Staged Concurrent Program Analysis

Nishant Sinha, Chao Wang
NEC Labs America, Princeton, NJ, USA.
{nishants,chaowang}@nec-labs.com

ABSTRACT

Concurrent program verification is challenging because it involves exploring a large number of possible thread interleavings together with complex sequential reasoning. As a result, concurrent program verifiers resort to bi-modal reasoning, which alternates between reasoning over intra-thread (sequential) semantics and inter-thread (concurrent) semantics. Such reasoning often involves repeated intra-thread reasoning for exploring each interleaving (inter-thread reasoning) and leads to inefficiency. In this paper, we present a new two-stage analysis which completely separates intra- and inter-thread reasoning. The first stage uses sequential program semantics to obtain a precise summary of each thread in terms of the global accesses made by the thread. The second stage performs inter-thread reasoning by composing these thread-modular summaries using the notion of sequential consistency. Assertion violations and other concurrency errors are then checked in this composition with the help of an off-the-shelf SMT solver. We have implemented our approach in the FUSION framework for checking concurrent C programs shows that avoiding redundant bi-modal reasoning makes the analysis more scalable.

Categories, Subject Descriptors: D.2.4 [Software/Program Verification]: Model Checking, Formal Methods.

General Terms: Algorithms, Verification.

Keywords: Thread-modular Summarization, Interference Abstraction, Interference Skeleton, Staged Analysis, Sequential Consistency, Axiomatic Composition, SMT solvers.

1. INTRODUCTION

Checking properties of shared memory based concurrent programs statically with model checking is expensive because it amounts to exploring large number of interleavings of the concurrent threads. Methods often ameliorate this cost by using partial order techniques [5, 14, 11] and causal orderings imposed by synchronization primitives, e.g., locks. Unfortunately, most of the methods, whether explicit [5, 14, 11] or symbolic [29, 13, 21, 20, 34], resort to redundant *bi-modal* reasoning: the analysis *alternates* between reasoning over the *intra*-thread and the *inter*-thread semantics. For example, consider two concurrent threads T_1 and T_2 as follows:
$T_1 : (x := 3; t := x; a := t + 1; b := a + 3; assert(b > 4);)$
$T_2 : (x := 5;)$

The thread T_1 contains an assertion $(b > 4)$; to check this assertion, various interleavings of T_1 and T_2 must be considered (inter-thread reasoning), based on when the statement $(x := 5)$ in T_2 is executed. Once the value of x is obtained $(x \in \{3, 5\})$, the statements $(t := x; a := t + 1; b := a + 3)$ in T_1 are composed via intra-thread reasoning to check the assertion. Bi-modal reasoning of this form is inherently wasteful because the analysis engine is forced to perform reasoning over similar intra-thread program regions repeatedly. Intra-thread (or thread-modular) program summarization is a possible solution to this problem. However, classical program summarization methods [31, 30] designed for sequential programs are not applicable in a concurrent setting because of interferences on shared locations from concurrent threads.

In this paper, we present a *staged* concurrent analysis, which avoids redundant bi-modal reasoning. The first stage (summarization) consists of a new algorithm to summarize the individual program threads in the presence of concurrency. The summary is represented in form of an *interference skeleton*, which is a partially-ordered set of all global read and write accesses of program threads. The second stage performs composition symbolically by encoding feasible *linearizations* of the above partial order using a sequential consistency (SC) criterion [1] on the global accesses. Finally, we check property violations by using an SMT solver [9, 8] which searches for a linearization that violates the property. Note that because the two stages perform either intra- or inter-thread reasoning, we achieve a complete separation between intra- and inter-thread reasoning, and thus avoid costly bi-modal alternation between them.

The key idea behind summarization in the first stage is that of *interference abstraction*: any read access to a shared memory location (global read) in the program is abstracted by a fresh symbolic variable. These fresh variables (linear in the number of reads) model arbitrary interference to the shared location via writes in the same or a concurrent thread. Interference abstraction, in effect, enables thread-local summarization, while assuring us that the interferences due to concurrent writes can be taken into account lazily and precisely in the subsequent composition stage.

The second stage linearizes the skeleton by *linking* the read accesses with appropriate write accesses in the skeleton. *Axiomatic* composition (AC) provides a natural way to do this (as opposed to introducing an explicit scheduler), i.e., we can use the sequential consistency (SC) axioms [1], specified in first-order logic, to enforce that the linearization corresponds to a feasible concurrent program trace. AC has been employed to check consistency of concurrent data structures under relaxed hardware memory models (e.g., [2]). In contrast, we propose to employ AC in the new setting of high-level static analysis of concurrent programs. In this setting, it is sufficient to consider only the SC model of execution for performing high-level static analysis (as opposed to more relaxed models). The central problem is how to encode SC efficiently to obtain a scalable analysis. To this goal, we propose a method to *prune* redundant SC constraints by analyzing the interference skeleton computed in the first stage.

Our staged analysis handles arbitrary concurrent C programs (e.g., using `Pthread` libraries) using pointers and arrays with the help of a precise memory representation [4]. As opposed to most concurrent analyses, program threads need not be demarcated beforehand; our analysis handles thread creation and destruction natively. Analysis of general recursive programs even with finite data is known to be undecidable. To analyze arbitrary C programs over infinite data types, we transform the input programs to their *structurally bounded* versions, where loops and recursive functions in the input program are unrolled to a fixed depth. Finite unrolling of this form indirectly finitizes the number of threads, and also fixes the size of heap that the bounded program may access. As a result, the analysis of bounded programs becomes decidable.

We have implemented our approach in the FUSION framework [34] for verification of concurrent C programs. Preliminary evaluation show that summarization and optimizations during composition are essential for scalable symbolic analysis of concurrent programs. The key contributions of this paper are as follows:

- A staged analysis algorithm for verifying concurrent programs consisting of (Stage 1) a precise thread-modular summarization of individual threads, and (Stage 2) an optimized composition step over the summary using sequential consistency axioms, followed by checking assertion violations using an SMT solver.
- A thread-modular data flow analysis based on interference abstraction to summarize structurally-bounded concurrent programs in form of an interference skeleton.
- Optimized composition of the interference skeleton by systematically pruning redundant SC constraints between the global accesses by static analysis of the skeleton.

1.1 Redundant bi-modal reasoning

```
        Thread 1 (T1)              |      Thread 2 (T2)
1   ...                            |
2   S1: a0 = x;                    | 1  S3: x = 0;
3   a1 = a0 + 1; z1 = a1;          | 2  ...
4   a2 = a1 + 1; z2 = a2;          | 3  S4: x = 5;
5   ...                            | 4  ...
6   a99 = a98 + 1; z99 = a99;      | 5  A: assert  (x == 5
7   a100 = a99 + 1;                | 6             || x >= 105);
8   S2: x = a100;                  |
```

Consider the fragments of concurrent threads T_1 and T_2 shown above; the variables $a0 - a100$ are local, and the rest (x, $z1 - z99$) are shared; our goal is to check the assertion A at line 5. T_1 and T_2 may interleave in many possible ways; an efficient analysis based on say, partial order reduction [5, 11] will consider a representative set of such interleavings, and compute the value of x at the assertion A. Analyzing each interleaving amounts to bi-modal reasoning which alternates between intra- and inter-thread reasoning. For example, consider the interleaving S3-S4-S1-S2-A: the analysis first considers T_2, setting x to 3 and 5 at S3 and S4 successively (intra-thread reasoning), and then switches to T_1 (inter-thread reasoning), successively computing $a0 - a100$, $z1 - z99$ and x (intra-thread), and finally, switches back to T_2 to check A using the computed value of x. This form of bi-modal reasoning is *redundant*, since similar intra-thread reasoning is repeated for each interleaving considered. For example, note that $a100 = (a0 + 100)$ irrespective of the interleaving of T_1 and T_2 considered. However, to analyze another interleaving, say S3-S1-S4-S2-A, the values $a1 - a100$ must be *recomputed* because $a0 = 0$ now as opposed to $a0 = 5$ in the previous interleaving. Redundant intra-thread reasoning (or composition) takes its toll on both explicit [5, 14, 11, 36] and symbolic [13, 21, 20, 34] analysis techniques, often decreasing their performance by an order of magnitude. Although summarization techniques are well-known for sequential program analysis [31, 30], they cannot be directly employed here: because of interference [26] on shared locations by concurrent threads, the value read from a shared location in a thread may not be the same as the previous value written to the location in the same thread.

There exist techniques that ameliorate this problem by *collapsing* a set of intra-thread transitions into a single one, e.g., path reduction [36]. However, they can only collapse transitions inside *transactions*, i.e., regions without any concurrent interference. Given a transaction between locations l_1 and l_2, path reduction collapses all the transitions between l_1 and l_2 to a single transition. Unfortunately, these techniques are not effective across arbitrary program regions or transactions. In the above example, assignments to shared variables $z1 - 99$ represent locations where interference may occur, which alternate with assignments to variables $a0 - a100$. Therefore, path reduction methods cannot collapse the assignments to infer that $a100 = (a0 + 100)$.

Our method avoids this problem by using the idea of interference abstraction, which, in turn, enables data-flow based summarization. First, the value of shared variable x read at line 2 in T_1 is abstracted by a fresh symbolic variable, say r_0. Since the assignments to variables $a1 - a100$ are not influenced by any other interferences, our data-flow propagation successfully computes that $a100 = r_0 + 100$. Further, our method precisely summarizes all local control and data flow in terms of global accesses, e.g., assignment to $z1$ at line 3 in T_1 gives rise to a global access event $Z1$ with value $val(Z1) = r_0 + 1$. Similarly, global accesses $Z2 - Z99$ are computed with their relative order ($Z1 \prec Z2 \prec Z3 \ldots$).

Lal and Reps [21] present address this problem in the setting of context-bounded analysis (CBA) of programs with finite-domain variables (extended to C programs in [20]). In CBA, a concurrent program is analyzed assuming only a fixed number of context switches occur between threads. Context-bounding allows fixing a set of context switch (CS) locations and model the interference at the CS locations only. This is done by *guessing* (or abstracting) the value of the global state at each CS location. Since no interference is possible between each pair of CS locations in a thread and the number of CS locations is finite, sequential summarization can be applied between each CS location pair. Guessing the global states, however, involves *unnecessary duplication* of the shared state at each CS location. This is because each program location may modify only a few shared variables, and hence it is extremely inefficient to duplicate all shared variables at every location.

For example, the method [21] will duplicate all the global variables ($z1 - z99$, x) at all locations in thread T_1 and compute summaries between all pairs of locations. This summarization is quadratic in the number of thread locations and the global variables and incurs a high overhead. Further, in the improved lazy algorithm [21], the summaries cannot be reused across multiple interleavings if the global state at a CS location is different across interleavings, which causes redundant bi-modal reasoning. In contrast, our method avoids bi-modal reasoning and the number of the global access events as well as the fresh variables introduced in our summaries is linear in the number of shared variable accesses, thus enabling a practical analysis. Moreover, it is possible to obtain CBA as a special case by fixing the context bounds in our analysis.

2. OVERVIEW

CCFGs. We represent concurrent programs in form of concurrent control flow graphs (CCFGs), which can be viewed as an extension of control flow graphs (CFGs) for sequential programs to concurrent programs. A CCFG = (V, E), consists of a set of nodes V and a set of edges E. Each edge in E is labeled by a guard condition g and a (possibly empty) set of assignments of form $(lhs := rhs)$. Intuitively, the assignments are executed iff the guard condition holds. The set of nodes V contains two special nodes FORK and JOIN, to model thread creation and termination, respectively: a FORK (JOIN) node has a single incoming (outgoing) edge, and multiple outgoing (incoming) edges. Individual program threads are modeled as sub-graphs of the CCFG. Function calls are modeled in the standard way [31] with call and return edges labeled by assignments to parameters and return variables respectively.

Synchronization constructs, e.g., mutex, condition variables, etc., are modeled using shared variables. For example, acquiring lock lk in thread T_i is modeled as an edge with guard (lk==0) and assignment ($lk := i$). Instantaneous *test-and-set* primitives are modeled by marking the corresponding sub-graphs of the CCFG as *atomic*, which are referred to as atomic regions. The assertions in the original program are transformed into *error nodes* while constructing the CCFG; assertion checking reduces to checking if there exists a feasible interleaving of concurrent thread paths in the CCFG that terminates at the error node. We distinguish the two kinds of join locations in the CCFG: the *intra*-thread joins occur due to path merging inside a thread, while the *inter*-thread join corresponds to the JOIN nodes. An example is presented in the next section.

Read and Write Accesses. We refer to each read or write to a memory location as a read or write access respectively. A memory location l is said to be shared if more than one thread reads or writes to l. In the following, we will mainly concern with accesses to shared memory locations, called *global accesses*. Each global access e is represented using a symbolic tuple (loc, val, occ), where $loc(e)$ and $val(e)$ correspond to the memory location and the value that is read/written during the access e, and $occ(e)$ is the necessary condition for e to occur. A global access e_1 is said to *interfere* with another global access e_2 if one of them is a write and $loc(e_1) = loc(e_2)$ is satisfiable. Note that every usage of the phrase 'global access' in this paper is implicitly identified with this tuple representation (loc, val, occ).

A Motivating Example. Consider the multi-threaded C program based on the `Pthread` library shown in Fig. 1(a) . The program contains a single shared variable x. Two threads are created from the main thread, which read and write x. Fig. 1(b) shows the corresponding concurrent control flow graph (CCFG). In the CCFG, FORK and JOIN represent thread creation and termination points. The CCFG consists of sub-graphs for three threads, main (nodes: `1`, FORK, Join, `10`, ERR), t_1 (nodes: `2-9`) and t_2 (nodes: `2'-9'`). For brevity, we have merged multiple consecutive FORK and JOIN nodes into a single node. Moreover, new assignments have been added to ensure that each statement makes *at most one* global access. Fig. 1(b) also shows the global accesses in the CCFG: $W1$, $W2, W3, W2', W3'$ are the global writes, while $R1, R2, R3, R1'$, $R2', R3', R4$ are the global reads. Let us see how our staged analysis works on the given CCFG.

Stage 1. The first stage performs a data flow analysis (Sec. 4) on the CCFG is used to compute a summary in form of a *interference skeleton* ($IS = (S, \prec_S)$). The IS summarizes the CCFG in terms of global accesses S and their partial order $\prec_S$. Starting from the entry node of CCFG, the analysis iteratively computes and propagates symbolic data consisting of a *path condition* and a *local state*. During propagation, each global read access is assigned a fresh symbolic value (interference abstraction), and the global write accesses are computed in terms of these symbolic values. Fig. 1 shows the details of the global accesses in the skeleton IS. The accesses $R1, R2, R3$ are assigned symbolic values r_1, r_2, r_3. Note that even though $R1$ and $R2$ are consecutive accesses to x, they are assigned different symbolic values r_1 and r_2. This allows us to take arbitrary interference (writes) from concurrent threads into account during composition. The analysis also collects the path conditions under which the global accesses happen, e.g., $W2$ occurs if the condition $occ(W2) = (r_1 < 1)$ holds.

The analysis merges the propagated data at the *join* points in a precise *path-sensitive* manner, to avoid (potentially exponential) path enumeration in the CCFG. At the intra-thread join points (cf. Sec. 2), e.g., node 9, the path conditions are *disjuncted* following the standard sequential semantics, while at the inter-thread join node (node JOIN), the path conditions are *conjuncted* to ensure simultaneous reachability of the node by all threads. To check assertion violations, we compute *error conditions* at the error nodes, which correspond to the computed path conditions at the node, e.g., $\phi = (r_4 \neq 3)$ at ERR node. These error conditions are checked during the second stage. The analysis also computes a partial order $\prec_S$ (see Fig. 1(c)) denoting the relative order of events. Note that IS abstracts away all the thread-local control and data flow from the CCFG and only contains the global access information. Computing the interference skeleton is non-trivial for arbitrary C programs (with pointers and complex data types); we present the full algorithm in Sec. 4.

Stage 2. The second stage of our analysis explores the feasible concurrent behaviors of the CCFG by performing inter-thread composition. Note that in the skeleton IS computed above, the values of the global reads are unconstrained symbolic variables. The composition step constrains these values by *linking* them to the global writes (cf. Sec. 5). Note that we cannot link reads with writes arbitrarily, because we only desire feasible program behaviors during composition, e.g., the read access $R2$ cannot be linked to $W2$ which follows $R2$ in the program order. The notion of *sequential consistency* (SC) [1] enables us to find a suitable relation between the reads and writes systematically: SC constraints enforce that each read access R must link with *some* write access, say W, such that both access the same memory location, the value written by W is the value read by R, and W must be the last such write that happens before R in an execution trace. In order to capture the feasible executions during composition, we add SC constraints between the reads and writes in IS (Sec. 5). For example, the SC constraints relating $R2$ ($loc = @x$, $val = r_2$, $occ = (r_1 < 1)$) to $W2'$ ($loc = @x$, $val = (r'_2 + 1)$, $occ = (r'_1 < 1)$) are of form:

$$copy(R2, W2') \Rightarrow (r_2 = r'_2 + 1) \wedge (r'_1 < 1) \wedge HB(W2', R2)$$

where the predicate $copy(R2, W2')$ denotes that $R2$ is linked to $W2'$ and $HB(W2', R2)$ denotes that $W2'$ must happen before $R2$. Constraints enforcing that no other write happens between $R2$ and $W2'$ and that $R2$ must link with some write are also added (see Sec. 5). In Sec. 6 we show how to add SC constraints in an optimized way to *prune* redundant constraints. Finally, the error condition $\phi = (r_4 \neq 3)$ is checked for feasibility, together with the encoding of IS and SC constraints by an SMT solver [9, 8]. If the constraints are satisfiable, then a sequence of accesses in IS is obtained, e.g., ($W1, R1, R1', R2, W2, R2', W2', R4$) for the above example. This sequence is then mapped to the CCFG to obtain a violation witness. By separating the intra-thread summarization in Stage 1 with inter-thread composition in Stage 2, our analysis is able to avoid redundant bi-modal reasoning completely.

3. MODELING C PROGRAMS

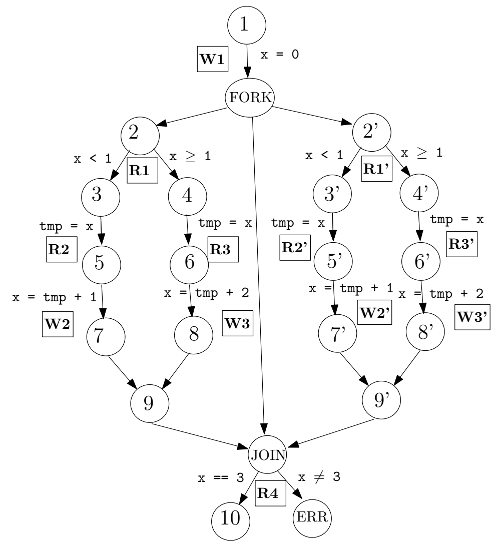
We first describe how to transform an arbitrary concurrent C program to a simplified intermediate program by adopting a memory representation which consists of a global memory map together with local memory maps for each program thread. The simplified program is then structurally bounded and represented as a *bounded* CCFG, which is used in our analysis.

3.1 Program Transformation

In order to handle C program constructs like pointers, arrays and structures uniformly, we fix a memory representation for our analysis in a manner similar to the HAVOC tool [7, 20]. Indirect memory accesses are handled using a memory map `Mem`, which models the program heap by mapping a memory location (address) to a symbolic value. All variables and objects whose address can be taken

```
int x;

void add_global ()
{
  if ( x < 1 ) x = x + 1;
  else x = x + 2;
}

int main (int argc, char *argv[])
{
  pthread_t t1, t2;
  x = 0;
  pthread_create(&t1, NULL, NULL,
 add_global);
  pthread_create(&t2, NULL, NULL,
 add_global);

  pthread_join(t1);
  pthread_join(t2);

  assert(x == 3);
}
```

(a)



(b)

$$\begin{tikzcd}
 & & \mathbf{W1} \arrow[dl] \arrow[dr] & & \\
 & \mathbf{R1} \arrow[dl] \arrow[d] & & \mathbf{R1'} \arrow[d] \arrow[dr] & \\
\mathbf{R2} \arrow[d] & \mathbf{R3} \arrow[d] & & \mathbf{R2'} \arrow[d] & \mathbf{R3'} \arrow[d] \\
\mathbf{W2} \arrow[drr] & \mathbf{W3} \arrow[dr] & & \mathbf{W2'} \arrow[dl] & \mathbf{W3'} \arrow[dll] \\
 & & \mathbf{R4} & &
\end{tikzcd}$$

(c)

Access	loc	val	occ
$W1$	$@x$	0	$true$
$R1$	$@x$	r_1	$true$
$R2$	$@x$	r_2	$r_1 < 1$
$W2$	$@x$	$r_2 + 1$	$r_1 < 1$
$R3$	$@x$	r_3	$r_2 \geq 1$
$W3$	$@x$	$r_3 + 2$	$r_2 \geq 1$
$R4$	$@x$	r_4	$true$

(d)

Figure 1: Example: (a) A multi-threaded C program with two threads, (b) its concurrent control flow graph (CCFG), and its global summary consisting of (c) the relative order of global accesses and (d) the values of the global accesses. The values for only unprimed accesses are shown: primed access values are similar. The memory location for variable x is denoted by $@x$.

are allocated on the heap. The address of a variable v is a fixed value denoted by $@v$. Let offs (f) denote the integer offset of the location of a field f inside its enclosing structure. Using the above map, we can transform the program statements (denoted by operator $\mathcal{T}$) as follows: (i) $\mathcal{T}(e \to f) = \texttt{Mem}[\mathcal{T}(e) + \texttt{offs}(f)]$, (ii) $\mathcal{T}(*e) = \texttt{Mem}[\mathcal{T}(e)]$, (iii) $\mathcal{T}(\&e \to f) = \mathcal{T}(e) + \texttt{offs}(f)$, (iv) $\mathcal{T}(e[i]) = \texttt{Mem}[\mathcal{T}(e) + i * \texttt{stride}(e)]$, where $\texttt{stride}(e)$ denotes the size of array e's type. All C program statements with indirect accesses can be transformed using the above rules [7, 20].

Shared variables. To detect shared variable accesses in concurrent programs with pointers, we use a conservative flow- and context-insensitive pointer analysis algorithm by Steensgaard [32]. All the variables that are declared as globals in the program or belong to an Steensgaard equivalence class [32] containing at least one globally declared variable, are said to be *shared*. Based on this, we partition the single memory map Mem above into (i) a *shared memory map* G, to denote the map containing the shared variables, and maps $\mathcal{L}_k$ to denote the local memory map for thread with identifier k. The domains of G and $\mathcal{L}_k$ maps are disjoint from each other (contain different memory locations), thus creating a valid partition. In contrast to previous approaches which partition the memory maps based on type- or field safety [7, 20], the above partition is more fine-grained and therefore improves the staged analysis by reducing the number of conflicting memory accesses.

All program statements are rewritten in terms of the above partition, e.g., a statement of form `l = (*p);` where l and p are local and shared respectively, is re-written as `l = G[p];`. As a result, we can now identify all global accesses in the program syntactically. Variables whose address is not taken in the program are referred to by their names, as before. Moreover, we rewrite the program statements so that no statement may perform more than one global read or write, i.e., no statement may contain more than one occurrence of G. For example, suppose a thread T contains a statement `x = (*p);` where both p and x are shared variables; this is rewritten as `lp = G[&p]; ap = G[lp]; G[&x] = ap;` where `lp` and `ap` are fresh variables local to T.

3.2 Structural Bounding

Analyzing concurrent programs with recursion is undecidable. We, therefore, obtain decidability by *structurally bounding* the concurrent program by unrolling loops and recursive functions to finite depth. Structural bounding ensures finite number of threads and heap size; we refer to the CCFG of the bounded program as *bounded CCFGs.* Our method then analyzes these bounded CCFGs for concurrent reachability properties (e.g., assertion violations or data races). The presented analysis is sound and complete with respect to these bounded CCFGs. Note that although we only consider bounded CCFGs, the CCFG representation is essential for modeling real-world programs since it allows specifying thread creation and destruction, and the relative order between threads (cf. Fig. 1). Both these aspects are not handled by most concurrent analyses.

4. THREAD-MODULAR SUMMARIZATION

The first stage of our analysis computes a thread-modular summarization of the CCFG: summarization gets rid of both local control and data flow in each thread and represents them precisely in terms of global accesses.

Global Skeleton. The analysis summarizes the CCFG in form of a interference skeleton $IS = (S, \prec_S)$, where S consists of the set of global accesses in the CCFG and $\prec_S$ denotes a partial order on elements of S. Recall that each access e in S contains the corresponding symbolic location $loc(e)$, value $val(e)$, and the occurring condition $occ(e)$. Each access e in S is global; hence, the $loc(e)$ values correspond to memory locations in G.

Thread-modular summarization is done using a precise data flow analysis that explores the CCFG in the standard reverse post-order [24] of the nodes while computing the *symbolic data* facts at each node of the CCFG and propagating the facts to the successors.

Symbolic Data. The data computed at a node n is a tuple of form $\langle \psi, \mathcal{L}, E \rangle$, where (i) ψ is the path condition formula for the set of paths reaching n, (ii) $\mathcal{L}$ is the local memory map for the thread that n belongs to, and (iii) E denotes the set of global accesses which happen immediately before (reach) the current location. We use program expressions (or terms) to represent both ψ and $\mathcal{L}$ precisely during the analysis. Intuitively, ψ captures the reachability condition for the node n, $\mathcal{L}$ captures the local state (map from memory locations to their symbolic values) at n and E is used to compute the interference skeleton iteratively. We also refer to the above tuple as the *symbolic state* s, and its fields as $s.\psi$, $s.\mathcal{L}$ and $s.E$.

Symbolic Summary. Given a fragment F of the CCFG (e.g., a function) having unique entry and exit nodes, the thread-modular summary of F consists of (i) a interference skeleton $IS = (S, \prec_S)$ over global accesses S in F, and (ii) a symbolic state $\langle \psi, \mathcal{L}, E\rangle$ at the exit node of F, where ψ, $\mathcal{L}$ and E denote the path condition, local map and the reaching accesses at the exit node, in terms of the input state map at the entry of F. Note that in the case where the fragment F (e.g., a function body) contains no global accesses, the function summary reduces to the traditional sequential function summary [31, 30] of form $\langle \psi, \mathcal{L}\rangle$, which represents the function outputs in terms of its inputs. For ease of presentation, we first describe the analysis assuming that all function calls are inlined in the CCFG. Subsequently, we discuss the general inter-procedural summarization algorithm.

Error Conditions. Recall that assertions are transformed to error node monitors in the CCFG. Our analysis retains these nodes in the skeleton IS and computes the symbolic state s at these nodes. The corresponding path condition $s.\psi$ is used to check precise reachability of the nodes during the composition stage.

Data-flow analysis. A well-known technique for precise program exploration is symbolic execution [19], which assumes symbolic values for program inputs and propagates the state (represented as program expressions) along all feasible program paths. Our data-flow analysis may be viewed as a form of symbolic execution for concurrent programs, with two key differences. First, we avoid costly path enumeration (as in symbolic execution) by merging symbolic data at the join locations (intra- and inter-thread joins, see Sec. 2) in a precise path-sensitive manner. Second, we avoid exploring exponential number of thread interleavings by performing interference abstraction: each global read access is assigned a fresh symbolic variable (placeholder). These placeholders model arbitrary concurrent writes to the read location; propagating these placeholders enables sequential (thread-modular) summarization in the presence of concurrency. The analysis propagates only the local state through the CCFG; the computed global accesses are not propagated but are used to construct the interference skeleton IS.

Figure 2 presents the rules for propagating data through a CCFG fragment to be summarized. They consist of rules for initialization (INIT) at the entry node of the CCFG, propagating data through guarded edges (GUARD), assignments with only local accesses (ASGN-LOC), assignments with global accesses (ASGN-GLB-R, ASGN-GLB-W), splitting data at the FORK node (FORK), and merging data at intra-thread (INTRA-JOIN) and inter-thread (INTER-JOIN) joins. The incoming data at a node n is denoted by $I_n(\langle \psi, \mathcal{L}, E\rangle)$; read-/write accesses e are represented as tuples $(loc(e), val(e), occ(e))$. The summary of the fragment F consists of the skeleton IS together with the data computed at the exit state of F. Let us consider these rules in more detail.

Assignments. Recall that CCFG assignments (cf. Sec. 3) either perform global accesses via shared map G, or local accesses via the map $\mathcal{L}$. Since no statement accesses G more than once, we consider three kinds of assignments: (global) $lhs := G[e]$, $G[e] := rhs$, and (local) $lhs := rhs$. Let us assume that a procedure $eval(e, \mathcal{L})$ evaluates expression e in the local memory map $\mathcal{L}$: this is done by employing the standard first-order logic operators $select(\mathcal{L}, l)$ and $store(\mathcal{L}, l, v)$ for manipulating arrays (cf. [23]), where l ranges over memory locations and v over values stored at these locations. To handle a local assignment $lhs := rhs$ (ASGN-LOC), the analysis first obtains the location by evaluating lhs in $\mathcal{L}$ ($eval(lhs, \mathcal{L})$), followed by evaluating rhs to obtain the new value v, and finally computing the update $store(\mathcal{L}, l, v)$ which is propagated. An assignment accessing the shared map (containing $G[e]$) is handled differently since it creates a global access event. Suppose a node n with assignment $G[e] := rhs$ has the incoming data $I_n(\langle \psi, \mathcal{L}, E\rangle)$. The rule ASGN-GLB-W handles this by creating a global write access $W = (l, r, \psi)$ with location $l = eval(e, \mathcal{L})$, value $r = eval(rhs, \mathcal{L})$ and the occuring condition ψ (path condition at n). Moreover, the skeleton IS is updated by adding W and partial orders between the reaching accesses in E and W. Similarly, the rule ASGN-GLB-R for handling $lhs := G[e]$ updates IS with a global read R, where the value of R is a fresh symbolic variable r (interference abstraction).

Handling Pointers. Recall that we model indirect accesses via pointers in an uniform manner by employing a precise memory representation using maps G and $\mathcal{L}$ (cf. Sec. 3). Note that by using $select$ and $store$ operators for manipulating symbolic data, we can handle arbitrary indirect memory accesses to $\mathcal{L}$ via pointers or arrays, in an implicit manner, without explicitly computing the alias sets of these pointers. Indirect memory accesses to the shared map G are captured by the location expression $loc(e)$ for each global access e; the subsequent composition stage employs $loc(e)$ to check for interfering accesses.

Forks and Joins. The analysis merges the data at join locations in the CCFG in a precise path-sensitive manner. At intra-thread joins (INTRA-JOIN), the incoming maps $\mathcal{L}_1$ and $\mathcal{L}_2$ are merged using an *if-then-else* operator to retain path-sensitivity while the path conditions ψ_1 and ψ_2 are disjuncted. At inter-thread joins, the local map for the child thread ($\mathcal{L}_c$) is discarded and the path conditions ψ_p and ψ_c conjuncted: this models the fact that both the parent and the child threads must execute the join location together. Note that the analysis creates a new local map $\mathcal{L}_c$ for the child thread at the thread creation node (FORK), which is discarded at the thread destruction node (JOIN).

By handling statements, forks and joins precisely during data-flow analysis and using interference abstraction for global reads, the algorithm gets rid of all local control and data flow in the CCFG: they are summarized to precise relations between global read and write accesses. Together with precise composition in the next stage, our analysis becomes sound and complete with respect to the bounded CCFG. Note that for the example in Sec. 1.1, summarization will be able to infer that $a100 = (a0 + 100)$ and hence repeated intra-thread reasoning is avoided during composition.

Example. The analysis of CCFG in Fig. 1 proceeds as follows. First, create a write access $W1 = (@x, 0, true)$ at node `1`. At the FORK node, initialize the local maps for thread $t1$ and $t2$, $\mathcal{L}_1$ and $\mathcal{L}_2$, to M_1 and M_2 respectively. Consider the propagation along the thread $t1$, for example. At node `2`, create access $R1 = (@x, r_1, true)$, add $R1$ to IS and update E to $\{R1\}$; At node `3`, update the path condition ψ to $(r_1 < 1)$, add access $R2 = (@x, r_2, (r_1 < 1))$ to IS, add $(R1, R2)$ to $\prec_S$, update E to $\{R2\}$, and update map $\mathcal{L}_1$ to $store(M_1, @tmp, r_2)$. At node `5`, add $W2 = (@x, V, (r_1 < 1))$ to IS where $V = (1 + select(\mathcal{L}_1, @tmp))$, i.e., $V = (1 + r_2)$, and so on. At the intra-thread join node `9`, the incoming states are merged to obtain $\psi = (r_1 < 1 \vee r_1 \geq 1) = true$ and $\mathcal{L}_1 = store(M_1, @tmp, ite(r_1 < 1, r_2, r_3))$ and $E = \{W2, W3\}$. At inter-thread join node `JOIN`, the incoming path conditions are conjuncted (trivially $true$) and E merged. Finally, the error condition is obtained from the path condition $(r_4 < 3)$ for the ERR node. The complete summary of the CCFG is given in Fig. 1 (c) and (d). Note that we do not propagate any global read or writes during CCFG exploration: all the global accesses are captured in IS. Although the data-flow analysis algorithm works on the complete CCFG, the analysis is thread-modular, i.e., each thread is analyzed independently using interference abstraction.

Function Summaries. The above algorithm can summarize arbitrary (bounded) concurrent programs assuming that functions are inlined. However, inlining causes blow up of the analyzed program and makes it difficult to exploit the modular sequential program structure. We can extend the above algorithm to perform a standard interprocedural analysis [31, 30] based on computing summaries at function boundaries and reusing these summaries at the calling contexts. A function summary consists of a interference skeleton

$$\text{INIT}\ \frac{n \in entry(CCFG)}{I_n(\langle true, L_0, \{\}\rangle)} \qquad \text{GUARD}\ \frac{\begin{array}{c} n \xrightarrow{g} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \\ \psi_g = eval(g, \mathcal{L}) \end{array}}{I'_n(\langle \psi \wedge \psi_g, \mathcal{L}, E\rangle)} \qquad \text{ASGN-LOC}\ \frac{\begin{array}{c} n \xrightarrow{lhs:=rhs} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \\ l = eval(lhs, \mathcal{L}) \quad v = eval(rhs, \mathcal{L}) \end{array}}{I'_n(\langle \psi, store(\mathcal{L}, l, v), E\rangle)}$$

$$\text{ASGN-GLB-R}\ \frac{\begin{array}{c} n \xrightarrow{lhs:=G[e]} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \quad IS = (S, \prec_S) \\ l = eval(lhs, \mathcal{L}) \quad l' = eval(e, \mathcal{L}) \quad R = (l', r, \psi) \quad r \textit{ is fresh} \end{array}}{I'_n(\langle \psi, store(\mathcal{L}, l, r), \{R\}\rangle) \quad IS = (S \cup \{R\}, \prec_S \cup \{(e, R) | e \in E\})}$$

$$\text{ASGN-GLB-W}\ \frac{\begin{array}{c} n \xrightarrow{G[e]:=rhs} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \quad IS = (S, \prec_S) \\ l = eval(e, \mathcal{L}) \quad r = eval(rhs, \mathcal{L}) \quad W = (l, r, \psi) \end{array}}{I'_n(\langle \psi, \mathcal{L}, \{W\}\rangle) \quad IS = (S \cup \{W\}, \prec_S \cup \{(e, W) | e \in E\})} \qquad \text{INTRA-JOIN}\ \frac{\begin{array}{c} m \to n \ \ m' \to n \ \ (tid(m) = tid(m') = tid(n)) \\ I_m(\langle \psi_1, \mathcal{L}_1, E_1\rangle) \ \ I'_m(\langle \psi_2, \mathcal{L}_2, E_2\rangle) \end{array}}{I_n(\langle \psi_1 \vee \psi_2, ite(\psi_1, \mathcal{L}_1, \mathcal{L}_2), E_1 \cup E_2\rangle)}$$

$$\text{FORK}\ \frac{\begin{array}{c} FORK(n) \quad n \to p \quad n \to c \\ tid(p) = tid(n) \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \end{array}}{I_p(\langle \psi, \mathcal{L}, E\rangle) \quad I_c(\langle \psi, \mathcal{L}_c, E\rangle) \quad \mathcal{L}_c \textit{ is fresh}} \qquad \text{INTER-JOIN}\ \frac{\begin{array}{c} JOIN(n) \quad p \to n \quad c \to n \quad tid(p) = tid(n) \\ I_p(\langle \psi_p, \mathcal{L}_p, E_p\rangle) \quad I_c(\langle \psi_c, \mathcal{L}_c, E_c\rangle) \end{array}}{I_n(\langle \psi_p \wedge \psi_c, \mathcal{L}_p, E_p \cup E_c\rangle)}$$

Figure 2: Transformation rules for thread-modular summarization of a CCFG fragment. For a node n, $I_n(\langle \psi, \mathcal{L}, E\rangle)$ denotes the incoming symbolic state at n; $tid(n)$ is the numeric identifier of the thread containing n; ite represents the *if-then-else* operator. The summary consists of the interference skeleton $IS = (S, \prec_S)$ and I_{ex} computed at exit node ex of the fragment.

(global accesses made in the function), together with the local symbolic state $\mathcal{L}$ at the exit node of the function. Here, the exit state $\mathcal{L}$ is computed using a fresh symbolic input state $\mathcal{L}_i$ at the function input. In contrast to explicit summarization approaches which depend on detecting transaction boundaries [28, 36], our method can compute symbolic summaries for arbitrary program regions across multiple transactions. The key problem is how to reuse precomputed summaries: given a calling context state $\mathcal{L}'$, the interference skeleton of the summary is duplicated and all global accesses evaluated in the incoming state $\mathcal{L}'$ by substituting $\mathcal{L}'$ for $\mathcal{L}_i$.

THEOREM 1. *The interference skeleton $IS = (S, \prec_S)$ is a precise thread-modular summary of the finite CCFG. Moreover, $\prec_S$ respects the program order* [1].

5. AXIOMATIC COMPOSITION

We now describe the second stage of our analysis which computes the inter-thread composition by using sequential consistency axioms that link the read and write accesses in the thread-modular summary IS correctly.

5.1 Linearization of the Global Skeleton

A *linearization* L of a interference skeleton $IS = (S, \prec_S)$ is a tuple $(S', <_{S'})$, where (i) $S' \subseteq S$, (ii) $<_{S'}$ is a total order, and (iii) for all $rw_1, rw_2 \in S'$, $rw_1 \prec_S rw_2 \Rightarrow rw_1 <_{S'} rw_2$. In other words, a linearization of IS is obtained by selecting a subset of accesses from IS and imposing a total order among them such that the total order respects the partial order in IS. A linearization L is said to be *program path-consistent* if its *projection* on to the CCFG corresponds to a single path for each program thread in the CCFG, and L should contain all the accesses in each path on which it is projected. Program path-consistency allows us to obtain concrete CCFG program paths from a linearization that leads to an error. A linearization L of IS is said to be *feasible* if there exists a concrete interleaved execution of the program CCFG corresponding to L. Note that a feasible linearization is always program path-consistent but not vice-versa.

Although each concrete execution corresponds to some linearization IS' of the skeleton IS, all linearizations may not be feasible program traces. Infeasible linearizations IS' occur, because the reads in IS' may not be linked to appropriate writes. In order to derive these constraints systematically, we define the *copy* relation.

[1]All proofs are available in the extended version of this paper.

Copy Relation. Let r and w be a read and write access in a read/write (total-ordered) sequence S. We say that r *copies* w, or *copy*(r, w) holds, if (a) r and w interfere, i.e., $(loc(r) = loc(w))$ (b) the value read by r is the same as the value written by w, $(val(r) = val(w))$ and, (c) there are no interfering write accesses w' to $loc(w)$ in S, such that $w <_S w'$ and $w' <_S r$. The main goal of composition, therefore, is to find a suitable write w for each read r so that r can copy w. The notion of sequential consistency (SC) [1] can be used to formally characterize this problem. A linearization $IS' = (S, <_S)$ is said to be *sequentially consistent* if the following axioms hold:

- **SC.1** (Program Order) Let rw_1 and rw_2 be read/write accesses to the same location l in the sequence S. If rw_2 follows rw_1 in the execution order of program P, i.e., $rw_1 \prec_P rw_2$, then $rw_1 <_S rw_2$.
- **SC.2** (Copy Some) Each read to location l in S must copy *some* write in S to l[2].

Axiomatic composition (AC) using the above SC axioms guarantees the feasibility of linearizations of IS.

THEOREM 2. *A program path-consistent linearization IS' of IS is feasible iff it is sequentially consistent.*

Axiomatic composition (AC) has been previously used to verify properties of concurrent data structures [2] executing on modern out-of-order processors. Here, AC was primarily used to precisely model various (intra-thread) read/write reorderings allowed by the processor. In contrast, we employ AC in an entirely new setting, i.e., static analysis of high-level programs. Here, the problem reduces to encoding only the SC constraints between reads/writes. However, the central challenge is to obtain an efficient encoding that enables a scalable analysis.

5.2 Copy Constraints

Note that by Theorem 1, any linearization of the interference skeleton IS must obey the program order **SC.1**. However, additional constraints must be imposed on a linearization to satisfy **SC.2**. We refer to such constraints as *copy constraints*, denoted by Φ_C. These constraints capture the *copy* relation and are modeled by a set of first-order logic formula quantified over reads and writes, consisting of Φ_C^1, Φ_C^2 and Φ_C^3.

[2]The initial value of location l is also represented by a write access with a fresh symbolic value.

$\Phi_C^1 : \forall r.\ occ(r) \Rightarrow \exists w.\ copy(r,w)$
$\Phi_C^2 : \forall r,w.\ copy(r,w) \Rightarrow occ(r) \wedge occ(w) \wedge$
$\quad (val(r) = val(w)) \wedge (loc(r) = loc(w)) \wedge HB(w,r)$
$\Phi_C^3 : \forall r,w.\ copy(r,w) \Rightarrow$
$\quad \forall (w' \neq w).\ (occ(w') \wedge HBet(w,w',r)) \Rightarrow loc(w) \neq loc(w')$

$\Phi_C = \Phi_C^1 \wedge \Phi_C^2 \wedge \Phi_C^3.$

The constraints Φ_C^1 capture the conditions **SC.2** (Sec. 5), i.e., each read (if it occurs) must copy some write access. The formula Φ_C^2 captures the data-flow constraints on the copy, i.e., the write w must occur ($occ(w)$), the values/locations of both r and w should be same and w must happen before r in the linearization. The predicate $HB(e_1, e_2)$ models a *strict* partial order relation that denotes that access e_1 *must happen before* e_2 in every linearization. In other words, if a linearization contains both e_1 and e_2 then e_1 must precede e_2. The formula Φ_C^3 captures the fact that no interfering write w' may happen between the write w and a read r that copies from w. The predicate $HBet(w, w', r)$ denotes that w' *may-happen-between* w and r, and is defined as $(\neg HB(w', w) \wedge \neg HB(r, w'))$. Details of encoding *HB* are presented in the next section. Φ_C^3 models that either w' does not happen between w and r, or does not interfere with w ($loc(w') \neq loc(w)$) if w' occurs in between.
Example. Recall the example and its IS in Fig. 1. A linearization L_0 = (*W1, R1, R3, R2, W2, R1', R2', W2', R4*) of the IS is not program path-consistent, since it does not project to a single path for thread $t1$. On removing *R3* from L_0, we obtain a linearization (say, L_1) which is path-consistent; however, L_1 is not feasible. To see this, note that because the reads $R1'$ and $R2'$ immediately follow the write $W2$ in the linearization, copy constraints Φ_C^1 and Φ_C^3 imply that both $copy(R1', W2)$ and $copy(R2', W2)$ must hold. Now, since $val(W2) = 1$, the constraints Φ_C^2 imply that $val(R1') = r_1' = val(R2') = r_2' = val(W2) = 1$. This, however, implies that $occ(R2') = (r_1' < 1) = false$, which violates Φ_C^2. Hence $R2'$ should not occur in the execution (and therefore in L_1). On replacing *R2', W2'* by *R3', W3'* in L_1, we obtain a feasible linearization (*W1, R1, R2, W2, R1', R3', W3', R4*).

5.3 Encoding the Composition

We encode the set of sequentially consistent linearizations of a interference skeleton IS as a formula in quantifier-free first-order logic: the skeleton $IS = (S, \prec_S)$ is encoded as a formula Φ_{IS}, and the copy constraints as Φ_C. The set of feasible linearizations of IS is then represented as a formula $\Phi = \Phi_{IS} \wedge \Phi_C$. Finally, given an error location with the path condition ψ, we can check if the error location is reachable via a feasible linearization by checking the satisfiability of the formula $\Phi \wedge \psi$ using an SMT solver.
Encoding Φ_{IS} and *HB*. Both Φ_{IS} and Φ_C depend on the strict partial order relation, *HB* between read/write accesses. To obtain an efficient encoding that avoids quantifiers, we encode the relation using the integer theory with the strict partial order operator $<$. More precisely, we assign an integer *clock* variable T_e to each access e. Now, $HB(e_1, e_2)$ is simply encoded as $T_{e_1} < T_{e_2}$. The accesses in IS are encoded in a straightforward manner: for each read/write access, we create three variables loc_e, val_e and occ_e and add constraints that equate each variable to the corresponding value. To model arbitrary initial values for locations in the map G lazily, we add a finite set of initial symbolic writes in IS as many as the number of reads in IS. Finally, we encode the partial order $\prec_S$ using the must-happen-before predicate, *HB*.
Encoding Φ_C. The quantified constraints in Sec. 5.2 can directly serve as input to an SMT solver that supports quantifiers, using interpreted functions for loc, val and occ. In practice, however, SMT solvers have difficulty in instantiating quantifiers efficiently. Therefore, we instantiate the copy constraints explicitly for all possible read and writes in IS using the corresponding loc, val and occ variables for each access. Modeling $copy(r, w)$ directly will introduce Boolean variables of form $copy\_r\_w$, quadratic in number of reads/writes, which we want to avoid. Therefore, we create an integer identifier variable ID_e for each access e, and assign a unique constant to ID_w for each write access w. Now, $copy(r, w)$ is encoded as $(ID_r = ID_w)$, which holds when the identifier to r is same as that for w. This encoding takes advantage of the fact that a read can only copy a single write.
Still, this explicit instantiation of Φ_C for all reads/writes is too eager and may result in a formula that is cubic in size of the read/write access set. The next section discusses optimizations to overcome the bottlenecks due to this eager encoding.

THEOREM 3. *Suppose we have an sequentially consistent encoding Φ for a CCFG C and a path condition ϕ for an error location e. If $(\Phi \wedge \phi)$ is satisfiable, then there exists a feasible interleaved execution of C to the location e.*

Example. Recall the program and its skeleton IS in Fig. 1. Checking the path condition $(r_4 \neq 3)$ for the ERR node together with Φ_{IS} and copy constraints Φ_C leads to a solution with a HB relation that refines $\prec_S$ (cf. Fig. 1) by adding pairs $(R1, R1')$, $(R1', R2)$ and $(W2, R2')$. The accesses $R3$, $W3$, $R3'$, $W3'$ do not occur, i.e., their occ evaluates to false, and they can be ignored. As a result, we obtain a linearization $(W1, R1, R1', R2, W2, R2', W2', R4)$ that witness the assertion violation.

6. INTERFERENCE PRUNING

Eager instantiation of the copy constraints Φ_C for all pairs of reads and writes in large programs proves to be a significant burden on the SMT solver during satisfiability check. Moreover, in case of indirect accesses, it is not clear upfront if a read r cannot interfere with a write w, i.e., $loc(r) = loc(w)$ is unsatisfiable, thus making the search more complex. However, many of these copy constraints may be *redundant*, i.e., $\neg copy(r, w)$ holds. For example, note that the constraints corresponding to Φ_C^2 may reduce to $\neg copy(r, w)$ if the right hand side (RHS) of the formula is unsatisfiable for some r and w. This may happen due to a number of reasons. For example, a read r cannot copy a write w that follows r in the program ($R2$, $W2$ in Fig. 1), or a read r in a child thread cannot copy a write w that occurs in the parent thread after the child thread terminates. Also, r can only copy from w if r and w may interfere, i.e., $loc(r) = loc(w)$ is satisfiable. In other words, each read may copy from only a restricted set of writes, and it is wasteful to add copy constraints for the writes not in the set. A large number of these redundant constraints can be detected statically by analyzing the interference skeleton IS and removed to optimize the composition (cf. Sec. 7). We now present a systematic method to *prune* these copy constraints, based on the following notions.
MHP and Kill Set. Given an interference skeleton $IS = (S, \prec_S)$, let $\prec_S^*$ denote the transitive closure of $\prec_S$. We say that two accesses e_1 and e_2 *may-happen-in-parallel*, i.e., $MHP(e_1, e_2)$ if both $e_1 \prec_S^* e_2$ and $e_2 \prec_S^* e_1$ do not hold. If a write w follows another write w' in the same thread, and w interferes with w' (i.e., $loc(w) = loc(w')$) in all program executions, then we say that w *kills* w'. More formally, the set of writes *killed* by w is given by $Kill(w) = \{w' \mid w' \prec_S^* w \wedge loc(w') \Rightarrow loc(w)\}$. Note that for symbolic values $loc(w)$ and $loc(w')$, $loc(w') \Rightarrow loc(w)$ must hold if w *kills* w' in all executions.
Reaching writes. We say that a write w may reach a read r, if (i) $MHP(r, w)$, or (ii) w happens before r and for all $w' \prec_S^* r$ $(w' \neq w)$, $w \notin Kill(w')$ holds. We denote the set of writes that may reach r by $\Pi(r)$. We compute $\Pi(r)$ for each read r as follows. First, we compute the transitive closure $\prec_S^*$ for the given skeleton IS. Computing (i) $MHP(r, w)$ requires checking if both (r, w)

or (w, r) do not belong to $\prec_S^*$. In order to compute the writes that are not killed, we perform a light-weight *Gen-Kill* [24] analysis of the partial order graph for $\prec_S$ (cf. Fig. 1). Starting from the node corresponding to the initial write, the analysis computes and propagates the set of reaching writes to each location in the graph (i.e., for each access in IS): each location that generates a write w may kill the incoming writes w' that belong to $Kill(w)$. Note that checking $loc(w') \Rightarrow loc(w)$ precisely is expensive if both $loc(w)$ and $loc(w')$ are symbolic values. So, we estimate the kill set conservatively by checking if $loc(w)$ and $loc(w')$ are exactly the same.

May-Copy Set. In order to prune the redundant constraints in Φ_C, we define the *may-copy* set $\mathcal{C}(r)$ for each r by restricting $\Pi(r)$ to *interfering* writes w that can occur, i.e., $\mathcal{C}(r) = \{w \mid w \in \Pi(r) \wedge (loc(r) = loc(w)) \wedge occ(w)\}$. Again, computing $\mathcal{C}(r)$ precisely is expensive: we syntactically check if $occ(w)$ or $(loc(r) = loc(w))$ is unsatisfiable. Finally, we instantiate Φ_C^1, Φ_C^2 and Φ_C^3 only for pairs r and w, where $w \in \mathcal{C}(r)$.

Instantiating the inner quantifier in Φ_C^3 for all possible writes w' may still produce redundant constraints. We prune such constraints by checking if (a) the write w' cannot occur $\neg(occ(w')$, or (b) cannot happen between w and r, $\neg(HBet(w, w', r))$, or (c) $loc(w') = loc(w)$ is unsatisfiable. We check (a) and (c) syntactically. To check (b), we use the $\prec_{IS}^*$ relation computed above, i.e., w' cannot happen between w and r if either $w' \prec_{IS}^* w$ or $r \prec_{IS}^* w'$. Our encoding can be further optimized to handle *atomic regions* and to perform a context-bounded [27, 21] analysis (see full version of the paper). Our experiments in the next section show that pruning redundant constraints as above leads to significant improvement of solver run times.

7. EXPERIMENTS

We implemented the staged analysis approach in the FUSION verification system [34] for concurrent C programs based on *PThreads*. The FUSION system combines dynamic and symbolic verification techniques in order to verify properties of concurrent programs. In the first step of its execution, FUSION instruments the given concurrent program and then runs the program P to obtain a slice of P. The slice is represented as a CCFG [34] and contains the threads created and the original program statements that each thread executed during the run. The slices themselves can grow quite large depending on the number of statements executed in P and the number of threads created, and are suitable for preliminary evaluation of our approach.

In order to evaluate our approach, our implementation focused on checking assertions as well as data races in the concurrent program slices obtained from benchmarks using FUSION. We first performed data flow analysis on the slices to obtain an interference skeleton and computed a pruned set of sequential consistency constraints over this skeleton. To check assertion violations at an error location, we add the feasibility path condition for the error location and employ the Yices SMT solver [9] to check if the set of constraints can be satisfied. To check data races between two locations, we add constraints modeling that the corresponding global accesses are simultaneously reachable, and again check for satisfiability using the solver. All experiments are conducted on a PC with 2.4Ghz Intel Core2Quad processor with 2GB memory limit running Fedora 10.

Our evaluation consists of two parts. First, we compared our staged analysis with a previous approach that uses concurrent single-static assignments (CSSA) (cf. [34]) for encoding concurrent programs. CSSA is an extension of single static assignments to concurrent programs to handle both intra-thread data flow (each variable must be assigned only once) as well interference (values of all concurrent writes are propagated to each read). The CSSA representation retains all the local control and data flow; moreover, a large number of fresh variables are introduced (in each thread) to model joins and interference. Due to the presence of complex local control and data flow, the approach performs repeated intra-thread reasoning for each interleaving of the program threads. Second, we evaluated the efficiency of optimizations when encoding composition using sequential consistency. More precisely, our goal was to estimate the impact of pruning redundant interferences.

We evaluated our approach using the following benchmarks obtained from the public domain. The first set of benchmarks consist of the C implementation of the $indexer$ example [11] using `Pthread` library, parametrized by the number of threads. In this example, multiple threads read and write to a hash table with 128 entries. As the number of threads increases beyond 12, the number of shared accesses also increases rapidly due to hash collisions. We check the property that no collision happens on a particular entry of the hash table. We evaluated the effectiveness of our approach for handling increasing number of threads as compared to the CSSA based implementation which does not perform summarization. We experimented with CCFGs with up to 32 threads to evaluate the scalability of our approach. The second set of benchmarks are obtained from traces of a bank account program (account) and a synchronization based module (SynchBench). Both these benchmarks were checked symbolically for existence of data races. The benchmarks are marked in the *name-(#T)* format where *#T* denotes the number of threads.

Fig. 3 shows the comparison of various modes of our tool with-/without summarization and optimized composition. The mode Old(+O) denotes an implementation of the symbolic checks based on CSSA encoding [34]. This implementation has been optimized extensively to reduce the number of redundant constraints (similar to Sec. 6), but does not use summarization. In the next mode (+S-O), we perform summarization but composition is done eagerly by instantiating copy constraints for all pairs of reads and writes. This eager instantiation leads to a large number of redundant constraints. Finally, (+S+O) denotes our approach with both summarization and optimized composition. We do not present results for the mode without summarization or optimization (-S-O) because of its poor performance.

Our experiments show that mode (+S+O) outperforms all other modes on our benchmarks. For example, as the number of threads in the $indexer$ example is increased successively from 20 to 31, both Old(+O) and (+S-O) modes scale much worse than (+S+O) mode. The mode (+S-O) without optimizations performs very poorly since the eager instantiation of sequential consistency (SC) constraints allows each global read to link with all global writes, e.g., in the $indexer(32)$ example, each read can copy from 1856 writes. The SMT solver is not able to handle such a large number of copy constraints effectively and timeouts in (+S-O) mode for 25 or more threads. This shows that a naive encoding of SC constraints is not useful for analyzing real-life benchmarks; an optimized encoding that avoids redundant constraints is needed. Similarly, the mode with optimized composition but without summarization Old(+O) timeouts for 31 and 32 threads. In contrast, the mode with both summarization and optimized composition (+S+O) finishes analyzing $indexer(32)$ in only 104s.

For the $account$ benchmark (similarly for the SynchBench example), we again observe that summarization (+S+O) leads to faster run times as the number of threads increase from 11 to 21. This supports the fact that performing repeated intra-thread reasoning when exploring large number of thread interleavings takes a significant toll on the overall efficiency of the solver. We observed that the average number of writes that a read may copy (cf. Sec. 6) after optimized composition in (+S+O) mode is $2 - 3$ for all the benchmarks (maximum varies between $10 - 20$). The results show that both summarization and optimized composition are indispensable for scaling up the analysis, and summarization can make verification tractable in cases where optimized composition is not sufficient. Moreover, note that the encoding used in context-bounded

Bm	$\|N\|$	$\|E\|$	$\|R\|$	$\|W\|$	Old(+O)	+S-O	+S+O
SynchBench(2)	108	107	6	19	1	1	1
SynchBench(13)	723	722	270	289	9	711	3
indexer(20)	1312	1439	110	291	0.1	355	0.1
indexer(27)	2142	2355	284	707	23	>1800	0.3
indexer(28)	2294	2523	322	797	97	>1800	4
indexer(29)	2446	2691	360	887	129	>1800	6
indexer(30)	2859	3149	468	1104	517	>1800	7
indexer(31)	3398	3747	594	1332	>1800	>1800	13
indexer(32)	4585	5065	888	1856	>1800	>1800	104
account1 (11)	906	905	134	372	1	121	1
account2 (21)	1748	1747	260	708	25	>1800	10

Figure 3: Experiments comparing (a) CSSA-based algorithm [34], without summarization, with optimizations (Old+O) (b) with summarization, no optimization (+S-O) (c) our method with summarization and optimization (+S+O). $|N|$ ($|E|$) = total number of nodes(edges) in the CCFG analyzed. $|R|$ ($|W|$) = total number of global reads (writes). All run-times are in seconds.

methods [21, 20] roughly corresponds to the (+S-O) mode where each global read may link with all possible shared variable writes. Therefore, the previous encoding is impractical for large programs.

8. RELATED WORK

Context-bounded Analysis. A number of approaches check concurrent software under a fixed context bound [27] since the verification problem is both decidable and practically useful [25]. Both symbolic [29, 21, 20] and explicit [25] approaches have been proposed for CBA. A recent approach [21] transforms a concurrent recursive Boolean program (with finite data) under a context bound to a sequential program, which is then analyzed using sequential analysis. [20] extend the method to perform context-bounded analysis of concurrent C programs by unrolling loops and recursion finitely to obtain decidability and employing a precise memory representation for C programs. In contrast, our goal is to verify real-life concurrent C programs where both (i) variable domains may be infinite and (ii) arbitrary context switches are allowed, (iii) without any redundant bi-modal reasoning. To achieve the condition (i) we need to structurally bound the loops and recursion in our programs, similar to [20]. Although [21] can be extended to achieve (ii) and avoid bi-modal reasoning, such an extension is impractical. More precisely, the extension will have to (a) duplicate and abstract all global state variables at *each* potential context switch location and (b) compute summaries for each pair of intra-thread locations [22], resulting in an extremely inefficient method. In contrast, our staged analysis abstracts interference by introducing fresh variables at global read locations only. Further, by structurally bounding the program (which is unavoidable due to decidability issues), we not only avoid redundant bi-modal reasoning, but in fact, separate the intra- and inter-thread reasoning completely.

Thread-modular summarization. Path compression techniques [36] rely on identifying interference-free *transactions* and are unable to summarize data facts across multiple transactions, causing redundant bi-modal reasoning. The Zing model checker employs function summarization [28] in presence of interference by identifying transactions during path-enumeration based *explicit-state* exploration. In contrast, our method performs precise summarization *symbolically* based on symbolic memory accesses as opposed to potentially infinite number of concrete-valued accesses. Further, we do not need to identify transactions and avoid path-enumeration by merging symbolic data facts at program *join* locations.

Thread-modular verification. Thread-modular or rely-guarantee techniques for software include the initial deductive methods [26, 18] followed by more recent methods that employ iterative compositional refinement [12, 17, 3, 6] or methods for handling heap-manipulating programs [15, 33]. Iterative refinement techniques first abstract the transition relation [12] or the reachable states [6] of the individual threads, by *over-approximating* the relation between global accesses in each thread. If the composition of these abstractions is not suitably precise to prove the given property, the methods refine the abstractions iteratively, e.g., based on counterexamples [17, 6] from the property check. Interference abstraction also makes our thread-modular summaries over-approximate; however, because our summaries contain the exact relation between non-concurrent reads and writes, we can obtain the fully precise system in one step by linking the reads to the writes during composition. This is in contrast to the previous methods where a large number of iterations may be required before the abstractions are made suitably precise for proving a property. Instead of performing rely-guarantee reasoning [18] as in [12, 17, 6], our focus is on compositional minimization [5, 16]: we use summaries to separate intra- and inter-thread analysis and obtain a compact representation of threads before composition.

Concurrent Data Flow Analysis. Most concurrent data flow analyses (cf. [10]) employ a finite-height data domain and perform redundant bi-modal reasoning by repeated intra-thread propagation of new symbolic domain values while exploring all relevant interleavings *explicitly*. In contrast, our staged analysis performs summarization over infinite domains using program expressions (terms), followed by *symbolic* exploration of interleavings inside an SMT solver [9, 8].

Other Symbolic Encodings. Another set of methods use SAT/SMT solvers to check concurrent programs with an encoding that does not employ an explicit scheduler [13, 34]. None of these approaches employ thread-modular summarization and therefore perform redundant bi-modal reasoning: the solver must reason over concurrent interleavings as well as complex local transitions alternately. The encoding presented in FSE 2009 [34] is based on transforming a (bounded) program into a concurrent single static assignments (CSSA) form and is restricted to handling simple integer programs, in contrast to ours, which handles arbitrary C programs and performs summarization. The idea of interference abstraction is employed implicitly in the CSSA representation, but not exploited to compute summaries.

Verification of concurrent data structures under relaxed low-level hardware memory models [2] employ axioms specifying the allowed load/store event orderings to the hardware memory, in order to precisely model the concurrent interleavings. Our method, instead targets high-level static analysis of C programs; for high-level analysis, it is sufficient to consider only sequentially consistent (SC) orderings of reads and writes, as opposed to more relaxed memory orderings considered in [2]. As a result of this restriction, our method focuses on encoding SC efficiently inside an SMT solver, which is not considered in [2].

9. CONCLUSIONS

We presented a staged analysis for verifying concurrent C programs which separates intra- and inter-thread reasoning and exploits sequential summarization to solve the pervasive problem of redundant bi-modal reasoning. The key contribution is a thread-modular program summarization algorithm which abstracts away all the local control and data flow in terms of global accesses. The summarized interference skeleton is then used for inter-thread analysis by employing sequential consistency axioms over the global accesses. Experimental results on benchmarks show our approach is more scalable than previous bi-modal methods because it avoids repeated intra-thread reasoning. Future work will focus on scaling our approach to larger concurrent systems: the key problem is to further minimize the set of quantifier instantiations, which are cubic in the size of reads and their *may-copy* sets, which causes blow up on larger benchmarks. Another approach is to avoid explicit instantiation by using quantified SC axioms inside the solver. Our method can also be extended to perform data flow analysis on less precise domains than terms, e.g., polyhedra.

Acknowledgements. We would like to thank the Verification group at NEC and the anonymous reviewers for their invaluable feedback.

10. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

[3] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC*, 2003.

[4] Shaunak Chatterjee, Shuvendu K. Lahiri, and Shaz Qadeer. A reachability predicate for analyzing low-level software. In *TACAS*. Springer, 2007.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, 2000.

[6] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 34(2):104–125, 2009.

[7] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314, 2009.

[8] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[9] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[10] A. Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *TACAS*, pages 102–116, 2007.

[11] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[12] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.

[13] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN*, pages 114–133, 2008.

[14] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[15] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.

[16] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *CAV '90*, pages 186–196, London, UK, 1991. Springer-Verlag.

[17] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[18] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[19] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[20] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV*, pages 509–524, 2009.

[21] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, pages 37–51, 2008.

[22] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, pages 282–298, 2008.

[23] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. M. Kaufmann, 1997.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[26] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[27] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[28] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL*, pages 245–255, 2004.

[29] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, pages 82–97, 2005.

[30] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, New York, NY, USA, 1995. ACM.

[31] M. Sharir and A. Pnueli. Two approaches to interprocedureal data flow analysis. In *Program Flow Analysis: Theory and Applications*, volume 5, pages 189–234. Prentice Hall, 1981.

[32] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[33] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[34] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ESEC/SIGSOFT FSE*, pages 23–32, 2009.

[35] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, pages 256–272, 2009.

[36] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Form. Methods Syst. Des.*, 25(1):67–96, 2004.

Summarization. Fig. 4 (next page) presents the complete set of rules for thread-modular summarization of a CCFG. For a node n, $I_n(\langle\psi, \mathcal{L}, E\rangle)$ denotes the incoming symbolic state at n; $tid(n)$ is the numeric identifier of the thread containing n; ite represents the *if-then-else* operator. On analyzing each function **f**, we obtain an interference skeleton IS for **f** having global accesses contained in **f**. Note that IS in the above rules denotes the skeleton for the function containing the CCFG node being analyzed.

Most rules are same as those in the Fig. 2 except the new rules: INIT-F, MK-SUMMARY and FUNC-CALL. At the entry location $entry(f)$ of each function **f**, the INIT rule initializes the local memory to $\mathcal{L}_0^f$ and the interference skeleton IS for **f** to an empty set. The rule MK-SUMMARY is used to construct the summary χ_f for a function **f** at its exit node $n = exit$(**f**) consisting of the skeleton IS computed for **f** and the state I_n.

The rule FUNC-CALL is used to evaluate and reuse a previously computed summary for **f** at a call node n using the evaluation operator Γ. Suppose we have a summary $\chi_f = (IS_f, I_f)$ for a function **f** where $IS_f = (S_f, \prec_{S_f})$ and $I_f = \langle\psi_f, \mathcal{L}_f, E_f\rangle$ such that both IS_f and I_f are represented in terms of the input local memory map L_f^0 to **f**. Given a partial order $\prec$, we use $\top(\prec)$ and $\bot(\prec)$ to denote the set of *maximal* (top) and *minimal* (bottom) elements in $\prec$. To evaluate χ_f under the calling context represented by the map $\mathcal{L}$, the function Γ substitutes L_f^0 by $\mathcal{L}$ in all components of χ_f. Further, all read placeholder values (due to interference abstraction) are substituted by fresh placeholders in all components of χ_f. After the above substitutions on χ_f, we obtain a skeleton $(S', \prec_{S'})$, path condition ψ' and memory map $\mathcal{L}'$. The data propagated to function return node n' in the FUNC-CALL rule is now given by

$$\langle\psi \wedge \psi', \mathcal{L}', \top(\prec_{S'})\rangle$$

where, the path condition is computed by conjoining the incoming path condition ψ with ψ', the new local map is $\mathcal{L}'$ and the set of last events contains the set of top events $\top(\prec_{S'})$ in S'. Moreover, the current interference skeleton IS is updated to add partial orders between all events in E and set $\bot(\prec_{S'})$ of bottom events in S'.

THEOREM 1. *The interference skeleton $IS = (S, \prec_S)$ is a precise thread-modular summary of the finite CCFG. Moreover, $\prec_S$ respects the program order.*

Proof. Follows by construction of the data flow analysis algorithm. All the transformers and joins are precise. The set E maintains the program order and is used to construct $\prec_S$.

THEOREM 2. *A program path-consistent linearization IS' of IS is feasible iff it is sequentially consistent.*

Proof.
($\Leftarrow$) A sequential consistent linearization L, by definition, always corresponds to a set of concrete execution traces. Let T be one such trace. Since L is program path-consistent, T can be obtained by interleaving paths from the individual threads in the CCFG. Hence L is feasible.
($\Rightarrow$) By definition, if a linearization IS' is feasible then it corresponds to a concrete interleaved execution of the CCFG. All concrete executions are sequentially consistent. Hence, IS' is sequentially consistent.

THEOREM 3. *Suppose we have an sequentially consistent encoding Φ for a CCFG C and a path condition ϕ for an error location l. If ($\Phi \wedge \phi$) is satisfiable, then there exists a feasible execution of C to the location l.*

Proof. Let $\Phi = \Phi_{IS} \wedge \Phi_C$. Let us associate a dummy global write access w with location l, so that $occ(w) = \phi$. If $\Phi \wedge \phi$ is satisfiable, then we get a partial order relation HB on the accesses in IS. Remove the global accesses that don't occur, i.e., $occ(e)$ is false, and those that happen after w from HB. Collapse the partial order HB into a linearization L arbitrarily. Since Φ_C is satisfied, so L is sequentially consistent. If L is program path consistent also, then by Theorem above, L is feasible, and there exists a feasible execution of C that ends at access w, i.e., location l. It remains to show that L is program path consistent. We show that L corresponds to exactly one path for each thread sub-graph in between a FORK-JOIN node pair. We can concatenate such paths to obtain a projection of L on the full CCFG C. Note that since path condition ϕ is satisfiable, and path conditions are conjuncted at JOIN nodes for encoding Φ_{IS}, L must correspond to *at least one* path for each thread in the sub-graph. It remains to show that L cannot contain accesses from multiple paths in a thread. Consider a basic conditional branch-join sub-graph in a thread and the outgoing paths from the branch node. Due to the encoding Φ_{IS}, either all accesses on these paths occur, or don't occur. So, exactly one outgoing path is feasible. By concatenating such feasible paths at each branch node, we obtain exactly one feasible path through the thread.

Context-bounded analysis. We can restrict the solver to search for executions limited to a fixed context bound by adapting the encoding in [35] to our setting. More precisely, we introduce a new predicate $Ctx(e)$ for each access e; We set $Ctx(w) = 0$ for all initial writes w. Given two accesses e_1 and e_2, we set $Ctx(e_1) \leq Ctx(e_2)$ if e_1 and e_2 belong to the same thread and $e_1 \prec_{IS} e_2$. Otherwise, we add constraints of the form

$$copy(r, w) \implies (Ctx(r) < Ctx(w))$$

for each read r that may-copy write w and $MHP(r, w)$ holds. These constraints ensure that if a read r copies a concurrent write w, then at least one context switch must happen in between. Let the global access e_f be such that $\forall e \in S.(e \prec_S e_f)$ (if e_f does not exist, we create a new dummy access). Finally, to restrict the search to a given bound k, we add the constraint $Ctx(e_f) \leq k$. The proof of correctness of this encoding is a straightforward extension of the proof in [35] and is omitted. Note that our approach does not duplicate the complete global state at each context switch location as in previous approaches [21], whose cost is proportional to product of the number of global variables and the number of thread locations. Instead, new placeholders are introduced lazily only at each global read location, making the cost linear in the number of global reads.

$$\text{INIT-F}\ \frac{n \in entry(\mathbf{f})}{I_n(\langle true, L_f^0, \{\}\rangle) \qquad IS = (\{\},\{\})} \qquad \text{GUARD}\ \frac{\begin{array}{c} n \xrightarrow{g} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \\ \psi_g = eval(g, \mathcal{L}) \end{array}}{I'_n(\langle \psi \wedge \psi_g, \mathcal{L}, E\rangle)} \qquad \text{ASGN-LOC}\ \frac{\begin{array}{c} n \xrightarrow{lhs:=rhs} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \\ l = eval(lhs, \mathcal{L}) \quad v = eval(rhs, \mathcal{L}) \end{array}}{I'_n(\langle \psi, store(\mathcal{L}, l, v), E\rangle)}$$

$$\text{ASGN-GLB-R}\ \frac{\begin{array}{c} n \xrightarrow{lhs:=G[e]} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \quad IS = (S, \prec_S) \\ l = eval(lhs, \mathcal{L}) \quad l' = eval(e, \mathcal{L}) \quad R = (l', r, \psi) \quad r \textit{ is fresh} \end{array}}{I'_n(\langle \psi, store(\mathcal{L}, l, r), \{R\}\rangle) \qquad IS = (S \cup \{R\}, \prec_S \cup \{(e, R) | e \in E\})}$$

$$\text{ASGN-GLB-W}\ \frac{\begin{array}{c} n \xrightarrow{G[e]:=rhs} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \quad IS = (S, \prec_S) \\ l = eval(e, \mathcal{L}) \quad r = eval(rhs, \mathcal{L}) \quad W = (l, r, \psi) \end{array}}{I'_n(\langle \psi, \mathcal{L}, \{W\}\rangle) \quad IS = (S \cup \{W\}, \prec_S \cup \{(e, W) | e \in E\})} \qquad \text{INTRA-JOIN}\ \frac{\begin{array}{c} m \rightarrow n \ \ m' \rightarrow n \ \ (tid(m) = tid(m') = tid(n)) \\ I_m(\langle \psi_1, \mathcal{L}_1, E_1\rangle) \ \ I'_m(\langle \psi_2, \mathcal{L}_2, E_2\rangle) \end{array}}{I_n(\langle \psi_1 \vee \psi_2, ite(\psi_1, \mathcal{L}_1, \mathcal{L}_2), E_1 \cup E_2\rangle)}$$

$$\text{FORK}\ \frac{\begin{array}{c} FORK(n) \quad n \rightarrow p \quad n \rightarrow c \\ tid(p) = tid(n) \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \end{array}}{I_p(\langle \psi, \mathcal{L}, E\rangle) \quad I_c(\langle \psi, \mathcal{L}_c, E\rangle) \quad \mathcal{L}_c \textit{ is fresh}} \qquad \text{INTER-JOIN}\ \frac{\begin{array}{c} JOIN(n) \quad p \rightarrow n \quad c \rightarrow n \quad tid(p) = tid(n) \\ I_p(\langle \psi_p, \mathcal{L}_p, E_p\rangle) \quad I_c(\langle \psi_c, \mathcal{L}_c, E_c\rangle) \end{array}}{I_n(\langle \psi_p \wedge \psi_c, \mathcal{L}_p, E_p \cup E_c\rangle)}$$

$$\text{MK-SUMMARY}\ \frac{n \xrightarrow{return} n' \quad n = exit(\mathbf{f}) \quad I_n(\langle \psi, \mathcal{L}, E\rangle)}{\chi_f = (IS, \langle \psi, \mathcal{L}, E\rangle)}$$

$$\text{FUNC-CALL}\ \frac{n \xrightarrow{call\ \mathbf{f}} n' \quad I_n(\langle \psi, \mathcal{L}, E\rangle) \quad IS = (S, \prec_S) \quad ((S', \prec_{S'}), \psi', \mathcal{L}') = \Gamma(\chi_f, \mathcal{L})}{I'_n(\langle \psi \wedge \psi', \mathcal{L}', \top(\prec_{S'})\rangle) \quad IS = (S \cup S', \prec_S \cup \{(e, e') | e \in E \wedge e' \in \bot(\prec_{S'})\})}$$

Figure 4: Complete set of summarization rules.