate enumeration scheme. We make available the tool and C++ library, VerC3.

I. Introduction

Program synthesis is the automatic completion of an implementation according to some specification [1], [2]. A variety of synthesis methods exist in the literature, but the most promising approach to real-world problems is based on *sketching* [3], or also called syntax-guided synthesis [4]. With sketching, the programmer not only provides a correctness specification, but also a skeleton (i.e. incomplete with holes) implementation. With the sketching approach to synthesis, the synthesizer should be seen as an assistant to the programmer, which aids in making the programmer more productive. Tasks where this approach is of great benefit are undoubtedly those where even experienced programmers can take significant time to derive a correct implementation.

One such task is the design and implementation of distributed protocols: finite-state systems that can be modelled using a collection of concurrent processes that run at arbitrary speeds and their respective steps interleave [5]. Prominent verification-only systems for such protocols are Murφ [5] (the modelling capabilities of our system are kept close to Murφ) and Spin [6]. The non-sequential behaviour makes human reasoning difficult for realistic problems and the aforementioned tools are already a must in every designers toolbox. Indeed, even reasonably small protocols (compared to sequential programs) quickly reach large state spaces [7], [8]. It is for these reasons that tool assistance during design—and not just during verification—with the help of synthesis would be of great benefit to reduce the time a protocol designer requires to arrive at a correct protocol.

An existing approach to distributed protocol synthesis is presented in TRANSIT [9], [10], which sidesteps the state explosion problem via the following strategy. Instead of synthesizing all of the missing pieces (holes) all at once, the transition rules corresponding to each hole are synthesized *independently*, relying on the designer who has to supply example executions to guide the synthesizer. Whether or not the individually synthesized transition rules combine to form a correct protocol is then verified with an external model checker. If the resulting protocol turns out to be incorrect, the whole process has to be repeated; indeed, the user has to integrate counter-examples from the model checker into the skeleton to better guide the synthesizer in the next iteration. This general strategy is an instance of *counter-example guided inductive synthesis* (CEGIS) [4].

One of our goals is to improve automation in the synthesis process, and rely only on protocol properties without the user providing example traces, or other hints to the synthesizer. As such, the synthesis procedure not only has to be aware of individual transition rules, but the complete protocol.

Many state-of-the-art synthesis procedures rely on symbolic techniques, usually relying on an external SMT solver [4], [11]. In distributed protocol verification, *symmetry reduction* techniques are essential [12]. Realizing symmetry reduction in symbolic verification, however, is significantly more complex [12]: we argue that for distributed protocol synthesis, explicit state methods, as successfully applied for verification [5], [6], [7], [8], may provide a better trade-off in complexity and performance.

In this paper, we present a tool that tightly couples synthesis procedure and an embedded explicit state model checker with promising results. At the core is a search algorithm that relies on a pruning technique for inferring failure candidates: we implement a dynamic programming based algorithm that prunes known failures due to specific holes, exploiting the fact that in an erroneous distributed protocol, a failure trace is unlikely

*Now at Google.

†This work is supported by EPSRC grant EP/M027317/1.

978-3-9819263-0-9/DATE18/©2018 EDAA

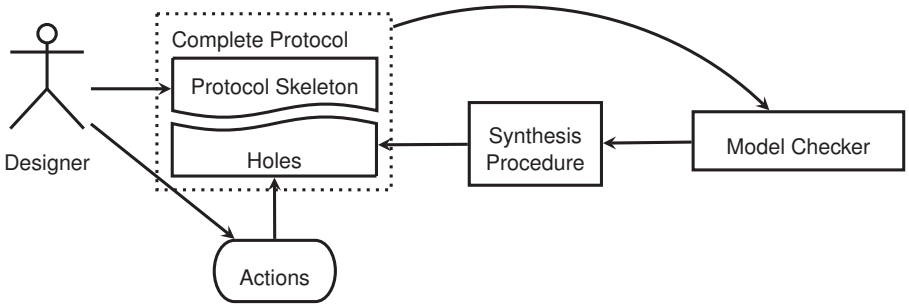


Figure 1: High-level overview of synthesis methodology with VerC3.

to have touched every hole.

Our case study focuses on multiprocessor cache coherence protocols [13], with the goal of synthesizing complete transition rules focusing on the protocol's transient states. Using our novel candidate pruning technique we achieve over an order of magnitude improvement over a naïve search strategy. We make available VerC3, a tool and modern C++ library which provides abstractions to implement, synthesize, and verify protocols: *https://github.com/icsa-caps/verc3*

II. Explicit State Synthesis of Concurrent Systems

A protocol designer would use our tool as follows: given a partial protocol design, with an understanding of its properties, the designer would provide a skeleton with the known aspects of the protocol implemented in the embedded DSL—which can be used to express any *guarded-command style finite-state transition system* (similar in expressiveness to Murφ [5]). Holes can be placed anywhere, and for each hole a pre-selected set of *pure functions* (with arbitrary arguments) can be selected to be enumerated by the synthesizer. For example, in coherence protocol synthesis (which we use for our case study in §III), the holes to be synthesized are actions (e.g. "respond to requester with data") similar to those found in protocols described in SLICC [14].

Figure 1 shows a high-level overview of the components included in VerC3. Given a *protocol skeleton* with *holes*: (1) the *synthesis procedure* selects combinations of known holes from the provided actions; (2) a complete *candidate* is then submitted to an *embedded explicit state model checker*, which returns information to the synthesis procedure about *hole discovery* and *correctness of candidates*. Upon discovery of a correct candidate, its configuration of holes is output to the user.

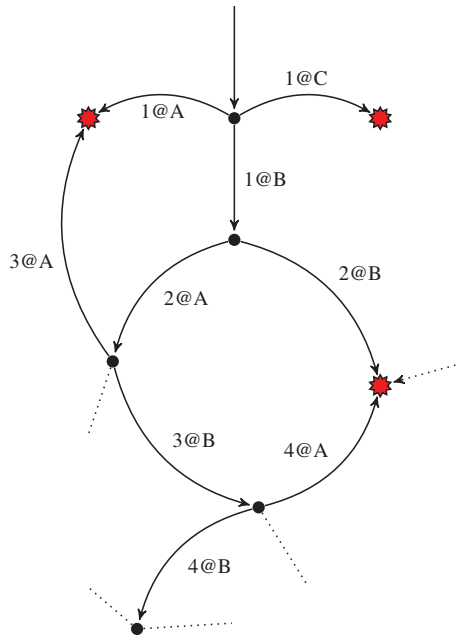
In the following we introduce each component of the system in more detail: first, we briefly introduce the modelling and model checking framework we implement, followed by synthesis and hole discovery without candidate pruning; we then introduce our key contribution, the *candidate pruning method* and conclude with a summary and worked example of the overall synthesis procedure; we also outline how the proposed synthesis procedure can be parallelized.

Modelling Transition Systems. We implement a minimal embedded model checker, with support for symmetry reduction [15], as a C++ template library. Transition systems are written in C++ and utilize the provided library of abstractions to describe the state, transition rules and properties of the system. The tool and library are usable without a frontend DSL, which we deemed beyond the scope of this work; future work includes the development of a more ergonomic frontend DSL.

Synthesis and Hole Discovery (without pruning). Holes can be placed anywhere, assuming that the expression to be synthesized can be captured in a set of actions of pure functions. Upon synthesis, the synthesis procedure selects the next combination of actions for holes and dispatches the completed protocol candidate to the model checker—initially, however, no holes are known to the synthesis procedure, i.e. holes are *discovered lazily*. Upon model checking, any newly encountered hole is registered and the default action substituted, such that the model checker may continue on the current branch of execution—this however changes with candidate pruning enabled (where the default action represents a wildcard), as will be introduced below. The synthesis procedure continues until no further combinations of actions for the discovered holes can be enumerated; correctly verified candidate hole configurations are displayed to the user.

Crucially, *lazy hole discovery avoids including holes which are never reachable* with a given protocol skeleton and holes. Internally, the set of holes discovered and the current hole configuration is represented as a *vector*—the "candidate configuration vector"—of indices pointing to the respective current action; upon hole discovery a new entry is appended. Holes are not replicated for each symmetric process, and are therefore *symmetry aware*. Notably, due to the explicit state nature of our approach, where holes and state are not represented together (unlike in symbolic methods), ensuring holes are not replicated unnecessarily becomes straightforward.

Candidate Pruning. The goal of candidate pruning is to infer ahead of dispatching a candidate to the model checker if it will fail, more specifically, if a configuration of holes will lead to failure. Our key insight is that, in a distributed protocol, it is extremely unlikely that every line of code nor hole will have been executed in a minimal error trace (the shortest possible



Run	*Candidate*	*Found Pruning Pattern*	*Discovered Holes*
1.	⟨ ⟩		1
2.	⟨ 1@A ⟩	⟨ 1@A ⟩	
3.	⟨ 1@B ⟩		2
4.	⟨ 1@C, 2@⋆ ⟩	⟨ 1@C, 2@⋆ ⟩	
5.	⟨ 1@B, 2@A ⟩		3
6.	⟨ 1@B, 2@B, 3@⋆ ⟩	⟨ 1@B, 2@B, 3@⋆ ⟩	
7.	⟨ 1@B, 2@A, 3@A ⟩	⟨ 1@B, 2@A, 3@A ⟩	
8.	⟨ 1@B, 2@A, 3@B ⟩		4
9.	⟨ 1@B, 2@A, 3@B, 4@A ⟩	⟨ 1@B, 2@A, 3@B, 4@A ⟩	
10.	⟨ 1@B, 2@A, 3@B, 4@B ⟩		

Figure 2: A worked example of synthesis. The graph represents a state graph, with nodes being distinct states, and edges the transitions between them. Each edge is annotated with a *hole@action* pair; the range of actions is $[\star, \text{A}, \text{B}]$ (with hole 1 having an additional action C) where $\star$ is the wildcard (viz. default action causing model checker to return the "unknown" result if no further failure is encountered). Due to lazy hole discovery, only as soon as a hole is discovered is it represented in the candidate vector. In the initial run (1) of the model checker, no holes are known and hole 1 is discovered; run (2) results in a failure, and the candidate configuration is added to the pruning patterns; run (3) discovers hole 2; run (4) advances hole 1 to C which results in another failure. Next, ⟨ 1@A, 2@A ⟩ is skipped as it matches with the first pruning pattern, and similarly matching candidates are skipped in the following. Run (5) discovers hole 3; run (6) results in a failure due to hole 2, and its candidate is entered into the list of pruning patterns; run (7) fails due to hole 3, and another pruning pattern is added; run (8) discovers hole 4, and finally runs (9) and (10) iterate through all actions for hole 4. With a naïve enumeration scheme, 24 candidates would have been evaluated, however, with candidate pruning, only 10 needed to be evaluated in this example.

sequence of transitions that leads to the error).[1] It follows that only a subset of a candidate configuration may be significant in a candidate failure.

More formally, let C be a candidate, i.e. the configuration of all holes, and $C_t \subseteq C$ the subset of holes in a configuration that are executed in a trace t; if C results in a candidate failure, with a particular error trace t due to C_t, then any other C' where $C_t \subseteq C'$ will also be a candidate failure with the same error trace t.

We use the above insight and implement a dynamic programming based algorithm that uses *pattern matching over candidate configurations* to infer if a next candidate will fail or not. In the candidate configuration patterns, we introduce *wildcards*, which will be the *default action* for newly discovered holes. Encountering a wildcard causes the model checker to abort execution on that execution branch. If wildcards have been encountered by the model checker, but no failures, a third verification result, "unknown" (in addition to "success" and "failure"), for the particular candidate is returned; note that it may be possible to encounter multiple undiscovered holes (viz. wildcards) in a single run of the model checker.

[1]This requires an appropriate evaluation algorithm in the model checker; we implement a standard Breadth-First-Search (BFS), which will yield the minimal trace to a property violation.

Only upon verification failure is the candidate configuration (including wildcards) entered into a lookup-table of *candidate pruning patterns*. The pruning patterns are queried for each new candidate's candidate configuration to infer if a property violation is certain to occur; if no failure inference can be made, the candidate is dispatched to the model checker.

Enumerating Candidate Configurations. Without wildcards, enumerating all candidate configurations is straightforward. However, with wildcards, a major challenge is to limit candidate configurations with wildcards, such that the complexity of evaluating the extra configurations (with wildcards) can still be offset by the net reduction in evaluated protocol candidates. Clearly, this implies we cannot enumerate all possible combinations with added wildcard actions.

As we represent the current candidate configuration as a vector, we found that the most effective policy is to partition the vector into two consecutive ranges of non-wildcard (always starting from the first discovered hole) and wildcard actions. By default, all undiscovered holes are considered as wildcards. Upon discovering a new hole, the non-wildcard range expands only when all combinations of actions for holes have been enumerated without the newly discovered hole(s). In other words, once a hole has been used as a non-wildcard in any candidate configuration, it cannot be used as a wildcard again.

The reason this works well can be explained as follows. When tracing execution from the initial states of a protocol candidate, a small number of initially discovered holes (usually just one) will likely only have one correct combination of actions, with other actions causing failure without involving any other holes (which are only encountered in longer traces). By considering all undiscovered holes as wildcards, we may enumerate all possible actions for the initially discovered holes and memoize failures in the lookup-table of pruning patterns. Upon including the next set of discovered holes, the previous set of actions for earlier discovered holes that are certain to cause failure alone, are never considered again.

Putting it all together. The synthesis procedure starts without knowledge of any holes in the transition system implementation: holes are discovered lazily, in the order they are encountered during model checking, i.e. the initial candidate is the empty candidate. The initial run of the model checker then returns one of three results: "unknown", "failure", or "success"; with the latter two indicating that the model is either inherently faulty or already completed respectively. Upon discovery of a hole, it is appended to the candidate configuration vector. Each hole is associated with a set of possible actions predefined by the designer; the default action of a hole upon discovery is the wildcard action, which causes the model checker to not explore executions beyond the new hole.

The candidate vector (now containing the first set of holes) is used to enumerate the next candidate, and evaluated by the model checker. Subsequently discovered holes are appended to the candidate vector as wildcards. Recall that, the enumeration policy is such that any hole that has already been a non-wildcard in any considered candidate, cannot be a wildcard again. Upon verification failure, the current candidate (including known wildcards) is entered into the lookup-table of pruning patterns. If a candidate matches any pattern in the pruning patterns discovered so far, it is skipped and not evaluated by the model checker.

This process continues until no further holes can be discovered, and all possible candidates have been evaluated. Any candidate which results in a verification result of "success" is displayed to the designer. Figure 2 shows a worked example of the proposed synthesis procedure with candidate pruning.

Parallel Synthesis. Distinct protocol candidates can be model checked independently: we use this as the basis to parallelize the synthesis procedure, by splitting the set of candidates to be evaluated and dispatching to multiple threads. Lazy hole discovery and maintenance of the candidate pruning patterns, however, make use of shared datastructures but have been optimized to minimize contention.

Initially, when no holes have been discovered yet, a single thread is dispatched to discover the first set of holes. Subsequently, multiple threads are assigned ranges of candidates to evaluate, which are maintained in thread-local candidate configuration vectors. However, we must avoid that independent threads race to discover the same holes. To resolve this, we maintain a global candidate vector which is used to register newly discovered holes only (during evaluation). When all threads are done with their currently assigned work, the global candidate vector is used to obtain the next range of candidates for each thread.

Consequently, the thread-safe implementations of the candidate vector and individual hole state have been the biggest source of contention. We minimize this contention by optimizing for the common case: to check if a hole has already been discovered and obtain its current action has been made lock-free.

III. Evaluation

In this section we present data for our case study. Our case study focuses on the synthesis of multiprocessor cache coherence protocols [13]: the coherence protocol is responsible for ensuring a consistent view of data replicated in multiple caches. A key safety property of traditional coherence protocols is the Single-Writer–Multiple-Reader (SWMR) invariant, which asserts that for all states there may only ever be a single writer but no readers, or multiple readers but no writers. Furthermore we implement several additional properties asserting liveness [16].

In our case study we want to complete a directory based MSI protocol as shown in Figure 3: the states shown are maintained for each cache line independently; each cache controller sends requests for reads and writes to a central directory which responds depending on a cache line's state. Upon a write request, the directory either ensures that all other sharers in Shared state are invalidated or (if there is a single owner in Modified state) ensures that ownership is transferred. Upon a read request, the directory either hits in Shared state or must send a downgrade request to the current owner. In Invalid state, the directory must obtain a copy from the next lower level memory.

The Problem of Transient States. Request and response messages, however, are exchanged on an unordered network which will require the introduction of numerous transient states to avoid race conditions and deadlocks. For example, the directory should serialize write accesses from multiple requesters by stalling until the first requester has acknowledged receipt of the data—this is accomplished by introducing a transient state (Invalid-to-Modified) in the directory that stalls on further read/write requests and only transitions to the stable state Modified upon acknowledgement receipt. Similarly, the cache controller has to introduce transient states when the data response is still outstanding. These transient states are often the cause of much confusion and bugs in coherence protocols [8], which is what we will focus on synthesizing in our case study.

Assumptions. Our case study assumes that the designer can complete the protocol's stable states and the transition rules leading from stable states to transient states. This implies that the designer can determine the required transient states and can provide a skeleton with their actions left blank; this process is straightforward if the designer starts with one transient state for each stable-stable pair and adding additional transient states if the synthesizer cannot find a solution (for this MSI protocol,

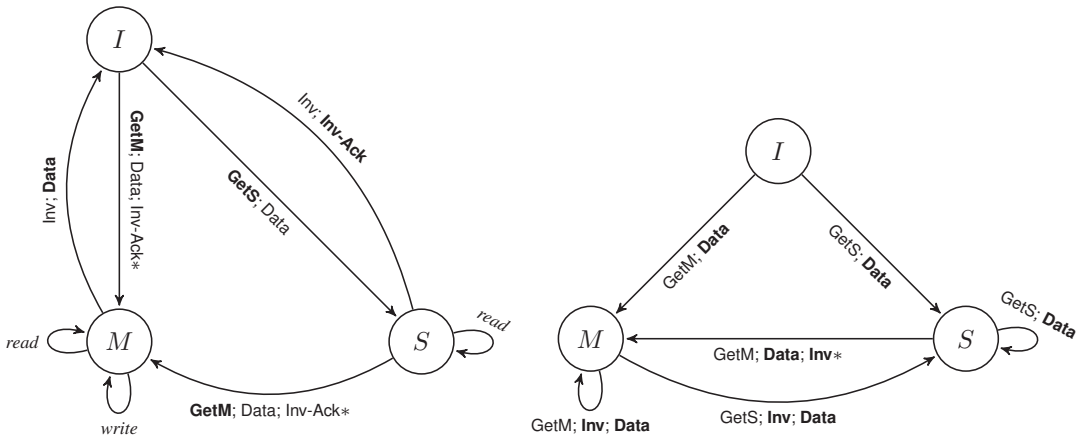


Figure 3: Directory based MSI protocol (stable states only, evictions omitted). *Left:* replicated cache controller; *right:* directory controller. All networks may be unordered. Each transition is labeled with a set of messages **sent (bold)** or received, where a $*$ denotes zero or more messages; the first message is the trigger.

Table I: Results for MSI coherence protocol case study.

Configuration	*Holes*	*Candidates*	*Pruning Patterns*	*Evaluated*	*Solutions*	*Exec. Time*
MSI-small 1 thread, no pruning	8	231,525	N/A	231,525	4	64.5s
MSI-small 1 thread, pruning	8	1,179,648	743	855	4	1.8s
MSI-small 4 threads, pruning	8	1,179,648	701	825	4	1.2s
MSI-large 1 thread, no pruning	12	102,102,525	N/A	102,102,525	12	31,573.5s
MSI-large 1 thread, pruning	12	1,207,959,552	34928	170,108	12	739.7s
MSI-large 4 threads, pruning	12	1,207,959,552	34888	170,087	12	295.7s

just one for each stable to stable transition). In addition, the designer has a library of protocol actions that the synthesizer can use for holes.

Each transition rule (from stable as well as transient states) can be split into two or three action types for cache and directory controller respectively; namely "response" (3 for cache controller, 5 for directory controller), "next state" (7 for cache controller, 7 for directory controller) and "track" (3 for directory controller) actions. Each hole corresponds to one of these action types, and a sequence of holes (of distinct action types) will make up a full transition rule.

Experimental Results. Table I shows experimental results for our MSI case study.[2] We evaluate our tool using two problem sizes that differ in the number of holes to be synthesized: MSI-small (8 holes = 2 directory + 1 cache transition rules) and MSI-large (12 holes = 2 directory + 3 cache transition rules). We note that synthesizing all transient states (35 holes = 7 directory + 7 cache transition rules) is not tractable for this protocol, but believe the true value of the synthesizer is in completing transition rules involved in non-trivial corner cases.

The synthesis problems MSI-small and MSI-large generate 4 and 12 distinct solutions respectively. In our initial experiments, however, the number of correct solutions was significantly larger due to lacking a key protocol property asserting that all stable states are indeed reachable ("all stable states must be visited at least once"). Without this property, valid protocols were generated, but many would have resulted in poor performance if implemented. For instance, a protocol which requests data in Invalid state, receives the response but immediately transitions straight back to Invalid is correct, but not very efficient (effectively renders the cache useless). Finally, for correctly verified solutions of the protocol, the model checker reports 5207, 6025 or 6332 visited states: even though up to 12 distinct solutions can be generated (for MSI-large), we could group them into 3 sets, where solutions within each set behave equivalently, yet subtly different from the other sets.

For MSI-small the candidate pruning optimization results in a 99.6% reduction of evaluated candidates (compared to no pruning), for an effective speedup of 35.8x. Similarly for MSI-large, resulting in a 99.8% reduction of evaluated candidates for an effective speedup of 42.7x. This shows that for small problem sizes the optimization makes interactive synthesis tractable, and for larger ones reduces synthesis time from hours to minutes.

Enabling parallel synthesis further improves the time to

[2]Experiment Platform: Intel(R) Core(TM) i7-4800MQ (2.70GHz, 8 hardware threads on 4 cores); 8GB RAM; Linux 4.8.4; Clang 3.8.1.

generate all possible solutions by an order of 2.5x (1.5x) for MSI-large (MSI-small). Parallel synthesis will yield the greatest benefit for larger problem sizes, as initial runs may incur frequent synchronization. It can also be seen that there is a subtle difference between the single- and multi-threaded evaluated candidates: this is a result of the shared pruning patterns, and the ability of each thread to make use of another thread's registered patterns as soon as they become available.

IV. Related Work

The closest related work is TRANSIT [9], [10]. Their methodology involves the user conveying information from example executions for transition rules. In doing so, they effectively side-step the complexity of making the synthesizer aware of the entire protocol, but rather focus on individual functions (the transition rules) synthesized by a "SyGuS" solver [4]. The completed protocol candidate is then dispatched to an explicit state model checker which provides feedback about the correctness of the protocol; in case of failure, it is the *user's* job to use the counter-example to further augment the protocol skeleton or refine example executions. Note that, TRANSIT uses dynamic programming to reduce the search space over *expressions* which are considered for *individual transition rules*; whereas our search technique uses dynamic programming to prune at a coarser granularity, i.e. over entire *protocol candidates* of actions. Our approach provides improved automation, albeit at coarser granularity of holes.

Unlike distributed protocol synthesis, concurrent program synthesis has seen more approaches in the literature. For instance, the language Sketch, which was one of the earliest sketching-based languages, supports synthesis of concurrent programs [3]: the synthesis procedure explicitly enumerates various schedules, and dispatches them to a SAT-based solver. This approach appears to work well for smaller programs, with little symmetry. The class of problems we considered are different, in that we are primarily targeting distributed protocol synthesis, with large amounts of inherent symmetry.

In the explicit model checking literature, Murφ [5] and Spin [6] are some of the most widely used model checkers for distributed protocols. Indeed, Murφ's origin can be found in coherence protocol verification, and as such we kept our model checker as close to what Murφ is capable of without introducing our own DSL, but rather expose the model checker as a C++ template library for seamless embedding into our synthesis procedure.

V. Conclusion

Program synthesis in general spans a large variety of techniques and applications [1]. In recent years, however, most synthesis tools rely on external SAT-based solvers due to their continually improving performance and investment [4], [11]. We observe, however, that distributed protocol synthesis may not be an ideal candidate for symbolic techniques and propose an explicit state approach to further improve automation.

Our key insight is that protocol synthesis with an explicit state model checker can make use of a candidate pruning optimization—a dynamic programming based algorithm that exploits the fact that in an erroneous distributed protocol a failure trace is unlikely to have touched every line of code (viz. holes)—for significantly reducing the evaluated candidates. For our case study, we synthesize the transient state actions of a directory based MSI cache coherence protocol; we synthesize up to 12 holes out of possible 35 and report up to 43x improvement over a naïve candidate enumeration scheme, reducing the synthesis time from hours to minutes. The presented tool and library, VerC3, provides the building blocks to synthesize distributed protocols, but also serve as a research vehicle to further explore distributed protocol synthesis.

Despite promising results, we acknowledge the difficulty of this problem, and note that the problem is far from solved (e.g. in our case study, synthesizing all transient states is still not tractable), and hope to inspire future work to take these ideas further. Indeed, verification problems in general suffer from enormous complexity. In the realm of distributed systems verification, especially coherence protocols, the state spaces that must be dealt with pose major challenges even for state-of-the-art model checking technology [7], [8]. The problem of synthesis adds another dimension to an already complex problem. For future work, we plan to find more ways to exploit symmetry and widen the scope of the tool as well as provide a lightweight frontend DSL.

References

[1] R. Bodík and B. Jobstmann, "Algorithmic program synthesis: introduction," *STTT*, vol. 15, no. 5-6, pp. 397–411, 2013.

[2] S. Gulwani, "Dimensions in program synthesis," in *PPDP*, 2010, pp. 13–24.

[3] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California, Berkeley, 2008.

[4] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*, 2013, pp. 1–8.

[5] D. L. Dill, "The Murφ Verification System," in *CAV*, 1996, pp. 390–393.

[6] G. J. Holzmann, "The model checker spin," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[7] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *IPDPS*, 2003, p. 11.

[8] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the complexity of hardware cache coherence and some implications," *TACO*, 2014.

[9] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Automatic completion of distributed protocols with symmetry," in *CAV*, 2015, pp. 395–412.

[10] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," in *PLDI*, 2013, pp. 287–296.

[11] E. Torlak and R. Bodík, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI*, 2014, p. 54.

[12] F. Pong and M. Dubois, "Verification Techniques for Cache Coherence Protocols," *ACM Comput. Surv.*, vol. 29, no. 1, pp. 82–126, 1997.

[13] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[14] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[15] C. N. Ip and D. L. Dill, "Better Verification Through Symmetry," in *CHDL*, 1993, pp. 97–111.

[16] K. L. McMillan and J. Schwalbe, "Formal verification of the gigamax cache consistency protocol," in *ISSM International Coherence on Parallel and Distributed Computing*, 1991.