Verification of Parameterized Concurrent Programs By Modular Reasoning about Data and Control

Azadeh Farzan   Zachary Kincaid
University of Toronto [*]
azadeh,zkincaid@cs.toronto.edu

Abstract

In this paper, we consider the problem of verifying thread-state properties of multithreaded programs in which the number of active threads cannot be statically bounded. Our approach is based on decomposing the task into two modules, where one reasons about data and the other reasons about control. The data module computes thread-state invariants (e.g., linear constraints over global variables and local variables of one thread) using the thread interference information computed by the control module. The control module computes a representation of thread interference, as an incrementally constructed *data flow graph*, using the data invariants provided by the data module. These invariants are used to rule out patterns of thread interference that can not occur in a real program execution. The two modules are incorporated into a feedback loop, so that the abstractions of data and interference are iteratively coarsened as the algorithm progresses (that is, they become weaker) until a fixed point is reached. Our approach is sound and terminating, and applicable to programs with infinite state (e.g., unbounded integers) and unboundedly many threads. The verification method presented in this paper has been implemented into a tool, called DUET. We demonstrate the effectiveness of our technique by verifying properties of a selection of Linux device drivers using DUET, and also compare DUET with previous work on verification of parameterized Boolean program using the Boolean abstractions of these drivers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion Checkers, Correctness Proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants, Assertions; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.4.6 [*Operating Systems*]: Security and Protection—Verification

General Terms Verification, Algorithms, Reliability

Keywords Concurrency, Abstract Interpretation, Compositional Reasoning, Data Flow Graphs, Parameterized Programs, Thread Invariants

1. Introduction

Concurrent programs are notoriously hard to verify. Verification of concurrent systems has been a very active area of research in the past few years. There has been significant progress with testing and bug finding techniques, but due to a huge number of possible schedules (even with a fixed input), it is hard to provide useful coverage guarantees. Standard model checking techniques provide coverage, but suffer from the state explosion problem. Static analysis techniques [7, 14, 26, 27, 31, 32] have been successful in checking simple generic properties such as race and deadlock freedom.

There is a large class of concurrent programs that are designed to be executed by an arbitrary number of clients (for example, device drivers, concurrent data structure libraries, and file systems). This class of programs, commonly called *parameterized concurrent programs*, are more difficult to verify than concurrent programs with a fixed number of threads. The verification problem for parameterized systems has been studied extensively [2, 3, 6, 10, 12, 13, 16, 20, 21, 23, 29, 30]; however, the focus has mostly been on verifying *protocols*. These protocols (for example, Lamport's Bakery protocol [24] and Peterson's mutual exclusion algorithm) are often *small* programs, but the reasoning behind their correctness is usually complicated (see Section 7 for a more detailed discussion), specially when functional correctness is the goal (e.g. mutual exclusion). In contrast, we are interested in *large* programs such as device drivers and file systems, where the reasoning behind their correctness is more straightforward. In these programs, undesired inter-thread interference is usually prevented by simple synchronization mechanisms, and the majority of the verification effort is spent in reasoning about the sequential behaviour of each thread. Moreover, we are interested in proving *program assertions*, which is easier to handle than proving functional correctness of protocols. Program assertions are Boolean combinations of program expressions that relate shared variables to local variables of a specific thread at a location in that thread. They are expressive enough to include interesting properties of concurrent programs such as the absence of null pointer dereferences or out of bounds array accesses. We believe that this combination of programs and properties is a good candidate for automated verification in the parameterized setting, and is the target of the work presented in this paper.

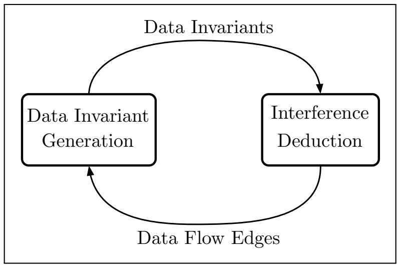
We propose a static analysis technique that separates the verification task into a data module and a control module. This separation achieves both *precision* and *scalability* for proving assertions correct in concurrent programs with unboundedly many threads. The data module, using an abstract interpreter, computes data invariants for each program location. These invariants are guaranteed to be consistent with the most recent information about inter-thread interference, provided by the control module. The control module, using a deduction system, determines the pattern of inter-thread interference by using the most recent information from the data module about data invariants. The two modules are combined into a

[*] Both authors were partially supported by an National Science and Engineering Research Council (NSERC) Discovery Grant for this work.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01. . . $10.00

feedback loop to collaboratively compute a solution to the verification problem. This strategy can be used to generate *thread-state invariants* for program locations, i.e. invariants that do not refer to the local variables of more than one thread. These invariants can be used for proving the absence of program assertion failures.

One of the enabling ideas of our modular approach is to use a data flow graph as a program representation for performing abstract interpretation. In data flow graphs, only the flow of data is modeled; control constructs (which are irrelevant to a data analysis) are abstracted away. Data flow graphs offer a convenient way to represent the interference between threads: for example, if one thread writes to a global variable at some location u which is subsequently read by another thread at location v, then u and v are connected by a data flow edge. Also, since each program location (regardless of the number of threads in the system) is represented by one vertex in the data flow graph (i.e. there is no explicit representation for threads), it is possible to use them to analyze data in parameterized systems. Computing data invariants over a data flow graph is mostly as straightforward as abstract interpretation for sequential programs; the idea is that the structure of a data flow graph captures the interference among threads, and therefore, one only needs to focus on how data flows through this structure.



The figure on the right illustrates the idea behind our approach. We start by assuming no interference among threads (as if threads are running sequentially), and through abstract interpretation compute the first set of *data invariants* (per program location). At this stage, the data flow graph only contains data flow edges that correspond to the sequential executions of program threads. Then, the data invariants are passed to a deduction system, which uses them to compute a new set of data flow edges. The deduction system uses the data invariants to reason about what patterns of inter-thread interference are feasible, and adds the corresponding data flow edges, which capture these new interference patterns, to the data flow graph. These new data flow edges may result in computing weaker data invariants in the next round. These weaker invariants may trigger the addition of more data flow edges (*coarsening* the data flow graph). This loop continues until a fixed point is reached.

The analysis we use for computing data flow edges is *semi-compositional*. Inter-thread data flow is a property that intuitively involves two threads: one that writes to a global variable, and one that reads from it. Ideally, one would like to reason about the existence of data flow edges by considering only two threads at a time. However, it is not generally sound to reason using only two threads, since a program path (e.g. from a write to a read) may involve many threads synchronizing with each other. We overcome this problem by using data invariants (from the data invariant generation module) to soundly approximate the effects of other threads that may contribute to a data flow path being realizable. An invariant associated with a program location corresponds to an overapproximation of the values of the program variables when at least one thread is at that location. Therefore, we may reason about data flow paths that may require the participation of arbitrarily many threads while considering only two threads at a time. We call our approach *semi-compositional* (in contrast to fully compositional) because our reasoning method over two threads is non-compositional (all pairs of locations are considered), while every other thread in the system is accounted for in a compositional manner.

We implemented our approach in a tool called Duet and evaluated it on a set of 15 Linux device drivers. Duet can prove a total of 1312 (out of 1597) of array bounds and integer over/underflow assertions safe in 13 minutes. We also compared Duet against two recent tools that deal with parameterized concurrent Boolean programs, namely Getafix [22], and the dynamic cutoff detection (DCD) algorithm of [19], which are both based on model checking. We compared against DCD [19] and Getafix [22] on the set of benchmarks provided by the authors, which are Boolean abstractions (generated by the SatAbs [9] tool) of the aforementioned Linux device drivers. Duet has the clear advantage of being directly applicable to the drivers (as opposed to their Boolean abstractions), but we performed these experiments to show that it does outperform these tools even at the level of Boolean programs. Our experiments shows that Duet substantially outperforms Getafix, by proving 2505 (compared to 1382 for Getafix) Boolean programs correct. Duet also outperforms DCD by proving 58 programs safe in contrast to only 19 programs that DCD can prove safe.

1.1 Motivating example

In concurrent programs, threads communicate using many different methods. They may communicate through synchronization primitives such as locks or wait/notify, which can be viewed as *control* type primitives that only affect the feasible control paths in a thread, and not the data. Threads may also communicate via reading from and writing to shared memory, which are *data* type primitives. In the presence of both patterns of communication among threads, precise reasoning about the program often involves reasoning about both *data* and *control* simultaneously. But, doing so can be very expensive, specially for a programs with an unbounded number of threads. In this paper, we present a scalable approach to reason about *data* and *control* in separate but collaborating modules. In this section, we use an example that includes both modes of communication among threads (through both data variables and locks) to provide a high level understanding of our approach.

Figure 1 illustrates a simplified concurrent producer-consumer example. Let us assume that each statement is executed atomically, and that initially no thread holds any locks, and `counter` is 0. The producer can produce items one-by-one or in batch mode, and keeps track of the number of produced items waiting to be consumed using the global variable `counter`. Two producers cannot produce items simultaneously, but a consumer can run in parallel with a producer in batch mode. The consumer, if there are items to process, consumes them one-by-one, by decrementing `counter`. The assertion at v_6 states a correctness property for the consumer: the number of items waiting to be consumed must be non-negative. At position u_{10} in the producer, the value of `counter` is always zero, so the assignment at u_{10} does not add any behaviour, but it helps us demonstrate an interesting point.

This program demonstrates the use of both synchronization primitives (locks in this case), and conditional statements to rule out undesired interference from other threads. For example, if the goal is to prove that the assertion at v_6 holds, then one must prove that the zero value assigned to `counter` at u_{10} cannot reach the decrement at v_5, and consequently falsify the assertion. In other words, we need to rule out a pattern of thread interference in order to prove the desired invariant at v_5 and v_6. The interesting aspect of this example is that the reverse is also true: since the locations u_{10} and v_5 are not protected by a common lock, in order to rule out the undesired interference, one needs to rely on the fact that at v_5, the value of `counter` is always strictly positive. In this scenario, if one starts from a weak invariant at v_5, one cannot rule out interference from u_{10}, and (reversely), if one starts by assuming the interference from u_{10} may occur, one cannot prove the data invariant needed to rule out this interference.

Here, we explain how our algorithm operates on the program in Figure 1 at a high level. In Section 5 (in Example 5.1), we will revisit this example and explain it in more detail. First, we assume that `Producer` and `Consumer` are executed sequentially with no interaction, and compute a sequential data flow graph $P_1^\sharp$. Next, a set of data invariants is generated from $P_1^\sharp$. This analysis determines that the `Consumer` threads cannot reach location v_5 and that in `Producer` threads, location u_4 is unreachable. Using the previously computed data invariants, the interference deduction unit computes new data flows, and adds the appropriate edges to $P_1^\sharp$ to get a data flow graph $P_2^\sharp$. For example, this analysis determines that the value of `counter` from u_{12} may reach the beginning of the `Producer` and `Consumer` threads, and so adds the edges $u_{12} \to v_4$ and $u_{12} \to u_3$. The interference deduction unit uses the fact that `counter=0` is an invariant at u_{10} to infer that the value of `counter` from u_{10} cannot flow to v_5 without passing through v_4, and therefore, does *not* add an edge from u_{10} to v_5. Data invariants are then generated for the data flow graph $P_2^\sharp$, which determines that u_4 and v_5 are now reachable, and that `counter` $\in [0, \infty)$ at u_1. The subsequent interference analysis computes new data flow edges, for example $u_4 \to u_1$, which is possible as the result of u_1's new (weaker) invariant. The invariants are still strong enough to prove that there is no data flow from u_{10} to v_5. These edges are added to $P_2^\sharp$ to get a new data flow graph $P_3^\sharp$. The data analysis runs on $P_3^\sharp$, but does not produce weaker invariants, and consequently, the subsequent interference analysis does not produce any new data flow edges. The algorithm then terminates, having computed a set of invariants that soundly approximate the dynamic behaviours of the program. These invariants are strong enough to prove that the assertion at v_6 never fails.

The rest of this paper is organized as follows. In Section 2, we define our program model. We define the data flow graphs in Section 3, and discuss how data invariants can be computed by abstract interpretation of data flow graphs; this constitutes the *data* portion of our analysis. We then discuss construction of the data flow graphs in Section 4 where we explain how data invariants are used to infer new data flow edges. The complete algorithm as an iterative framework is presented in Section 5. Section 6 presents our experiments, Section 7 discusses related work, and Section 8 concludes.

2. Notation and the program model

We define a program to be a 4-tuple $P = \langle H, GV, LV, \mathcal{L}[\![\cdot]\!]\rangle$, where $H = \langle Loc, CF\rangle$ is a finite control flow graph (CFG) whose vertices we call *locations*, *GV* is a finite set of global variables, *LV* is a finite set of local variables, and $\mathcal{L}[\![\cdot]\!]$ assigns to each control location a transition relation. For the rest of this section, we will formalize this program model and introduce notation to be used in the rest of the paper.

We will identify threads with natural numbers, where the behaviour of an individual thread is given by the control flow graph $H = \langle Loc, CF\rangle$ together with the semantic function $\mathcal{L}[\![\cdot]\!]$. The initial vertex of H, which is assumed to have no control flow predecessors, is denoted by $init_{loc}$. The behaviour of the program P is defined to be that of the infinite parallel composition of all threads $n \in \mathbb{N}$. Since we are interested in thread invariants, we can restrict ourselves to finite executions, in which only finitely many threads can participate. Thus, for proving thread invariants, our program model is equivalent to the parameterized model in which P is taken to be the finite parallel composition of all threads up to some $k \in \mathbb{N}$, where k is a parameter. We also note that we lose no generality in assuming that every thread $n \in \mathbb{N}$ executes the same code, since several code segments can be simply combined into one.

```
     int Producer(int batch)
     {
u1:    acquire(lock2)
u2:    acquire(lock1)
       if (*) {
u3:      assume(counter>0)
u4:      counter++
u5:      unlock(lock1)
u6:      unlock(lock2)
u7:      return 1
       } else {
u8:      assume(counter<=0)
u9:      unlock(lock1)
u10:     counter = 0
         while(*) {
u11:       assume(batch>0)
u12:       counter++
u13:       batch--
         }
u14:     assume(batch<=0)
u15:     unlock(lock2)
u16:     return batch
       }
     }
```

```
     void Consumer()
     {
v1:    lock(lock1)
       while(*) {
v2:      unlock(lock1)
v3:      lock(lock1)
       }
v4:    assume(counter>0)
v5:    counter--
v6:    assert(counter>=0)
v7:    unlock(lock1)
     }
```

Figure 1. Producer-Consumer Example.

We will assume that *GV*, *LV*, and $\mathbb{N} \times LV$ are pairwise disjoint (a pair $\langle n, x\rangle \in \mathbb{N} \times LV$ is conceptually thread n's copy of the local variable x) and define the set of variables $Var = GV \cup LV$. For simplicity, we will assume that all variables are of type integer.

A *global state* is a pair $s = \langle s_{env}, s_{loc}\rangle$ consisting of a global environment $s_{env} \in GEnv = (GV \cup \mathbb{N} \times LV) \to \mathbb{Z}$ and an assignment of a control location to each thread $s_{loc} : \mathbb{N} \to Loc$. A *thread state* is a mapping $e \in TEnv = Var \to \mathbb{Z}$ from variables to values. For a given thread $n \in \mathbb{N}$ and global state s, the *thread state of n in s*, which we denote by $s[n]$, and which is defined by

$$s[n](x) = \begin{cases} s_{env}(x) & \text{if } x \in GV \\ s_{env}(\langle n, x\rangle) & \text{otherwise} \end{cases}$$

The function $\mathcal{L}[\![\cdot]\!] : Loc \to TEnv \times TEnv$ associates a transition relation on thread states to every location. We call a pair $a = \langle n, \langle v, v'\rangle\rangle$ consisting of a thread $n \in \mathbb{N}$ and a control flow edge $\langle v, v'\rangle \in CF$ an *action*. If the target v' of the control flow edge $\langle v, v'\rangle$ is understood from the context or irrelevant, we simply write $\langle n, v\rangle$. We use $\mathcal{A}[\![a]\!]$ to denote the (global state) transition relation associated with the action a, which is obtained by lifting $\mathcal{L}[\![v]\!]$ from thread states to global states as follows: $\langle s, s'\rangle \in \mathcal{A}[\![\langle n, \langle v, v'\rangle\rangle]\!] \iff \exists \langle e, e'\rangle \in \mathcal{L}[\![v]\!]$ such that

- $s[n] = e$ and $s'[n] = e'$
- $s_{loc}(n) = v$ and $s'_{loc}(n) = v'$
- $\forall n' \in \mathbb{N} \setminus \{n\}.\ s_{loc}(n') = s'_{loc}(n')$
- $\forall n' \in \mathbb{N} \setminus \{n\}, \forall x \in LV.\ s_{env}(\langle n', x\rangle) = s'_{env}(\langle n', x\rangle)$.

Given a sequence of actions ρ and a subset $N \subseteq \mathbb{N}$, the *projection* of ρ onto N, denoted $\rho|_N$, is the subsequence of ρ consisting of all actions whose thread identifiers belong to N. A *trace* σ is a finite sequence of actions such that, when projected onto a single thread, corresponds to a path in the CFG H beginning at the initial location $init_{loc}$. For a trace σ, the *post states of* σ (denoted $\sigma^\bullet$) is the set of final states of that execution. A trace τ is *feasible* if $\tau^\bullet$ is nonempty. For a trace σ and a thread $n \in \mathbb{N}$, we use $endloc(\sigma, n)$ to denote the end location of the CFG path

$\sigma|_{\{n\}}$. Note that *endloc* has the property that $\forall s \in \sigma^\bullet$, $\forall n \in \mathbb{N}$, $s_{loc}(n) = \mathit{endloc}(\sigma, n)$.

A thread-state property is an assertion φ with free variables in *Var*. We will denote the set of such formulae by $\mathcal{F}(\mathit{Var})$ (and more generally, $\mathcal{F}(V)$ will denote first-order formulae with free variables in V). For a thread state e, we write $e \models \varphi$ to denote that e satisfies φ.

To provide some intuition on our program model, we will describe how to represent locking. A lock is represented by a global variable $\mathit{lock} \in \mathit{GV}$. Let *acq* be a location where *lock* is to be acquired, and let *rel* be a location where it is to be released. Then we can define

$$\mathcal{L}[\![\mathit{acq}]\!] = \{\langle e, e[\mathit{lock} \leftarrow 1]\rangle : e(\mathit{lock}) = 0\}$$

$$\mathcal{L}[\![\mathit{rel}]\!] = \{\langle e, e[\mathit{lock} \leftarrow 0]\rangle : e(\mathit{lock}) = 1\}$$

Note that, given a feasible trace τ, the fact that $\tau^\bullet$ is nonempty implies that there is no point along τ in which two threads hold the same lock.

Our program model does not support conditional branching, but this can be simulated with nondeterministic branching and `assume` actions, where $\mathcal{L}[\![\texttt{assume}(c)]\!] = \{\langle e, e\rangle : e \models c\}$. Programs that depend on the initial state satisfying some property φ can be simulated by defining $\mathcal{L}[\![\mathit{init}_{loc}]\!]$ as $\mathcal{L}[\![\texttt{assume}(\varphi)]\!]$. Although our algorithm (and our implementation) handles dynamic thread creation, we omit it from this presentation for the sake of simplicity.

3. Data flow graphs

Data flow graphs (DFGs) are a program representation that explicitly represents the flow of data in a program, rather than the flow of control as in a control flow graph. Our analysis uses DFGs to compute program invariants by interpreting each edge of the DFG as a constraint and then computing an overapproximation of the least solution to this constraint system via abstract interpretation.[1]

A DFG for a program P is a directed graph $P^\sharp = \langle \mathit{Loc}, \mathit{DF}\rangle$, where $\mathit{DF} \subseteq \mathit{Loc} \times \mathit{Var} \times \mathit{Loc}$ is a set of directed edges labeled by program variables, and where we assume that *Loc* contains an additional location `uninit` (with no incoming or outgoing edges in the control flow graph of P). We will use $u \to^x v$ to denote the triple $\langle u, x, v\rangle \in \mathit{Loc} \times \mathit{Var} \times \mathit{Loc}$.

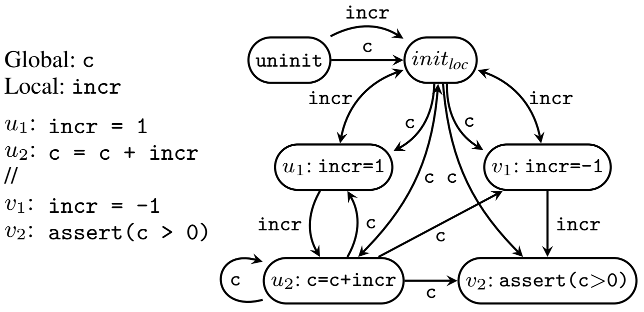
We define the collecting semantics of a DFG $P^\sharp = \langle \mathit{Loc}, \mathit{DF}\rangle$ to be the least solution to the following set of equations:

$$\mathrm{VAL}(u, x) = \{e(x) : e \in \mathrm{OUT}(u)\}$$

$$\mathrm{IN}(v) = \bigcap_{x \in \mathit{Var}} \{e \in \mathit{TEnv} : \exists u \to^x v \in \mathit{DF}. e(x) \in \mathrm{VAL}(u, x)\}$$

$$\mathrm{OUT}(v) = \begin{cases} \mathit{TEnv} & \text{if } v = \texttt{uninit} \\ \{e' : \exists e \in \mathrm{IN}(v). \langle e, e'\rangle \in \mathcal{L}[\![v]\!]\} & \text{otherwise} \end{cases}$$

Consider the example in Figure 2. This figure depicts a program with two code segments, each of which may be executed by arbitrarily many threads. Variable `c` is global, and variable `incr` is local. Each vertex (except the special vertex `uninit`) has at least one incoming edge for each variable. Each of these incoming edges provides a value for a particular variable. For example, the edge $u_2 \to^{\texttt{c}} v_2$ represents the constraint that any value for `c` *after* executing u_2 is a possible value for `c` *before* executing v_2 (that is, $\{e(\texttt{c}) : e \in \mathrm{IN}(v_2)\} \subseteq \{e(\texttt{c}) : e \in \mathrm{OUT}(u_2)\}$). The two edges from `uninit` to init_{loc} indicate that any value is possible for `incr` and `c` at init_{loc}, the location at which both threads begin execution. The vertex init_{loc} sets the initial condition of the program, acting as $\texttt{assume}(\texttt{c} = 0 \land \texttt{incr} = 0)$. Thus, edges originating at init_{loc} indicate that a particular variable may get its value from the initial state. The fact that $v_1 \to^{\texttt{incr}} u_2$ does *not* belong to this graph indicates that the value of `incr` at v_1 is not observable at u_2.



```
Global: c
Local: incr
u1: incr = 1
u2: c = c + incr
//
v1: incr = -1
v2: assert(c > 0)
```

Figure 2. A program and a data flow graph representing it.

Intuitively, a DFG for a program P represents a trace σ if it has "enough" edges to ensure that any thread state reached by σ belongs to $\mathrm{IN}(v)$ for some v. The remainder of this section formalizes this notion.

We will assume the existence of a function $\mathit{mod} : \mathit{Loc} \to \mathcal{P}(\mathit{Var})$ that maps every control location to the set of variables modified at that location. We require that *mod* satisfies the following: for any $x \in \mathit{Var}$, if there exists some $\langle e, e'\rangle \in \mathcal{L}[\![v]\!]$ with $e(x) \neq e'(x)$, then $x \in \mathit{mod}(v)$. A notable feature of the *mod* we use in practice is that a location of the form $\texttt{assume}(\varphi)$ is considered to be a modification of every variable occurring in φ. This allows us to take advantage of information at conditional branches that otherwise would not be possible in a data flow graph.

For a trace σ, variable x, and thread n, we define $\mathit{latest}(\sigma, x, n)$ to be the location of the last action to write to the variable x along σ (or thread n's copy of x, if x is local). More formally, if x is a global variable, $\mathit{latest}(\sigma, x, n)$ is the unique location v such that $\sigma = \pi\langle m, v\rangle\rho$ (for some $m \in \mathbb{N}$) and where no action (of any thread) along ρ modifies x if such a v exists, and `uninit` otherwise. Similarly, if x is a local variable, $\mathit{latest}(\sigma, x, n)$ is the unique location v such that $\sigma = \pi\langle n, v\rangle\rho$ and no action of thread n along ρ modifies x if such a v exists, and `uninit` otherwise.

DEFINITION 3.1 (Witness). *Let u, v be locations, x be a variable, and σ be a trace. We say that σ is a* witness *for the data flow edge $u \to^x v$ if there exists some thread $n \in \mathbb{N}$ such that $\mathit{latest}(\sigma, x, n) = u$ and $\mathit{endloc}(\sigma, n) = v$.*

Conceptually, σ is a witness for $u \to^x v$ if, on σ, u sets a value for x which is not changed until the end of the trace, where some thread is at v. We are now ready to define a representation condition for traces and program.

DEFINITION 3.2 (Representation). *A DFG $P^\sharp = \langle \mathit{Loc}, \mathit{DF}\rangle$* represents *a trace σ iff for every $u, v \in \mathit{Loc}$ and $x \in \mathit{Var}$, if some subtrace (not necessarily proper) σ' of σ is a witness for $u \to^x v$, then $u \to^x v \in \mathit{DF}$. $P^\sharp$* represents *a program P if it represents τ for every feasible trace τ of P.*

The relationship between the collecting semantics of a DFG and the traces it represents is given by the following:

THEOREM 3.3 (DFG Soundness). *Let σ be a trace and let $P^\sharp$ be a DFG such that $P^\sharp$ represents σ. Then the collecting semantics of $P^\sharp$ overapproximates the set of thread states reached by σ. Formally, for all $s \in \sigma^\bullet$, for all $n \in \mathbb{N}$, we have that $s[n] \in$ IN$(s_{loc}(n))$.*

[1] For a similar use of data-flow graphs, see for example [17].

Proof sketch Let $P^\sharp = \langle \mathit{Loc}, \mathit{DF} \rangle$ be a DFG and let σ be a trace represented by $P^\sharp$. We proceed by induction on σ.

Base case: follows from the fact that for all x, $\texttt{uninit} \rightarrow^x \mathit{init}_{\mathit{loc}} \in \mathit{DF}$ (since $P^\sharp$ represents ϵ) and the fact that for all x, $\mathrm{VAL}(\texttt{uninit}, x) = \mathbb{Z}$.

Inductive step: suppose σ is a trace such that $P^\sharp$ represents σ, let $n \in \mathbb{N}$, and let $v = \mathit{endloc}(\sigma, n)$. We need to show that $s[n] \in \mathrm{IN}(s_{\mathit{loc}}(n))$. To prove this, it is sufficient to show that $\forall x \in \mathit{Var}$, $\exists u \in \mathit{Loc}$ such that $u \rightarrow^x v \in \mathit{DF}$ and $s[n](x) \in \mathrm{VAL}(u, x)$.

Let x be a variable, and take u to be $\mathit{latest}(\sigma, x, n)$. Then $u \rightarrow^x v \in \mathit{DF}$ because σ witnesses this data flow and $P^\sharp$ represents σ. If $u = \texttt{uninit}$, we are done since $\mathrm{VAL}(\texttt{uninit}, x) = \mathbb{Z}$. If $u \neq \texttt{uninit}$, it follows that $\sigma = \pi\langle m, u\rangle\rho$ for some thread $m \in \mathbb{N}$, trace π, and sequence of actions ρ such that x is not modified along ρ (by the definition of $\mathit{latest}(\sigma, x, n)$). Moreover, we must have that there is some $s' \in \pi^\bullet$ such that $\exists s''$ with $\langle s', s''\rangle \in \mathcal{A}[\![\langle m, u\rangle]\!]$ and $s''[n](x) = s[n](x)$. By the induction hypothesis, $s'[m] \in \mathrm{IN}(u)$, so $s''[m] \in \mathrm{OUT}(u)$ and $s[n](x) = s''[n](x) = s''[m](x) \in \mathrm{VAL}(u, x)$. □

We also note that, unlike typical definitions of data flow graphs, we require that each location has inputs for every variable, rather than just the variables read by that location. This is a technical convenience that simplifies the presentation of our algorithm.

3.1 Abstract interpretation of data flow graphs

In this section, we discuss how invariants are computed over a data flow graph. For clarity of this presentation, we will assume a concrete representation of an abstract domain as a subset of $\mathcal{F}(\mathit{Var})$. The semantics of program locations is given by an abstract transition relation $\mathcal{L}[\![\cdot]\!]^\sharp : \mathit{Loc} \rightarrow \mathcal{F}(\mathit{Var}) \rightarrow \mathcal{F}(\mathit{Var})$ that overapproximates the strongest postcondition (i.e., $\langle e, e'\rangle \in \mathcal{L}[\![v]\!]$ implies $e' \models \mathcal{L}[\![v]\!]^\sharp(e)$). An *annotation* for a DFG $P^\sharp$ is a map $\iota : \mathit{Loc} \rightarrow \mathcal{F}(\mathit{Var})$ that assigns each location $v \in \mathit{Loc}$ a thread-state formula $\iota(v)$. We define an inductiveness condition for annotations that follows the structure of the collecting semantics, and which holds when the annotation overapproximates the collecting semantics of the DFG.

DEFINITION 3.4. *An annotation ι is* inductive *for a data flow graph $\langle \mathit{Loc}, \mathit{DF}\rangle$ if:*

- $\iota(\texttt{uninit}) = \mathit{true}$
- *For all $v \in \mathit{Loc}$,*

$$\Big[\bigwedge_{x \in \mathit{Var}} \Big(\bigvee_{u \rightarrow^x v \in \mathit{DF}} \overline{\mathcal{L}[\![u]\!]^\sharp(\iota(u))}^x \Big) \Big] \Rightarrow \iota(v)$$

where $\overline{\mathcal{L}[\![u]\!]^\sharp(\iota(u))}^x$ denotes the formula obtained from the formula $\mathcal{L}[\![u]\!]^\sharp(\iota(u))$ by existentially quantifying every variable except x.

Standard techniques can be used to compute inductive annotations from a DFG (inductive annotations correspond to post-fixpoint solutions in the terminology of abstract interpretation). For example, in our implementation, we use a variation of the well-known worklist algorithm. The following is a consequence of Theorem 3.3 and the fact that inductive annotations overapproximate the collecting semantics:

COROLLARY 3.5. *Let σ be a trace, let $P^\sharp$ be a dataflow graph that represents σ (Definition 3.2), and let ι be an inductive annotation for $P^\sharp$ (Definition 3.4). Then for all states $s \in \sigma^\bullet$, and all threads $n \in \mathbb{N}$, $s[n] \models \iota(s_{\mathit{loc}}(n))$ (i.e., the thread state of thread n in the global state s is overapproximated by the annotation at the location of thread n).*

Note that in the collecting semantics, and therefore in inductive annotations, the values of different variables cannot be correlated. For example, if v is a location, x and y are variables, and $e, e' \in \mathrm{IN}(v)$ are reachable thread states such that $e(x) = e(y) = 0$ and $e'(x) = e'(y) = 1$, then there exists an $e'' \in \mathrm{IN}(v)$ in which $e''(x) = 0 \neq 1 = e''(y)$. This suggests that DFGs are most appropriate for analyses based on *non-relational* abstract domains, such as intervals, signs, or the even/odd domain, which are also incapable of representing relationships between variables. In Section 5.1, we will discuss a variation of DFGs which are more appropriate for relational abstract domains.

4. Interference analysis

We now address the problem of how to compute a DFG that represents a program. We start by defining a subset of traces, called ι-feasible traces (where ι is a given annotation), and then develop an interference analysis that computes the set of data flow edges that are witnessed by ι-feasible traces. The definition of ι-feasibility is such that if ι is a "weak enough" annotation, then every feasible trace is ι-feasible. With such an annotation ι, every edge which is witnessed by a feasible trace will also be witnessed by an ι-feasible trace, and thus will be found by our interference analysis.

Our interference analysis relies on a finite domain of data invariants, which is defined using finite set of observable conditions.[2] An *observable condition* is a predicate c with free variables in GV. In the remainder of this section, we assume a fixed finite set of observable conditions, which we denote by $\mathcal{C}$. We define the set of observable formulae $\mathcal{F}^\sharp(\mathit{GV}) \subseteq \mathcal{F}(\mathit{GV})$ to be the set of formulae φ that can be expressed as a conjunction $\bigwedge_i \varphi_i$, where for each i, $\varphi_i \in \mathcal{C}$ or $\neg\varphi_i \in \mathcal{C}$.

An annotation $\iota : \mathit{Loc} \rightarrow \mathcal{F}(\mathit{Var})$ (along with the set of observable conditions $\mathcal{C}$) determines an *abstract annotation* $\iota^\sharp : \mathit{Loc} \rightarrow \mathcal{F}^\sharp(\mathit{GV})$ that assigns to each location $u \in \mathit{Loc}$ an observable formula φ that is implied by $\iota(u)$ and which is at least as strong as any other observable formula with this property. Thus, going from concrete to abstract (and using $\subseteq$ to denote "is more precise than"), we have $\mathrm{IN}(v) \subseteq \iota(v) \subseteq \iota^\sharp(v)$ for any location v.

The set of observable conditions $\mathcal{C}$ determines an *enabling condition enabled* $: \mathit{Loc} \rightarrow \mathcal{F}^\sharp(\mathit{GV})$ where $\exists e'.\langle e, e'\rangle \in \mathcal{A}[\![v]\!] \Rightarrow \langle e, v\rangle \models \mathit{enabled}(v)$ and $\mathit{enabled}(v)$ is at least as strong as any other observable formula with this property.

We are now ready to state our definition of ι-feasibility:

DEFINITION 4.1. *Let σ be a trace and ι be an annotation. Then σ is ι-*feasible *if:*

- *$\sigma = \epsilon$, or*
- *$\sigma = \sigma'\langle n, v\rangle$, where σ' is an ι-feasible trace, and for all $m \in \mathbb{N}$, $\iota^\sharp(\mathit{endloc}(\sigma', m)) \wedge \mathit{enabled}(v)$ is satisfiable.*

Note that the condition for extending an ι-feasible path σ by an action $\langle n, v\rangle$ depends only on the annotations at the end locations of each thread, rather than on the states in $\sigma^\bullet$.

EXAMPLE 4.2. *Consider the trace $\sigma = \langle 0, u_1\rangle\langle 0, u_2\rangle\langle 0, u_3\rangle$ of the program in Figure 1. This trace is not feasible, because every state in $(\langle 0, u_1\rangle\langle 0, u_2\rangle)^\bullet$ has* `counter = ` *0, so $\langle 0, u_3\rangle$ is not enabled. However, assuming that $\iota(u_2) = \texttt{counter} \geq 0 \wedge \texttt{lock1} = 1$, $\iota(u_1) = \texttt{counter} \geq 0$, and $\mathcal{C} = \{\texttt{counter} > 0, \texttt{lock1} = 0, \texttt{lock2} = 0\}$ (which implies $\iota^\sharp(u_1) = \mathit{true}$, $\iota^\sharp(u_1) = \neg(\texttt{lock1} = 0)$, and $\mathit{enabled}(u_3) = \texttt{counter} > 0$), this trace is ι-feasible. To illustrate how ι-feasibility depends on ι, consider the infeasible trace $\langle 0, u_1\rangle\langle 0, u_2\rangle\langle 1, v_1\rangle$. This trace is*

[2] This finiteness condition is not strictly necessary, but makes for a more efficient analysis.

INIT-COREACH

$$\frac{}{coreachable(init_{loc}, init_{loc})}$$

COREACH-SYM

$$\frac{coreachable(u, v)}{coreachable(v, u)}$$

COREACH-STEP

$$\frac{coreachable(u_0, v) \qquad enabled(u_0, \iota^\sharp(v)) \qquad \langle u_0, u_1 \rangle \in \mathit{CF}}{coreachable(u_1, v)}$$

MAYREACH-BASE

$$\frac{coreachable(u_0, v) \qquad x \in mod(u_0) \qquad enabled(u_0, \iota^\sharp(v)) \qquad \langle u_0, u_1 \rangle \in \mathit{CF}}{mayReach(u_0, x, u_1, v)}$$

MAYREACH-STEPL

$$\frac{mayReach(u_0, x, u_1, v) \qquad x \notin mod(u_1) \qquad enabled(u_1, \iota^\sharp(v)) \qquad \langle u_1, u_2 \rangle \in \mathit{CF}}{mayReach(u_0, x, u_2, v)}$$

MAYREACH-STEPR

$$\frac{mayReach(u_0, x, u_1, v_0) \qquad x \notin mod(v_0) \qquad enabled(v_0, \iota^\sharp(u_1)) \qquad \langle v_0, v_1 \rangle \in \mathit{CF}}{mayReach(u_0, x, u_1, v_1)}$$

MAYREACH

$$\frac{mayReach(u_0, x, u_1, v)}{u_0 \rightsquigarrow^x v}$$

Figure 3. Interference analysis.

ι-feasible if $\iota(u_3) =$ `counter` $\geq 0 \land$ `lock1` $= 1$, and ι-infeasible if $\iota(u_3) = \iota(u_8) =$ `counter` $\geq 0 \land$ `lock1` $= 1 \land$ `lock2` $= 1$.

By combining the definition of ι-feasibility with the definition of a witness of a data flow edge, we arrive at the following:

DEFINITION 4.3. *Let σ be a trace, ι be an annotation, u and v be locations, and x be a variable. Then σ is an ι-feasible witness of the data flow $u \rightarrow^x v$ if σ is ι-feasible and witnesses the data flow $u \rightarrow^x v$ (that is, σ simultaneously satisfies definitions 3.1 and 4.1).*

A key property of our notion of ι-feasibility is that it is preserved under projections; that is, for any ι, if σ is an ι-feasible trace, then for any sets of threads $N \subseteq \mathbb{N}$, $\sigma|_N$ is also ι-feasible. The following lemma states a projection result that forms the basis of our semi-compositional algorithm for interference analysis. It implies that, in order to compute the set of data flow edges that are witnessed by ι-feasible traces, it is sufficient to consider only traces that involve two threads.

LEMMA 4.4 (Projection). *Let ι be an annotation, u, v be locations, and x be a variable. Let σ be an ι-feasible witness for the data flow $u \rightarrow^x v$. Then there exists $m, n \in \mathbb{N}$ such that $\sigma|_{\{m,n\}}$ is an ι-feasible witness for $u \rightarrow^x v$.*

Proof sketch We will first prove that for any $N \subseteq \mathbb{N}$ and any ι-feasible trace σ, $\sigma|_N$ is an ι-feasible trace, by induction on σ.

The base case is obvious. For the inductive step, let $\sigma\langle n, v\rangle$ be an ι-feasible trace, and assume that $\sigma|_N$ is ι-feasible. If $n \notin N$, then $\sigma\langle n, v\rangle|_N = \sigma|_N$, and the result is immediate from the induction hypothesis.

If $n \in N$, then $\sigma\langle n, v\rangle|_N = \sigma|_N\langle n, v\rangle$. In this case, we need to show that for all $m \in \mathbb{N}$, $\iota^\sharp(endloc(\sigma|_N, m)) \land enabled(v)$ is satisfiable. Let $m \in \mathbb{N}$ and distinguish two cases:

- $m \in N$: then $endloc(\sigma|_N, m) = endloc(\sigma, m)$, and the fact that $\iota^\sharp(endloc(\sigma|_N, m)) \land enabled(v)$ is satisfiable follows from the fact that $\sigma\langle n, v\rangle$ is ι-feasible.
- $m \notin N$: then $endloc(\sigma|_N, n) = init_{loc}$. Since σ is finite, only finitely many threads execute actions in σ, so there exists a thread $i \in \mathbb{N}$ that does not execute actions in σ. Since $\sigma\langle n, v\rangle$ is ι-feasible, $\iota^\sharp(endloc(\sigma, i)) \land enabled(v)$ is satisfiable. Since $endloc(\sigma, i) = init_{loc} = endloc(\sigma|_N, m)$, we are done.

Now, we must prove that the property of being a witness is preserved by projections. Let u, v be locations and x be a variable, and let σ by a witness of the data flow $u \rightarrow^x v$. We assume that x is a global variable – the case of local variables is similar. It follows from Definition 3.1 that there exists some $m, n \in \mathbb{N}$, a trace π, and a sequence of actions ρ such that $\sigma = \pi\langle n, u\rangle\rho$ such that $x \in mod(u)$, x is not modified by any action along ρ, and $v = endloc(\sigma, m)$. It is easy to check that $\sigma|_{\{m,n\}}$ witnesses $u \rightarrow^x v$. □

4.1 Inferring data flow edges

Our algorithm for inferring data flow edges is stated declaratively in Figure 3 as a set of deduction rules for ι-feasible witnesses. For $u, v \in Loc$ and $x \in Var$, we write $u \rightsquigarrow_\iota^x v$ iff an ι-feasible witness for the data flow $u \rightarrow^x v$ exists. These proof rules are sound and complete for determining whether a witness for an inter-thread data flow edge exists – intra-thread data flows can be computed independently using a standard sequential reaching definitions analysis.

The rules use an input relation $enabled(u, \varphi)$ which holds iff $\varphi \land enabled(u)$ is satisfiable. Additionally, two auxiliary relations are used:

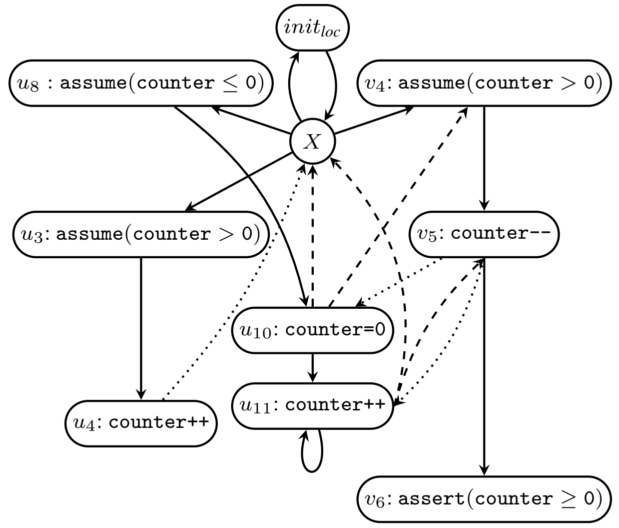
- $coreachable(u, v)$ holds iff there is some ι-feasible trace σ such that $u = endloc(\sigma, 0)$ and $v = endloc(\sigma, 1)$.
- $mayReach(u, x, v, w)$ holds iff there is some ι-feasible trace σ such that $u = latest(\sigma, x, 0)$, $v = endloc(\sigma, 0)$ and $w = endloc(\sigma, 1)$.

Since the set of locations and the set of variables are finite, $coreachable$, $mayReach$, and $\rightsquigarrow_\iota$ all must be finite. As a result, we may compute all members of $\rightsquigarrow_\iota$ in finite time by iteratively applying these rules until no new members of any relation are deduced (i.e., until a fixed point is reached). Moreover, the fact that these relations are all finite allows us to leverage efficient propositional techniques, for example representing relations by binary decision diagrams.

LEMMA 4.5 (Interference analysis soundness & completeness). *Let $u, v \in Loc$ and $x \in Var$. There exists an ι-feasible trace that witnesses data flow the $u \rightarrow^x v$ iff there exists a single-threaded witness, or if $u \rightsquigarrow_\iota^x v$ belongs to the least fixpoint solution of the system of interference rules in Figure 3.*

Note that this lemma implies that, although we lose information in our interference analysis by going from feasible traces to ι-feasible traces, we do *not* lose information by going from n-thread ι-feasible traces to 2-thread ι-feasible traces.

The rules in Figure 3 are a simplified version of the ones we implement in DUET. DUET handles some additional language features (thread creation and atomic blocks) and has several optimizations



$init_{loc}$
u_8 : assume(counter $\leq$ 0)
v_4: assume(counter $>$ 0)
X
u_3: assume(counter $>$ 0)
v_5: counter--
u_{10}: counter=0
u_{11}: counter++
u_4: counter++
v_6: assert(counter $\geq$ 0)

	ι_1	ι_2	ι_3
$init_{loc}$	true	true	true
u_3	counter=0	counter≥0	counter≥0
u_4	counter=0	counter≥0	counter≥0
u_8	counter=0	counter≥0	counter≥0
u_{10}	counter=0	counter=0	counter=0
u_{11}	counter≥0	counter≥0	counter≥0
v_4	counter=0	counter≥0	counter≥0
v_5	false	counter>0	counter>0
v_6	false	counter≥0	counter≥0

Figure 4. COARSEN computation on the program in Figure 1.

to make it more efficient. However, all the essential ideas of the analysis are present in the rules of Figure 3.

5. Iterative coarsening

In Section 3, we gave a method for computing an annotation for a data flow graph; in Section 4, we gave a method for computing data flow edges given an annotation. By incorporating both components into a feedback loop, we obtain our main algorithm, COARSEN (Algorithm 1). Given a parameterized multi-threaded program, COARSEN computes a DFG that represents that program as well as an annotation that is inductive for that DFG.

Given a program P, COARSEN begins by computing a data flow graph $P_1^\sharp$ with only intra-thread (sequential) data flow edges. It then computes an inductive annotation ι_1 for $P_1^\sharp$ as discussed in Section 3. This annotation ι_1 is used as input to the interference analysis of Section 4, which computes the set of ι_1-feasible data flow edges and adds them to $P_1^\sharp$ to obtain a DFG $P_2^\sharp$. After adding these edges, a new (possibly weaker) annotation is computed that is inductive for $P_2^\sharp$. This process continues until a fixed point is reached; that is, until we reach some k such that $P_k^\sharp = P_{k+1}^\sharp$. At this point, $P_k^\sharp$ represents the program P and ι_k is inductive for $P_k^\sharp$, and therefore ι_k overapproximates the reachable thread states of P by Theorem 3.3.

This algorithm makes use of several auxiliary functions, which are defined below.

- SequentialDFG(P) computes a sequential data flow graph for P. This computation is a standard sequential reaching definitions analysis. This graph contains all intra-thread data flow edges, including all those for local variables, and all those originating from uninit.
- ExtractCond(P) computes a set of observable conditions for P by mining the program for locks and predicates q such that q only uses global variables and assume(q) occurs in P.
- Invariants($\langle Loc, DF\rangle$) computes an annotation that is inductive for the DFG $\langle Loc, DF\rangle$ using a modification of the standard worklist algorithm, as discussed in Section 3.
- AbstractAnnotation($\iota, \mathcal{C}$) computes an observable formula $\iota^\sharp(v)$ for every location v that is implied by $\iota(v)$ and is at least as strong as any other observable formula with that property.
- FeasibleDataflows($P, \iota^\sharp$) computes the relation $\rightsquigarrow_\iota$ through a bottom-up evaluation of the logic program given in Figure 3.

EXAMPLE 5.1. *Consider the program in Figure 1. Figure 4 depicts how the DFG and annotation computed by* COARSEN *evolve on this program. For simplicity, we show only information that is relevant to the* counter *variable. In particular, the DFG contains only vertices that modify or block on* counter *and all the* counter*-labeled edges between them, and the annotation is restricted to refer only to the variable* counter.

A special vertex X appears in the DFG to improve readability by "factoring" edges; it does not represent a real DFG vertex. An edge $u \to X$ from some arbitrary vertex u to X represents four edges: $u \to init_{loc}$, $u \to u_8$, $u \to u_3$ and $u \to v_4$. The vertex $init_{loc}$ is the initial location of the program where every thread begins its execution, and which has u_1 and v_1 as its control flow successors. Its action is to assume the condition of the initial state, as in:

$$\texttt{assume}(\texttt{counter} = 0 \land \texttt{lock1} = 0 \land \texttt{lock2} = 0 \land \texttt{batch} \geq 0)$$

The solid edges in Figure 4 are added in the first round, the dashed edges are added in the second round, and the dotted edges are added in the third round. The columns labeled ι_1, ι_2, and ι_3 represent the annotation of the corresponding location in the first, second, and third rounds. Note that there is no edge from $u_1 0$ to v_5. This is very important for proving the assertion at v_6. Since the invariant at $u_1 0$ always remains counter=0*, the interference analysis can infer that there is no feasible path of the program witnessing this edge. Any feasible path of the program that visits $u_1 0$ has to go through a counter increment ($u_1 1$) or an assume statement (v_4) before it can reach v_5, and since each of those paths contains a location modifying counter in the segment from $u_1 0$ to v_5, they cannot be witnesses for a data flow edge from $u_1 0$ to v_5.*

Finally, the correctness condition of Algorithm 1 is stated in the following theorem:

THEOREM 5.2 (Soundness). *For any program P,* COARSEN *computes an annotation ι and a DFG $P^\sharp$ such that $P^\sharp$ represents P, and for every reachable state s of P and thread n, $s[n] \models \iota(s_{loc}(n))$.*

Proof sketch Let ι and $P^\sharp$ be the be the annotation and data flow graph computed by COARSEN. The termination condition

Algorithm 1 Coarsen

```
Input: A program $P = \langle\langle Loc, CF\rangle, GV, LV, \mathcal{L}[\![\cdot]\!]\rangle$
Output: A sound annotation for $P$
  $\langle Loc, DF\rangle \leftarrow$ SequentialDFG($P$)
  $\mathcal{C} \leftarrow$ ExtractCond($P$)
  $DF' \leftarrow \emptyset$
  repeat
    $DF \leftarrow DF \cup DF'$
    $\iota \leftarrow$ Invariants($\langle Loc, DF\rangle$)
    $\iota^\sharp \leftarrow$ AbstractAnnotation($\iota, \mathcal{C}$)
    $DF' \leftarrow$ FeasibleDataflows($P, \iota^\sharp$)
  until $DF' \subseteq DF$
  return $\iota$
```

of Coarsen implies that ι is inductive for $P^\sharp$ and every ι-feasible trace of P is represented in $P^\sharp$. We first prove that every feasible trace τ of P is ι-feasible by induction on τ.

The base case is trivial, since ϵ is ι-feasible for any ι. For the induction step, assume that $\tau\langle n, v\rangle$ is a feasible trace and τ is ι-feasible; we must prove that $\tau\langle n, v\rangle$ is ι-feasible.

Since $\tau\langle n, v\rangle$ is feasible, there must exist some $s \in \tau^\bullet$ and $s' \in State$ such that $\langle s, s'\rangle \in \mathcal{A}[\![\langle n, v\rangle]\!]$. By the definition of $enabled(v)$, we have that $s \models enabled(v)$. We note that for a formula φ with free variables in *GV*, the meaning of $s \models \varphi$ is unambiguous since s_{env} assigns a single value to each global variable $x \in GV$; moreover, we note that if φ's free variables are in *GV* then $s[n] \models \varphi$ iff $s \models \varphi$, since $s[n]$ and s agree on the values of all global variables.

Let $m \in \mathbb{N}$ be an arbitrary thread. Since τ is ι-feasible, τ is represented by the DFG $\langle Loc, DF\rangle$. Since ι is inductive for $\langle Loc, DF\rangle$ and $s \in \tau^\bullet$, we have that $s[m] \models \iota(s_{loc}(m))$ by Corollary 3.5. It follows from the definition of $\iota^\sharp$ that $s \models \iota^\sharp(s_{loc}(m))$, and thus $s \models \iota^\sharp(s_{loc}(m)) \wedge enabled(v)$. Since this holds for all m, $\tau\langle n, v\rangle$ is ι-feasible (noting that $s_{loc}(m) = endloc(\tau, m)$).

Since every feasible trace is an ι-feasible trace, and every ι-feasible trace is represented by $P^\sharp$, every feasible trace is represented by $P^\sharp$, and so P is represented by $P^\sharp$. Finally, let s be a reachable thread state and let $n \in \mathbb{N}$ be a thread. Then there exists some feasible trace τ such that $s \in \tau^\bullet$. Since τ is ι-feasible (by the above argument), it follows that τ is represented by $P^\sharp$. Finally, since τ is represented by $P^\sharp$ and ι is inductive for $P^\sharp$, we have that $s[n] \models \iota(s_{loc}(n))$ by Corollary 3.5. □

5.1 Relational abstract domains

We have discussed only the use of non-relational (also known as independent attribute) abstract domains up to this point. Although such domains are typically very efficient, the fact that they cannot encode relationships between variables limits their expressive power. In our framework, it is possible to use relational domains, such as octagons and polyhedra, by modifying the data flow graph. Instead having data flow graph edges labeled with a single variable, we allow *sets* of variables as labels, indicating that the value of every variable in this set flows from the source to the target. Since a value for each $x \in X$ flows along such an edge $u \rightarrow^X v$, relationships between variables in X can be maintained.

A particularly simple instance of this idea is to create a partition $\mathbb{P}$ of the set of variables *Var* into semantically related sets. Intuitively, we can think of each cell $X \in \mathbb{P}$ as a record-typed variable, with one field for each $x \in X$. For a given partition $\mathbb{P}$ of *Var*, the collecting semantics for a relational DFG $\langle Loc, DF\rangle$ with $\mathbb{P}$-labeled edges is given by the following:

$$\mathrm{VAL}(u, X) = \{e' : \exists e \in \mathrm{OUT}(u).\forall x \in X.e(x) = e'(x)\}$$

$$\mathrm{IN}(v) = \bigcap_{X \in \mathbb{P}} \{e \in TEnv : \exists u \rightarrow^X v \in DF.e \in \mathrm{VAL}(u, X)\}$$

$$\mathrm{OUT}(v) = \begin{cases} TEnv & \text{if } v = \texttt{uninit} \\ \{e' : \exists e \in \mathrm{IN}(v).\langle e, e'\rangle \in \mathcal{L}[\![v]\!]\} & \text{otherwise} \end{cases}$$

Using the collecting semantics as a guideline, it is straightforward to define inductive invariants for relational DFGs. The interference analysis of Section 4 must then be adapted to infer relational data flow edges. Towards this end, we redefine mod to act on cells rather than variables as follows:

$$mod_{\mathbb{P}}(v) = \{X \in \mathbb{P} : mod(v) \cap X \neq \emptyset\}$$

By re-instantiating the interference analysis in Figure 3 with $mod_{\mathbb{P}}$ in place of mod, we obtain our algorithm for calculating data flows in a relational DFG.

Since nonrelational analyses and relational analyses operate on different data flow graphs, it is not generally true (as in the case of sequential analyses) that a relational analysis is necessarily more accurate than a non-relational analysis. There is a positive side and a negative side to grouping variables when it comes to concurrent program analysis. On the positive side, grouping variables together makes it possible to infer relationships between variables, which results in a more precise analysis. On the negative side, grouping variables together may create additional interference edges (resulting in a less precise analysis), as $mod_{\mathbb{P}}(v)$ is generally larger than $mod(v)$ (for $v \in Loc$). For example, if `batch` and `counter` are grouped together in the relational DFG construction for Figure 1, then the interference analysis will infer the $\{\texttt{batch}, \texttt{counter}\}$-labeled edges $v_5 \rightarrow u_{13}$ and $u_{13} \rightarrow v_5$. With these edges present, the invariant that `counter` ≥ 0 at v_5 can no longer be proved, because the inductiveness condition for annotations implies that there is no lower bound for `counter` at v_5. In our experiments in Section 6, there are cases when interval analysis succeeds in proving a property correct when octagon analysis fails, and vice versa.

In Section 6.1, we briefly discuss the simple algorithm that we use to partition variables into semantically related sets. While simple, this algorithm performs fairly well on our benchmarks. However, we believe that there is considerable room for improvement with a better variable grouping algorithm.

6. Experiments

The approach presented in this paper is implemented into a tool called Duet.[3] We used a benchmark suite of 15 Linux device drivers to evaluate Duet. Additionally, we ran Duet on the set of Boolean programs generated by SatAbs [9] from these Linux drivers to compare Duet with two recent techniques on verification of parameterized Boolean programs.

6.1 Implementation

Duet is written in OCaml, and makes use of the CIL front-end for the C language [28] and the goto program front-end distributed with CBMC [8]. Our abstract interpreter uses the APRON library [5] for its numerical abstract domains. We use the BDD-based Datalog implementation bddbddb[33] to perform the interference analysis described in Section 4. Currently, Duet accepts three types of inputs: (1) C programs using *pthreads* library for thread operations, (2) Boolean programs in the input language of *Boom* [19] as an input, or (3) goto programs, as produced by the goto-cc C/C++ frontend (part of the CProver project [1].

[3] For more information on this tool, see `http://duet.cs.toronto.edu`

Device Drivers	#assertions	DUET: Interval Analysis		DUET: Octagon Analysis	
		safe	time	safe	time
i8xx_tco	90	75	1m51s	71	1m25s
ib700wdt	75	64	30s	64	20s
machzwd	87	73	39s	67	14m44s
mixcomwd	91	72	22s	74	25
pcwd	240	147	2m43s	145	23m48s
pcwd_pci	204	187	2m18s	188	2m59s
sbc60xxwdt	91	77	28s	69	11m27s
sc520_wdt	85	71	28s	65	13m20s
sc1200wdt	77	66	34s	66	33s
smsc37b787_wdt	93	80	47s	80	47s
w83877f_wdt	92	78	29s	72	13m24s
w83977f_wdt	101	90	34s	82	34s
wdt	99	88	25s	86	25s
wdt977	88	77	27s	75	28s
wdt_pci	84	67	33s	66	5m33s
total	1597	1312	13m9s	1277	90m21s

Table 1. DUET's Performance on Parameterized Integer Programs, run on an 3.16GHz Intel(R) Core 2(TM) machine with 4GB of RAM.

Our implementation currently inlines all function calls and then performs an intraprocedural analysis.

Alias Analysis. We use a type-based alias analysis to handle pointers. For each variable whose address is not taken, we assign a memory location which receives strong updates. For every type in the program, we assign a memory location which receives weak updates, and each access path of that type (other than variables whose address is not taken) is considered to be a reference to that memory location. The interference analysis implemented in DUET operates on these memory locations rather than variables. This scheme is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is not particularly important for proving array bounds and integer overflow properties. Therefore, we expect the consequences of our unsound and imprecise alias analysis to be negligible.

We use Algorithm 2 to partition variables into semantically related sets for our implementation of octagon analysis. While simple, it seems to be effective in practice when performing an octagon analysis on Boolean abstractions of Linux device drivers.

Algorithm 2 Variable partitioning algorithm

```
Input: A set of program locations $Loc$, a set of local variables $LV$
    and global variables $GV$
Output: A partition of $Var$
  //$\mathbb{P}$ is a disjoint set data structure
  $\mathbb{P} \leftarrow \{\{x\} : x \in Var\}$
  for $v \in Loc$ do
    if $v$ is an assignment statement then
      $vs \leftarrow mod(v) \cup ref(v)$
      if $|vs| = 2 \wedge (vs \subseteq LV \vee vs \subseteq GV)$ then
        Merge the partitions of each $x \in vs$
      end if
    else if $v =$ assume(p) then
      if $vars(v) \subseteq LV \vee vars(v) \subseteq GV$ then
        Merge the partitions of each $x \in vars(v)$
      end if
    end if
  end for
  return $\mathbb{P}$
```

6.2 Evaluation

Below, we provide the results of experimenting with DUET on a collection of Linux device drivers and on Boolean abstractions of those drivers. Since a driver may have arbitrary many clients, it is important to verify these drivers in a parameterized setting.

Parameterized Integer Programs. Table 1 presents the result of running DUET on a collection of 15 Linux device drivers. These drivers are all written in C, and include infinite data (such as integer types). We know of no other tool that can verify numerical properties of such large programs with arbitrarily many threads,[4] and therefore we present the result of running DUET on these integer benchmarks without comparison with other tools.

DDVerify and goto-cc are used to (automatically) process each driver into a fully-inlined goto program annotated with assertions checking array bounds and integer overflows/underflows. With an interval analysis, DUET manages to prove most of the assertions correct (1312 out of a total 1597), and does so in 13 minutes. DUET's performance using an octagon analysis is slightly worse, proving 1277 assertions correct in 90 minutes.

Most false positives for DUET appear to be caused by one of two reasons: imprecision in the abstract domain, and imprecision in how DUET handles the treatment of spinlocks in goto programs. In particular, many drivers use traverse zero-terminated arrays as in the snippet below:

```
for (i=0; array[i]; i++) { ... }
```

Since our abstraction of arrays has no special representation for zero-terminated arrays, DUET flags this as an array bound error (since no upper bound for i can be inferred). Our handling of spinlocks is imprecise because goto programs model them as pointers to integers (which take value either 0 or 1, depending on whether the lock is acquired), access to which is protected by atomic blocks. Due to our imprecise alias analysis, lock acquisitions can only *weakly* update this integer field, which means that two threads can

[4] It is possible to run Boom on the integer benchmarks by first extracting Boolean programs with SatAbs. However, SatAbs is designed for sequential programs rather than concurrent ones, and the Boolean programs extracted by the version of SatAbs that was available at the time we ran our experiments produced poor results: the combined SatAbs+Boom procedure took 2 days and did not prove any assertions correct.

Device Drivers	#programs	LI				Duet: Octagon Analysis			Duet: Interval Analysis		
		safe	unsafe	unknown	timeout	safe	unsafe	timeout	safe	unsafe	timeout
`i8xx_tco`	338	214	14	0	110	259	79	0	198	140	0
`ib700wdt`	181	109	13	0	59	124	56	1	91	90	0
`machzwd`	255	56	24	94	81	182	70	3	148	105	2
`mixcomwd`	178	103	24	0	51	117	59	2	81	95	2
`pcwd`	100	81	1	0	18	74	16	10	44	50	6
`sbc60xxwdt`	174	92	23	0	59	113	60	1	79	94	1
`sc1200wdt`	247	138	13	0	96	178	67	2	138	107	2
`sc520_wdt`	186	15	23	97	51	123	61	2	89	95	2
`smsc37b787_wdt`	340	154	13	0	173	272	65	3	151	187	2
`w83877f_wdt`	230	15	23	97	95	150	77	3	98	128	4
`w83977f_wdt`	389	147	13	0	229	322	65	2	144	243	2
`wdt`	230	109	17	0	104	161	59	10	108	114	8
`wdt977`	351	139	13	0	199	282	67	2	132	218	1
`wdt_pci`	217	10	34	4	169	146	69	2	146	69	2
total	3416	1382	248	292	1494	2503	870	43	1647	1735	34

Table 2. Comparison with linear interfaces [22] for Parameterized Boolean Programs. Average time per benchmark was 16.9s for LI and 3.4s for Duet. Benchmarks were run on an 3.16GHz Intel(R) Core 2(TM) machine with 4GB of RAM.

acquire the same lock. Neither of these sources of imprecision is due to a fundamental limitation of the analysis technique proposed in this paper (or related to concurrency), and we expect that our false positive rate to drop considerably with the core algorithm unchanged.

Parameterized Boolean Programs. Although Boolean programs are not the target of this work, we experimented with them for two reasons: (1) two recent approaches [19, 22] for verification of *parameterized* concurrent programs only accept Boolean programs as their input, and (2) there is no aliasing present in Boolean programs, which limits the scope of implementation-related imprecision for a better evaluation of the core method.

Duet does not require a predicate abstraction phase to handle Linux device drivers, but to present a more fair comparison with the existing tools [19, 22], we also ran Duet on the Boolean abstractions. We compare Duet against two recent algorithms that handle parameterized Boolean programs: dynamic cutoff detection (DCD) from [19], as implemented in Boom, and linear interfaces (LI) from [22], as implemented in Getafix. We compared these tools against our own on the benchmarks used in the papers (as provided by the authors). The programs were generated by SatAbs from a set of Linux device drivers. The input formats of Boom and Getafix are slightly different, so we report the results separately. The `ib700wdt` and `mixcomwd` benchmarks were generated from the same device drivers, but refer to a different set of Boolean programs in Tables 2 and 3. Each LI benchmark consists of a server and a client thread template, where the client template is replicated arbitrarily many times. The client thread template is the device driver code, and the server thread template simulates the OS interacting with the drivers. In the DCD benchmarks, there is a single thread template that is replicated. All benchmarks were run with a timeout of 5 minutes.

Table 2 presents the results of comparison with the LI algorithm on the set of Boolean programs used in [22]. In the LI algorithm, the system is tested under 4 rounds of scheduling to look for a counter example, and if one is not found then an adequacy checker is executed that *may* succeed in proving the program safe for arbitrarily many threads and rounds of scheduling. The *safe* columns refer to the number of instances that were proved safe (for each analysis). The *unsafe* column for LI refers to the instances for which LI found a counterexample (a confirmed bug), while in Duet, it refers to the instances where assertions could not be proved safe.[5] The *timeout* column for LI refers to instances where LI cannot finish checking the program under 4 rounds, or cannot find a counterexample under 4 rounds and the adequacy checker times out while trying to prove the program safe. The *timeout* column for Duet refers to all the instances that Duet cannot prove safe within the timeout limit. The *unknown* column for LI refers to the instances that no counterexample is found, and the adequacy checker finishes but fails to prove the program safe for arbitrary number of threads.

In almost all benchmarks (other than `pcwd`), Duet can prove many more instances safe (and for `pcwd`, it is close: 74 vs. 81). Duet can prove many of the *unknown* and *timeout* instances of LI safe. The small table below presents a different view of the same results (not distinguishing among individual drivers).

		LI			
		safe	unsafe	timeout	unknown
OCT	safe	1320	0	267	916
	unsafe	60	247	25	538
	timeout	2	1	0	40

The above table compares the results of the octagon analysis in Duet with LI. Duet can prove an additional 1183 (267+916) programs correct compared to LI. There are 60 instances that Duet reports a *confirmed false positive* (a program that is known to be safe, but Duet fails to prove safe). There are a total of 2503 (1320+267+916) programs that are proved safe by Duet, and 247 programs that are correctly declared unsafe, and therefore Duet generates a total of 2750 correct answers. This puts the percentage of incorrect answers out of the total number of *confirmed* correct and incorrect answers for Duet at 2.1% (60/(2750+60)).

Table 3 presents the results of comparison with the DCD algorithm on the set of Boolean programs used in [19]. In the DCD algorithm, there is only one thread template which is increasingly replicated until a counterexample or a cutoff point is found (a cutoff point is a number of threads n such every thread state that is reachable with $m \geq n$ threads is also reachable with n threads). For the subset of these benchmarks where DCD does not time out, the cutoff is at most 3 threads. Duet's interval and octagon analysis substantially outperforms DCD in proving programs correct. In

[5] Note that since our approach is not complete, failure to prove an assertion does not imply that the assertion is necessarily false.

Device Drivers	#programs	DCD			DUET: Octagon Analysis			DUET: Interval Analysis		
		safe	unsafe	timeout	safe	unsafe	timeout	safe	unsafe	timeout
`ib700wdt`	132	10	102	20	16	113	3	28	101	3
`mixcomwd`	138	9	108	21	16	118	4	27	107	4

Table 3. Comparison with dynamic cutoff detection (DCD) [19] for parameterized Boolean programs. Average time per benchmark was 24.9s for DCD and 8.2s for DUET. Benchmarks were run on an 800MHz AMD Opteron(tm) machine with 32 GB of RAM.

particular, DUET can prove a total 58 programs correct (with interval and octagon analysis combined) in contrast to 19 for DCD, and there are no programs which DCD proves safe and which DUET cannot.

7. Related Work

Verification and analysis of concurrent programs has been vastly studied. Here, we focus on verification of parameterized concurrent programs and systems which is more relevant to our work.

Extensive research has been done in the area verification of parameterized protocols. These include (but are not limited to) *split invariants* [10], regular model checking [6, 20], parameterized model checking [12], network invariants [16, 21], and exploiting symmetry [13] in the Murϕ tool [16]. *Counter abstraction* [3, 30] has been a useful technique in verifying replicated components, although bounded. As discussed in Section 1, we believe that proving functional correctness of a protocol is much more involved compared to proving program assertions correct in program such as a device driver (the focus of our work). For example, the correctness of Lamport's Bakery protocol [24] requires complex global invariant with quantifiers. In contrast, we expect driver code to use significantly simpler invariants to enforce synchronization, such as a flag being set or a lock being held. Moreover, *global* program properties (such as mutual exclusion) are part of the correctness of a protocol (such as Bakery), whereas we focus on program assertions, which are not expressive enough to relate the states of different threads to each other.

To the best of our knowledge, the body of work on verification of parameterized protocols (some of it mentioned above) does not contain a single tool that can, for example, effectively verify numerical properties of Linux device drivers with unboundedly many threads. Recently, two new approaches [19, 22] have been proposed for verification of parameterized Boolean programs that target Boolean abstractions of Linux device drivers, and are more closely related to our work. In [19], a *cutoff detection* algorithm is proposed to determine a bound on the number of threads needed to explore all thread states. In [22], an under approximation method based on limiting the number of scheduler rounds (as opposed to context-switches) and the use of *linear interfaces* to summarize interference in a round is proposed. We compare against both these algorithms in our experiments in Section 6. The most important advantage of our framework is that we handle unbounded data domains such as integers, whereas these other techniques are limited to Boolean programs. Based on our experiments, it is a significant advantage to be able to apply directly to integer programs, since predicate abstraction does not make any considerations for threads (a very recent work [11] takes a first step towards closing this gap by making predicate abstraction *symmetry-aware*).

In [4], an abstract domain construction was presented which can represent invariants of the form $\forall t.\varphi(t)$, where t is a variable representing a thread $\varphi(t)$ is an abstract value in some base domain. Since such an invariant is an assertion about *all* threads rather than some fixed set, this technique is applicable to parameterized programs. The authors apply their construction to a shape domain and successfully verify linearizability for a number of concurrent data structures, a property that is more complex than the type we consider in this paper. However, computation of abstract transformers for quantified domains is potentially very expensive. Moreover, our computation of thread interference using traces is potentially more accurate than their method, which is state-based.

There are a few recent approaches in concurrent program verification [15, 18, 25] which assume a *fixed* number of threads (and therefore not applicable to parameterized programs and not directly comparable to our work), but are worth mentioning here. The work in [15] uses predicate abstraction to automatically generate environment abstractions for threads in a rely-guarantee based proof method. They attempt to find a modular proof (where the environment of every thread refers only to global variables) if one exists, and generate a non-modular one (local variables can occur everywhere) otherwise. They can prove more sophisticated properties (such as correctness of Bakery algorithm) than DUET, but only for a small fixed number of threads (2-4); therefore, it is hard to draw any fair comparison between the two tools. The work in [18] is based on abstract interpretation, where they *refine* the *transaction graphs* which model interference among threads, using abstract interpretation. At a high level, our work is similar to the approach in [25]; however, there are significant differences (besides the fact that our approach deals with parameterized programs). Our notion of interference is quite different, and our need to iterate our analysis until convergence is for a different reason. In particular, in [25], the notion of interference is *static* in the sense that the structure of the dataflow equations do not change from one round to the next; only the abstract values involved change.

8. Conclusion and future work

We propose a solution to the problem of verifying thread invariants for parameterized multithreaded programs. Our approach is based on an iterative framework consisting of a feedback loop between two components: one that computes data invariants using a data flow graph representation of the program and another that uses the data invariants to infer new data flow edges and update the data flow graph. Our algorithm is sound and terminating, and is applicable to programs with infinite state (e.g., unbounded integers) and unboundedly many threads. We have implemented our approach into a tool, called DUET, and applied it to a selection of Linux device drivers and a large suite of Boolean programs. Our experiments demonstrate the effectiveness of the approach in proving properties of parameterized concurrent programs in terms of both speed and precision.

Aliasing is a big obstacle for any program verification method, and more specifically for concurrency verification. In our framework, distinguishing two global variables as non-aliases can potentially have a huge impact on the patterns of interference among threads, and make or break a proof of correctness. Also, naturally, information about data invariants and thread interferences can result in a more precise alias analysis for concurrent programs. We believe that combining alias analysis inside the feedback loop in our framework is an interesting research question for future work. As discussed in Section 5.1, a more intelligent algorithm for computing variable groups for relational analyses will also definitely improve the precision and scalability of our analysis.

Acknowledgments

We wish to thank Andreas Podelski for his significant role in improving the presentation of this paper. We would also like to thank Patrick and Radhia Cousot for their comments on an earlier version of this paper. Finally, we thank Alexander Kaiser and Gennaro Parlato for providing the Boolean programs and for their help with running Boom and Getafix.

References

[1] J. Alglave, D. Kroening, N. He, A. Ranjan, N. Seghir, and M. Tautschnig. CPROVER project, Nov. 2011. URL `http://www.cprover.org/`.

[2] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234, 2001.

[3] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, pages 64–78, 2009.

[4] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, volume 5123 of *LNCS*, pages 399–413. 2008.

[5] J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[6] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.

[7] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.*, 37:258–269, 2002.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[9] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *TACAS*, pages 570–574, 2005.

[10] A. Cohen and K. S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, pages 149–161, 2008.

[11] A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, pages 356–371, 2011.

[12] E. A. Emerson and V. Kahlon. Model checking large-scale and parameterized resource allocation systems. In *TACAS*, pages 251–265, 2002.

[13] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9:105–131, 1996.

[14] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252, 2003.

[15] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[16] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murϕ;, 1997.

[17] R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89, 1993.

[18] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings in concurrent programs. In *TACAS*, pages 124–138, 2009.

[19] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659. 2010.

[20] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV*, pages 424–435, 1997.

[21] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *CONCUR*, pages 101–115, 2002.

[22] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644. 2010.

[23] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9, 2007.

[24] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[25] A. Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP*, pages 398–418, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8.

[26] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. *SIGPLAN Not.*, 42:327–338, 2007.

[27] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. *SIGPLAN Not.*, 41:308–319, 2006.

[28] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pages 213–228, 2002.

[29] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

[30] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *CAV*, pages 107–122, 2002. ISBN 3-540-43997-8.

[31] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41:320–331, 2006.

[32] N. Sterling. Warlock - a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39:131–144, June 2004. ISSN 0362-1340.