A New Notion of Compositionality for Concurrent Program Proofs

Azadeh Farzan[1] and Zachary Kincaid[2]

1 **University of Toronto, Ontario, Canada**
`azadeh@cs.toronto.edu`
2 **Princeton University, NJ, USA**
`zkincaid@cs.princeton.edu`

Abstract

This paper presents a high level overview of Proof Spaces [11] as an instance of a new approach to compositional verification of concurrent programs and discusses potential future work extending the approach beyond its current scope of applicability.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Concurrency, Proofs, Dynamic Memory, Recursion

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.4

Category Invited Talk

1 Verification of Parameterized Concurrent Programs

Compositional proofs have always been the holy grail of reasoning about concurrent programs. Two seminal contributions, namely Owicki-Gries style [22] and *rely-guarantee* [16], have set the tone for compositional proofs of concurrent programs and inspired decades of research in this area. In both techniques (and the wealth of follow-up work), compositionality strictly means composing correctness arguments over individual threads into a correctness argument for the entire program, with some extra supporting arguments that are preferably kept at a minimum required. In a recent line of work [11, 10, 9, 12], we proposed a new approach to static analysis and verification of concurrent programs which include an unbounded number of concurrent threads with local and global memory, with a different notion of compositionality. This talk will provide an overview of this approach, which takes a different view of compositionality. In this paper, we hope to look forward and through examples discuss early evidence of why we believe our approach to be a promising framework for future research into extensions of the program models (of the already published work [11, 10, 9, 12]) into more expressive models including features like dynamic memory and recursion.

We will do an informal exposition of our approach from [11] using a simple example. The interested reader can refer to [11, 10, 9, 12] for the technical details. The principle behind our methodology is simple. It is difficult to reason about the correctness of a complex concurrent program, however, it is much simpler to reason about the correctness of a single behaviour (i.e. a single run) of the same program. Consider, for example, the complexity of reasoning about unbounded concurrency. Any (terminating) run of such a program always includes a bounded (by the length of the run) number of participating threads. Therefore, the complexity of dealing with unboundedly many threads can be circumvented whilst reasoning about the run. The idea is then to *mine* these simple proofs of program runs for ingredients to construct a correctness proof for the program.

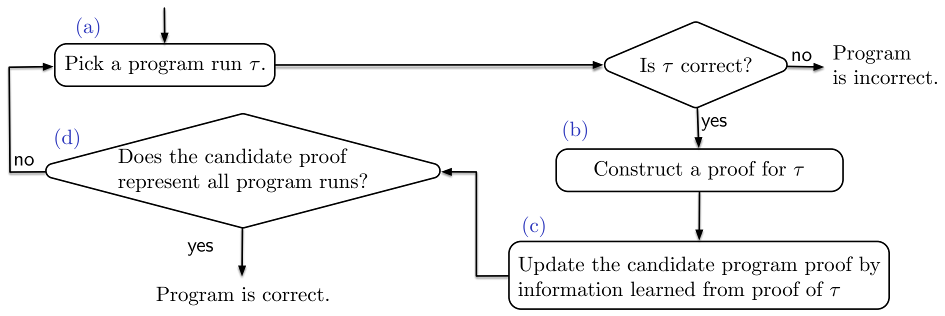

(a)
Pick a program run τ.
Is τ correct?
no
Program is incorrect.
yes
(b)
Construct a proof for τ
(c)
Update the candidate program proof by information learned from proof of τ
(d)
Does the candidate proof represent all program runs?
no
yes
Program is correct.

Figure 1 Proof Construction Methodology.

This effectively introduces a different way of decomposing the correctness argument for a concurrent program. The new proof is a non-trivial composition of proofs of correctness constructed for *finitely* many runs of the program. The correctness proof for each run represents a class of behaviours for the program with a common correctness argument. Instead of being forced to decompose the proof argument using the program syntax as a strict guide, one can observe how the program behaves in different *scenarios* and reason that in each case it exhibits the correct behaviour. Our constructed proofs may end up with structures very similar to or wildly different from the original program. This is extra flexibility is useful specially when a compositional proof in the classic sense is either nonexistent or hard to automatically construct.

The diagram in Figure 1 illustrates our methodology. The main goal of this methodology is to introduce a clean separation between

- logical reasoning about data, that is over domains of the program variables, and
- purely combinatorial reasoning about sufficiency of a candidate proof for a given program.

In many settings, distinctly so for concurrent programs, an argument for correctness often involves both types of reasonings. The thesis of this methodology is that this separation could make each argument type simpler, and moreover, each side of the argument becomes amenable to existing technology. In particular, on the logical reasoning side we can make use of the wealth of techniques that have been developed for loop invariant generation, such as abstract interpretation [6, 7], Craig interpolation [15, 18, 2], and constraint-based techniques [5, 14]. On the combinatorial side, we can make use of technology behind finite-state model checking, such as partial order reduction [32, 23, 13] and symbolic state-space exploration [4, 17].

The key steps in this methodology are:

- **Proof generation for a run.** Depending on the richness of the program model, this task can be straightforward with an array of existing solutions, or more tricky where new algorithms need to be proposed for it. In particular:
  - In [9], program runs are simple (straight line sequential) programs with integer typed variables, where constructing a proof for a run is a well-understood task. Since concurrent programs are assumed to have a fixed number of threads, the challenge is the well-known state explosion problem, and therefore the goal is compactness of proofs. The innovation of proof construction step in this case is focused on constructing *parallel* in proofs in form of *inductive data flow graphs*, that would generalize the proof of correctness of a run to many other *similar* runs.
  - In [10], where the program model was extended to include unboundedly many threads, a special type of proof, namely a counting proof, is constructed for a run. A special

```
global t:  int
global s:  int
local m:  int
initially s <= t

do forever {
  m = t++;
  // busy wait until your turn
  while (m > s);
  // enter critical section
  s++;
}
```

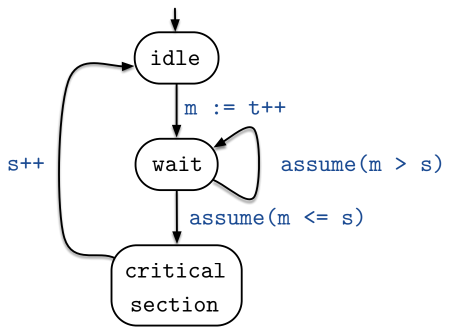


Figure 2 Ticket protocol code and control flow automaton.

technique for proof generation is proposed that automatically constructs a counting argument (i.e. a proof making use of counters) for a run.

- In [11], program runs are syntactically identical to those in [9], however, these program runs are (terminating) runs for a program with unboundedly many threads. A proof generated for a run in this context needs to be generalizable to the proof for a set of runs for such a program, and therefore, a different notion of proof, namely a *proof space* was introduced for these programs, a special proof generalization *symmetry* rule to tackle unbounded concurrency (more on this later).
- In [12], the program model is identical to the previous item, but the runs can potentially be non-terminating, since proving termination (and liveness in general) is the goals. Proofs for such runs, therefore, need to include termination arguments, which were ranking functions in this case.

- **Proof checking.** This is an algorithmic step that needs to (i) check if the candidate proof is indeed a proof of the given property for the given program, and (ii) provide a counter example if it is not, to ensure progress in the overall proof construction scheme (i.e. the choice of the run τ in (a) in Figure 1). This requires effective representations of both the program and the candidate proof so that sufficiency of the proof can be algorithmically checked. We used alternating finite automata in [9], Petri nets in [10], and a new notion of data automata in [11, 12] to represent various different classes of concurrent programs (and their proofs) for this purpose.

Here, we will focus on one instance of this methodology [11] from our past work, and discuss the ideas behind algorithmic solutions to the steps above over a running example.

Consider the program in Figure 2 where the illustrated code is executed by unboundedly many threads.This program implements the *ticket* mutual exclusion protocol. The safety property of interest is that no two threads are ever simultaneously in their critical sections. This program has two global integer typed variables `s` and `t`, while it has unboundedly many integer typed local variables `m`, one per each thread. The challenge of proving this property for this program, beyond the standard challenges of dealing with shared memory and infinite-state programs, is that the local variables of unboundedly many threads effectively require an unbounded sized memory to keep track of their values in the proof.

Figure 2 also illustrates the control flow automaton for each thread in this program. Let us start by looking at a *run* of this program that would lead to two threads being in a

$$\begin{array}{l}
\{s \le t\} \\
\quad m_1 \texttt{ = t++ :1} \\
\{s \le m_1 \ \wedge\ m_1 < t\} \\
\quad m_2 \texttt{ = t++ :2} \\
\{s \le m_1 \ \wedge\ m_1 < m_2\} \\
\quad \texttt{assume(}m_1 \le \texttt{s) :1} \\
\{s \le m_1 \ \wedge\ m_1 < m_2\} \\
\quad \texttt{assume(}m_2 \le \texttt{s) :2} \\
\{\text{false}\}
\end{array}$$

$$\boxed{\begin{array}{c}\{s \le t\} \\ m_1 \texttt{ = t++ :1} \\ \{s \le m_1\}\end{array}} \quad \boxed{\begin{array}{c}\{\text{true}\} \\ m_1 \texttt{ = t++ :1} \\ \{m_1 < t\}\end{array}}$$

$$\boxed{\begin{array}{c}\{m_1 < t\} \\ m_2 \texttt{ = t++ :1} \\ \{m_1 < m_2\}\end{array}} \quad \boxed{\begin{array}{c}\{s \le m_1 \ \wedge\ m_1 < m_2\} \\ \texttt{assume(}m_2 \le \texttt{s) :1} \\ \{\text{false}\}\end{array}}$$

Figure 3 Correctness proof of a run (left) and the Hoare triples extracted from it (right).

critical section at the same time. The run is a string accepted by the parallel composition of unboundedly many of versions of the control flow automaton illustrated in Figure 2. Below, is a run of the program with only two participating threads (blue and red), where each instruction includes the id of the thread executing it in form of $: i$ $(i \in \{1, 2\})$, and the two local variables for the two threads are distinguished by giving them two different subscripts for clarity:

$$m_1 \texttt{ = t++ } :1;\ m_2 \texttt{ = t++ } :2;\ \texttt{assume(}m_1 \le \texttt{s) } :1;\ \texttt{assume(}m_2 \le \texttt{s) } :2;$$

Note that the run does not correspond to a real execution of the program; in other words, it is *infeasible*. It is easy to argue that such a run is *infeasible* and therefore a spurious counter example to violation of mutual exclusion. One possible proof for this is illustrated in Figure 3. What can we learn from this proof that would help with the construction of a proof of correctness for the entire program?

A natural and minimal unit of reasoning that can be extracted from a proof of the correctness of a run is *a set of Hoare triples*. For example, the Hoare triples demonstrate in Figure 3 can be learned from the proof of the run in the same figure. Consider a deductive system in which these triples can be considered as axioms. What should be the inference rules of this deductive system that would produce new judgements based on a *finite* set of base axioms? The answer lies in the definition of a *proof space* [11].

A proof space $\mathcal{H}$ is a set of valid Hoare triples which is:

- Closed under symmetry: Let $\pi : \mathbb{N} \to \mathbb{N}$ be any index permutation.

$$\{\phi\}\ \langle\sigma_1 : \texttt{i}_1\rangle \cdots \langle\sigma_n : \texttt{i}_n\rangle\ \{\psi\} \in \mathcal{H} \implies \{\phi[\pi]\}\ \langle\sigma_1 : \pi(\texttt{i}_1)\rangle \cdots \langle\sigma_n : \pi(\texttt{i}_n)\rangle\ \{\psi[\pi]\} \in \mathcal{H}$$

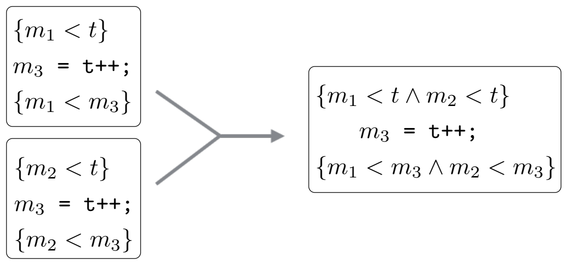
  For example

$$\boxed{\begin{array}{c}\{s \le m_1 \ \wedge\ m_1 < m_2\} \\ \texttt{assume(}m_2 \le \texttt{s);} \\ \{\text{false}\}\end{array}} \longrightarrow \boxed{\begin{array}{c}\{s \le m_3 \ \wedge\ m_3 < m_4\} \\ \texttt{assume(}m_3 \le \texttt{s);} \\ \{\text{false}\}\end{array}}$$

- Closed under conjunction:

$$\text{If } \{\phi\}\ \tau\ \{\psi\} \in \mathcal{H} \ \wedge\ \{\phi'\}\ \tau\ \{\psi'\} \in \mathcal{H} \implies \{\phi \wedge \phi'\}\ \tau\ \{\psi \wedge \psi'\} \in \mathcal{H}$$

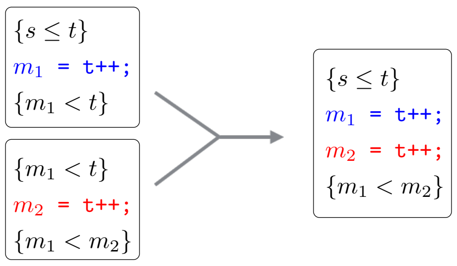
For example



- Closed under sequencing:

$$\text{If } \{\phi_0\} \;\; \tau_0 \;\; \{\phi_1\} \in \mathcal{H} \;\land\; \{\phi_1'\} \;\; \tau_1 \;\; \{\phi_2\} \in \mathcal{H} \;\land\; \phi_1 \Vdash \phi_1' \implies \{\phi_0\} \;\; \tau_0\tau_1 \;\; \{\phi_2\} \in \mathcal{H}$$

For example



Note how proofs of runs of arbitrary length can be constructed using the sequencing rule.

One can effectively consider symmetry, conjunction, and sequencing as the inference rules of the aforementioned deductive system, and a proof space as a theory of this system. Note that the symmetry rule is the one that facilitates reasoning about unboundedly many threads. Without it, sequencing and conjunction can be used to produce proofs of concurrent programs with a fixed number of threads [9].

The next natural question to ask is whether the closure (under symmetry, conjunction, and sequencing) of the set of Hoare triples in Figure 3 includes the proof of every run of the program in Figure 2. The answer is no. The complete proof corresponds to the *basis* illustrated in Figure 4 which includes one additional Hoare triple. This basis is a finite set of valid Hoare triples that generates the complete proof space for the ticket program through its closure under sequencing, symmetry, and conjunction rules. This additional Hoare triple can be mined out of a second sample run. Therefore, the proof of the program in Figure 2 is decomposed into two correctness arguments of two sample runs (one illustrated in Figure 3 and discussed above, and the other skipped for brevity).

It is important to note that the proof space does not make use of any of the exotic features that are common to logics for reasoning about concurrency, most notably auxiliary variables and universal quantification over threads.

How is the proof in Figure 4 checked? To reduce the problem of checking whether a candidate proof space serves as a correctness proof for a program, we developed the notion of predicate automata (PA), an infinite-state, infinite-alphabet generalization of alternating finite automata. Predicate automata are equipped with a finite vocabulary of predicates parameterized over natural numbers, and their states are propositions over this vocabulary. The transition function of a PA maps each predicate symbol and letter to a positive Boolean formula over its vocabulary. For example, the transition

$$\delta(p(i,j), a:k) = (p(i,j) \land i \neq k) \lor (q(i) \land q(j) \land i = k)$$

$\{s \le t\}$ `m₁ = t++ :1` $\{s \le m_1\}$	$\{\text{true}\}$ `m₁ = t++ :1` $\{m_1 < t\}$	$\{m_1 < t\}$ `m₂ = t++ :1` $\{m_1 < m_2\}$
$\{s \le m_1 \wedge m_1 < m_2\}$ `assume(m₂ ≤ s) :1` $\{\text{false}\}$	$\{s \le m_1 \wedge m_1 < m_2\}$ `s++ :1` $\{s \le m_2\}$	

Figure 4 Complete basis for the proof of ticket protocol in Figure 2.

indicates that, if the PA is at state $p(1,2)$ and reads $a:2$, then it transitions to $p(1,2)$; if it then reads $a:1$, then it transitions to both the state $q(1)$ and $q(2)$. A finite basis B of Hoare triples gives rise to a predicate automaton which recognizes the same set of runs as the proof space generated by B.

The *proof checking* problem for proof spaces reduces to the inclusion problem for predicate automata, which in turn reduces to the emptiness problem of predicate automata. Although this problem is undecidable in general, we proposed an algorithm which is a decision procedure for the special case of PAs where each predicate symbol in its vocabulary has arity at most one.

2 Dynamic memory

Separation logic is an extension of Hoare logic for reasoning about memory [21, 26]. It is based on the addition of a new logical connective, the *separating conjunction* $P * Q$, which asserts that the heap can be split into two disjoint parts such that P holds in one and Q in the other. Separating conjunction allows for *local reasoning* in the sense that a specification of a program fragment need only involve the portion of the heap that is relevant to that fragment – the rest (the *frame*) is automatically proved to remain untouched via the Frame rule, stated below:

$$\textsc{Frame}\quad \frac{\{P\}\ S\ \{Q\}}{\{P * F\}\ S\ \{Q * F\}}$$

The capacity of separation logic for reasoning about disjointness of memory makes it appealing for reasoning about concurrency. This power can be illustrated by the appealing inference rule for parallel composition [19]:

$$\textsc{Parallel Composition}\quad \frac{\{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 * P_2\}\ S_1 \parallel S_2\ \{Q_1 * Q_2\}}$$

This rule gives concurrent separation logic (CSL) an intuitive way to prove properties of concurrent programs where threads act on disjoint memory. Generally, it is too much to expect that threads do not share memory, but a kind of disjointness can be achieved through the notion of *ownership*. The essential idea can be summarized as follows [20]:

- *Ownership Hypothesis.* A code fragment can access only those portions of state that it owns.
- *Separation Property.* At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

```
type node = { data:  int; next:  node }
global top ::  node
push(item):                         pop():
  mtop := new node                    do:
  mtop.data := item                     mtop := *top
  do:                                   if(mtop = null):  break
    mtop.next := *top                   mnext := mtop.next
  while(!cas(top, mtop.next, mtop))   while(!cas(top, mtop, mnext))
```

Figure 5 A nonblocking stack.

The earliest variation of CSL managed ownership using resource invariants, following Owicki and Gries [22]. Subsequently, there has been a great deal of work on increasing the power of CSL, a recent survey of which appears in [3]. Each advance increases the scope of what is possible to prove. However, increasing the expressivity of these logics takes us farther away from the class of proofs that we know how to completely automate. We thus raise a challenge problem: *can we automate proofs for concurrent heap-manipulating programs using the ideas of proof spaces and separation logic?*

A natural starting point in adapting proof spaces to separation logic is to replace the Conjunction rule with the Frame rule. Call a set of valid separation logic triples closed under Sequencing, Symmetry, and Frame a *separation proof space*.

The power of separation proof spaces can be illustrated with the *push* routine of Treiber's non-blocking stack [29], illustrated in Figure 5. The difficulty in verifying memory safety of *push* is that no process owns the global `top` variable. There are various methods that can be used to verify memory safety and even full functional correctness of the lock-free stack [31, 30, 27, 8]. However, such proofs are difficult to automate. Figure 6 pictures a basis for a proof space that proves the memory safety of `push`. We observe the following features:

1. Separating conjunction and the Frame rule allow the proof to scale to an arbitrary number of threads by maintaining disjointness between each thread's local `mtop`.
2. Separation logic triples are ordinary – they do not make use of exotic features that require ingenuity (except perhaps the separating implication $-\!\!*$, the adjoint of $*$).
3. The stack is not owned by any process – the fact that `top` points to the top of the stack is preserved by every statement of the program, either explicitly or by application of the Frame rule.

There are three challenges in making the combination of proof spaces and separation logic practical.

1. **Inference rules.** While it may be possible to get some mileage out of the Frame rule combined with Sequencing and Symmetry, we believe that additional inference rules will be required in order to verify systems of practical interest. For example, existentially quantified variables commonly used in separation logic, and some means for reasoning about them is likely required (in the Treiber stack, for instance, we resorted to using separating implication to avoid proliferation of existential variables).
2. **Proof generation.** While there has been a great deal of work on generating correctness proofs for single runs in Hoare logic, there is relatively little in separation logic [1]. The additional challenge imposed by proof spaces is that we need methods to synthesize proofs that decompose into small, re-usable components.
3. **Proof checking.** The problem of checking that a proof space can prove every run of a program correct is reduced to the emptiness problem of predicate automata. This

$$\{\mathsf{emp}\}\quad \langle \texttt{mtop := new node} : 1\rangle \quad \{\texttt{mtop}_1 \mapsto [\_,\_]\}$$

$$\{\texttt{mtop}_1 \mapsto [\_,\_]\}\quad \langle \texttt{mtop.data := item} : 1\rangle \quad \{\texttt{mtop}_1 \mapsto [\_,\_]\}$$

$$\{\texttt{mtop}_1 \mapsto [\_,\_] * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$
$$\langle \texttt{mtop.next := top} : 1\rangle$$
$$\{\big((\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n)) \mathrel{-\!\!*} \mathsf{list}(\texttt{mtop}_1)\big) * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$

$$\{\texttt{mtop}_1 \mapsto [\_,\_] * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$
$$\langle \texttt{mtop.next := top} : 1\rangle$$
$$\{\texttt{mtop}_1 \mapsto [\_,\_] * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$

$$\{\big((\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n)) \mathrel{-\!\!*} \mathsf{list}(\texttt{mtop}_1)\big) * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$
$$\langle \texttt{assume(cas(\&top, mtop.next, mtop))} : 1\rangle$$
$$\{\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n)\}$$

$$\{\texttt{mtop}_1 \mapsto [\_,\_] * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$
$$\langle \texttt{assume(!cas(\&top, mtop.next, mtop))} : 1\rangle$$
$$\{\texttt{mtop}_1 \mapsto [\_,\_] * (\exists n.\texttt{top} \mapsto [n] * \mathsf{list}(n))\}$$

Figure 6 A basis for a separation proof space for Figure 5.

reduction exploits the duality between disjunction and conjunction to complement the language of a predicate automaton recognizing all runs that are proved correct by a proof space. This duality does not exist between disjunction and separating conjunction. Some new mechanism (e.g., a new class of automaton) is likely needed to encode the proof checking problem for separation proof spaces.

3 Recursion

The difficulty of analyzing concurrent programs with recursive procedures is well-known: reachability is undecidable even for simple program models [25]. From the perspective of constructing a proof by considering program runs, the set of runs of a (sequential) program without procedures is regular, and with procedures, it is context-free. However for concurrent programs, adding procedures jumps over the context-freedom, requiring a *multi-stack* automaton to recognize the language of runs.

There are (at least) two possibilities for solving this problem: (1) use an automaton model that is capable of recognizing the languages of multi-threaded recursive program runs; and (2) use summaries to treat complete procedure calls as atomic actions, restoring regularity to the program model. We will discuss both possibilities in turn.

The advantage of using an automaton model that can recognize the interleaved runs of recursive procedures is that the proof system used for proving the correctness of those runs may remain unchanged. In our framework the structure of the proof is divorced from the structure of the program. For example, a proof space may very well cover all the runs of a multi-threaded program with recursive procedures – the added difficulty is only in the proof *checking* problem, where we must *prove* that it does. This problem requires us to devise a class of automata capable of recognizing the language of runs in the program *and* the language of runs proved correct in a proof.

```
int g = 0;
void foo(int r) {
  if (r == 0) {
    foo(r);
  else {
  g++;
  }
  return;
}
```

```
void main() {
  int c = *;
  foo(c);
  assert( g >= 1 );
  return;
}
```

Figure 7 An example recursive concurrent program with unbounded recursion from [24].

In fact, predicate automata are already a suitable model for this task. Procedure calls can be simulated by fork/join parallelism: whenever a thread would make a recursive call, it instead forks a new thread to make the call on its behalf and then immediately joins to retrieve the result. Fork/join parallelism can be modelled with predicate automata using a binary predicate $\mathit{child}(i, j)$ that retains the information that thread j was forked by thread i. However, admitting a binary predicate into the vocabulary of a predicate automaton makes the emptiness problem undecidable. Thus, the research problem is: *can we develop adequate semi-algorithms for emptiness checking for general predicate automata*?

The classical tool for context-sensitive analysis of recursive procedures in sequential programs is to compute *summaries* that abstract the behaviour of each procedure call [28]. Shared memory concurrency breaks the abstraction barrier: the assumption that procedure calls execute atomically will cause us to miss potential bugs. In [24], for the case where concurrent programs have a *fixed number of threads*, a summarization-based technique is proposed where atomicity is argued through Lipton's theory of reduction.

Although naive summarization (that is summarization that is unaware of the environment of concurrently executing threads) is unsound in general, in many cases a software developer's mental model likely assumes *some* atomic specification for each procedure. Consider the example in Figure 7. This example was taken from [24] where the program consisted of parallel composition of two threads each executing the `main` procedure, however the same assertion holds true for unboundedly many threads running `main` in parallel. The main procedure has a local variable `c` which is initialized nondeterministically. Procedure `main` then calls `foo` with `c` as the actual parameter. Procedure `foo` falls into infinite recursion if the parameter r is 0. Otherwise, it increments global variable `g` and returns. After returning from `foo`, the main procedure asserts that `(g >= 1)`. The specification that `foo` increases `g` by 1 or does not return at all is too strong: two threads may concurrently execute `foo` and increment `g` by 2. However, the specification $\{\mathtt{g} \geq 0\}$ $\mathtt{foo}(c)$ $\{\mathtt{g} \geq 1\}$ is unproblematic (i.e. holds under environment interference) and strong enough to prove that the assertion in `main` holds. Note that this is independent of an argument about the atomicity of the procedure, that is in the style of [24]. This is about atomicity of the specification in a concurrent environment independent of atomicity of the code. The relevant research question is then: *how can our framework discover these atomic specifications for procedures and prove that they hold?*.

References

1 Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. Spatial interpolants. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 634–660, 2015.

2 Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.

3 Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.

4 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 428–439, 1990.

5 Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

6 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

7 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

8 Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper - automatic verification for fine-grained concurrency. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 420–447, 2017.

9 Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 129–142, 2013.

10 Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 151–164, 2014.

11 Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proof spaces for unbounded parallelism. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 407–420, 2015.

12 Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 185–196, 2016.

13 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, University of Liege, 1994.

14 Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *TACAS*, pages 262–276, 2009.

15 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

16 Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

17 Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993.

18 Kenneth. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

19 Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 49–67, 2004.

20 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

21 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, pages 1–19, 2001.

22 Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *CACM*, 19:279–285, May 1976.

23 Doron Peled. All from one, one for all: On model checking using representatives. In *CAV*, pages 409–423, 1993.

24 Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 245–255, 2004.

25 G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, March 2000.

26 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.

27 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87, 2015.

28 Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, chapter 7. Prentice-Hall, Inc., 1981.

29 R. Treiber. Systems programming: coping with parallelism. Technical report, Almaden Research Center, 1986.

30 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 377–390, 2013.

31 Viktor Vafeiadis. Rgsep action inference. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 345–361, 2010.

32 Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, pages 491–515, 1991.