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 cannot 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
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 which 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 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 \( v_9 \) 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 \( v_9 \) holds, then one must prove that the zero value assigned to counter at \( u_{10} \) cannot reach the decrement at \( v_9 \), 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 \( v_9 \) and \( v_9 \). The interesting aspect of this example is that the reverse is also true: since the locations \( u_{10} \) and \( v_9 \) are not protected by a common lock, in order to rule out the undesired interference, one needs to rely on the fact that at \( v_9 \), the value of counter is always strictly positive. In this scenario, if one starts from a weak invariant at \( v_9 \), 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 in more detail. First, we assume that Producer and Consumer are executed sequentially with no interaction, and compute a sequential data flow graph $P_1^S$. Next, a set of data invariants is generated from $P_1^S$. 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 $P_1^S$. For example, this analysis determines that the value of $\text{counter}$ from $u_{12}$ may reach the beginning of the Producer and Consumer threads, and so adds the edges $u_{12} \rightarrow v_4$ and $u_{12} \rightarrow u_3$. The interference deduction unit uses the fact that $\text{counter}=0$ is an invariant at $u_{10}$ to infer that the value of $\text{counter}$ from $u_{10}$ cannot flow to $v_5$ without passing through $u_4$, and therefore, does not add an edge from $u_{10}$ to $v_5$. Data invariants are then generated for the data flow graph $P_2^S$, which determines that $u_4$ and $v_5$ are now reachable, and that $\text{counter} \in [0, \infty)$ at $u_1$. The subsequent interference analysis computes new data flow edges, for example $u_4 \rightarrow u_1$, 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 $P_2^S$ to get a new data flow graph $P_3^S$. The data analysis runs on $P_3^S$, 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 $H = \langle \text{Loc}, CF \rangle$ 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 $H = \langle \text{Loc}, CF \rangle$ 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 initloc. The behaviour of the program $P$ is defined to be that of the infinite parallel composition of all threads $n \in N$. 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 $k \in N$, where $k$ is a parameter. We also note that we lose no generality in assuming that every thread $n \in N$ executes the same code, since several code segments can be simply combined into one.

```plaintext
int Producer(int batch) {
    u1: acquire(lock2)
    u2: acquire(lock1)
    if (*) {  // assume(batch<=0)
        v3: unlock(lock1)
        v4: unlock(lock2)
        v5: return 1
    } else {
        v6: unlock(lock1)
        v7: unlock(lock2)
        v8: return batch
    }
}

void Consumer() {
    u1: lock(lock1)
    while(*) {
        v2: unlock(lock1)
        v3: lock(lock1)
        v4: assume(counter>0)
        v5: counter--
        v6: assert(counter>0)
        v7: unlock(lock1)
        v8: return 1
    }
}
```

Figure 1. Producer-Consumer Example.

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

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

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

The function $\mathcal{L}[\cdot] : \text{Loc} \rightarrow TEnv \times TEnv$ 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 $n \in N$ and a control flow edge $(v, v') \in CF$ an action. If the target $v'$ of the control flow edge $(v, v')$ is understood from the context or irrelevant, we simply write $\langle n, v \rangle$. We use $A[a]$ 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:

$$\langle s, s' \rangle \in A[a] \iff \exists (e, e') \in \mathcal{L}[v] \text{ such that}$$

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

Given a sequence of actions $\rho$ and a subset $N \subseteq N$, the projection of $\rho$ onto $N$, denoted $\rho|_N$, is the subsequence of $\rho$ consisting of all actions whose thread identifiers belong to $N$. A trace $\sigma$ 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 $\text{initloc}$. For a trace $\sigma$, the post states of $\sigma$ (denoted $\sigma^*$) is the set of final states of that execution. A trace $\tau$ is feasible if $\tau^*$ is nonempty. For a trace $\sigma$ and a thread $n \in N$, we use $\text{endloc}(\sigma, n)$ to denote the end location of the CFG path
\[ \sigma_{|c(n)} \]. Note that endloc has the property that \( \forall s \in \sigma^* \), \( \forall n \in \mathbb{N} \), \( s_{loc}(n) = endloc(\sigma, n) \).

A thread-state property is an assertion \( \varphi \) with free variables in \( \text{Var} \). We will denote the set of such formulae by \( \mathcal{F}(\varphi) \) (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 \( \varphi \).

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

\[
\mathcal{L}[\text{acq}] = \{(e, e|\text{lock} \leftarrow 1) : e(\text{lock}) = 0\}
\]

\[
\mathcal{L}[\text{rel}] = \{(e, e|\text{lock} \leftarrow 0) : e(\text{lock}) = 1\}
\]

Note that, given a feasible trace \( \tau \), the fact that \( \tau^* \) is nonempty implies that there is no point along \( \tau \) 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 \( \mathcal{L}[\text{assume}(c)] = \{(e, e) : e \models c\} \). Programs that depend on the initial state satisfying some property \( \varphi \) can be simulated by defining \( \mathcal{L}[\text{initloc}] = \mathcal{L}[\text{assume}(\varphi)] \). 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 \( P^D = (\text{Loc}, \text{DF}) \), where \( \text{DF} \subseteq \text{Loc} \times \text{Var} \times \text{Loc} \) is a set of directed edges labeled by program locations, and where we assume that \( \text{Loc} \) contains an additional location \( \text{uninit} \) (with no incoming or outgoing edges in the control flow graph of \( P \)). We will use \( u \rightarrow v \) to denote the triple \( (u, v, w) \in \text{Loc} \times \text{Var} \times \text{Loc} \).

We define the collecting semantics of a DFG \( P^D = (\text{Loc}, \text{DF}) \) to be the least solution to the following set of equations:

\[
\text{VAL}(u, x) = \{e(x) : e \in \text{OUT}(u)\}
\]

\[
\text{IN}(v) = \bigcap_{x \in \text{Var}} \{e \in \text{TEnv} : \exists u \rightarrow v \in \text{DF}, e(x) \in \text{VAL}(u, x)\}
\]

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

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 \( \text{uninit} \)) 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 \( u_2 \rightarrow v_2 \) represents the constraint that any value for \( c \) after executing \( u_2 \rightarrow v_2 \) is a possible value for \( c \) before executing \( v_2 \) (that is, \( \{e(c) : e \in \text{IN}(v_2)\} \subseteq \{e(c) : e \in \text{OUT}(u_2)\}\)). The two edges from \( \text{uninit} \) to \( \text{initloc} \) indicate that any value is possible for \( incr \) and \( c \) at \( \text{initloc} \), the location at which both threads begin execution. The vertex \( \text{initloc} \) sets the initial state of the program, acting as \( \text{assume}(c = 0 \land \text{incr} = 0) \). Thus, edges originating at \( \text{initloc} \) indicate that a particular variable may get its value from the initial state. The fact that \( v_1 \rightarrow incr \rightarrow u_2 \) does not belong to this graph indicates that the value of \( incr \) at \( v_1 \) is not observable at \( u_2 \).

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

We will assume the existence of a function \( \text{mod} : \text{Loc} \rightarrow \mathcal{P}(\text{Var}) \) that maps every control location to the set of variables modified at that location. We require that \( \text{mod} \) satisfies the following: for any \( x \in \text{Var} \), if there exists some \( (e, e') \in \mathcal{L}[\text{initloc}] \) with \( e(x) \neq e'(x) \), then \( x \in \text{mod}(\sigma) \). A notable feature of the mod we use in practice is that a location of the form \( \text{assume}(\varphi) \) is considered to be a modification of every variable occurring in \( \varphi \). 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 \( \sigma \), variable \( x \), and thread \( n \), we define \( \text{latest}(\sigma, x, n) \) to be the location of the last action to write to the variable \( x \) along \( \sigma \) (or thread \( n \)'s copy of \( x \), if \( x \) is local). More formally, if \( x \) is a global variable, \( \text{latest}(\sigma, x, n) \) is the unique location \( v \) such that \( \sigma = \pi(v, \rho) \) (for some \( m \in \mathbb{N} \)) and where no action (of any thread) along \( \rho \) modifies \( x \) if such a \( v \) exists, and \( \text{uninit} \) otherwise. Similarly, if \( x \) is a local variable, \( \text{latest}(\sigma, x, n) \) is the unique location \( v \) such that \( \sigma = \pi(n, \rho) \) and no action of thread \( n \) along \( \rho \) modifies \( x \) if such a \( v \) exists, and \( \text{uninit} \) otherwise.

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

Conceptually, \( \sigma \) is a witness for \( u \rightarrow v \) if, on \( \sigma \), \( u \) 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 \( P^D = (\text{Loc}, \text{DF}) \) represents a trace \( \sigma \) if, for every \( u, v \in \text{Loc} \) and \( x \in \text{Var} \), if some subtrace (not necessarily proper) \( \sigma' \) of \( \sigma \) is a witness for \( u \rightarrow v \), then \( u \rightarrow v \in \text{DF} \). \( P^D \) represents a program \( P \) if it represents \( \tau \) for every feasible trace \( \tau \) 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 \( \sigma \) be a trace and let \( P^D = (\text{Loc}, \text{DF}) \) be a DFG such that \( P^D \) represents \( \sigma \). Then the collecting semantics of \( P^D \) overapproximates the set of thread states reached by \( \sigma \). Formally, for all \( s \in \sigma^* \), for all \( n \in \mathbb{N} \), we have that \( s[n] \in \text{IN}(s_{loc}(n)) \).

\(^1\) For a similar use of data-flow graphs, see for example [17].
Proof sketch. Let $P^s = \langle \text{Loc, DF} \rangle$ be a DFG and let $s$ be a trace represented by $P^s$. We proceed by induction on $s$.

Base case: follows from the fact that for all $x$, $\text{uninit} \rightarrow x$ is in the initial worklist $\text{DF}$ (since $P^s$ represents $s$ and the fact that for all $x$, $\text{VAL}(\text{uninit}, x) = Z$).

Inductive step: suppose $s$ is a trace such that $P^s$ represents $s$, let $n \in \mathbb{N}$, and let $v = \text{endloc}(\sigma, n)$. We need to show that $s[n] \in \text{IN}(s\text{init}(n))$. To prove this, it is sufficient to show that $s[n] \in \text{VAL}(\text{endloc}(\sigma, n), x, u, x)$. Then $u \rightarrow v \in DF$ because $s$ witnesses this data flow and $P^s$ represents $s$. If $u = \text{uninit}$, we are done since $\text{VAL}(\text{uninit}, x) = Z$. If $u \neq \text{uninit}$, it follows that $\sigma = \pi_m \rho$ for some thread $m \in \mathbb{N}$, trace $\pi$, and sequence of actions $\rho$ such that $x$ is not modified along $\rho$ (by the definition of $\text{latest}(\sigma, x, n)$).

Moreover, we must have that there is some $s' \in \pi^*$ such that $\exists s''$ with $\langle s', s'' \rangle \in A[(m, u)]$ and $s''[n](x) = s[n](x)$. By the induction hypothesis, $s''[m] \in \text{IN}(u)$, so $s''[m] \in \text{OUT}(u)$ and $s[n](x) = s''[n](x) = s''[m](x) \in \text{VAL}(u, x)$.

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 convenience that simplifies the presentation of our algorithm.

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

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 $\iota$-feasible traces (where $\iota$ is a given annotation), and then develop an interference analysis that computes the set of data flow edges that are witnessed by $\iota$-feasible traces. The definition of $\iota$-feasibility is such that if $\iota$ is a “weak enough” annotation, then every feasible trace is $\iota$-feasible. With such an annotation $\iota$, every edge which is witnessed by a feasible trace will also be witnessed by an $\iota$-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. An observable condition is a predicate $e$ with free variables in $G\mathcal{V}$. In the remainder of this section, we assume a fixed finite set of observable conditions, which we denote by $C$. We define the set of observable formulae $\mathcal{F}(\mathcal{G}(\mathcal{V})) \subseteq \mathcal{F}(\mathcal{G})$ to be the set of formulae $\varphi$ that can be expressed as a conjunction $\lor_{i \in C} \varphi_i$, where for each $i$, $\varphi_i \in C$ or $\neg \varphi_i \in C$. An annotation $\iota: \text{Loc} \rightarrow \mathcal{F}(\mathcal{V})$ (along with the set of observable conditions $C$) determines an abstract annotation $\iota^2: \text{Loc} \rightarrow \mathcal{F}(\mathcal{G}(\mathcal{V}))$ that assigns to each location $v \in \text{Loc}$ an observable formula $\varphi$ that is implied by $\iota(v)$ and which is as least as strong as any other observable formula with this property. Thus, going from concrete to abstract (and using $C$ to denote “is more precise than”), we have $\text{IN}(v) \subseteq \iota(v) \subseteq \iota^2(v)$ for any location $v$.

The set of observable conditions $C$ determines an enabling condition enabled: $\text{Loc} \rightarrow \mathcal{F}(\mathcal{V}^2)$ where $\exists e \langle e, e' \rangle \in A[v] \Rightarrow \langle e, v \rangle \mid \text{enabled}(v)$ and enabled$(v)$ is at least as strong as any other observable formula with this property.

We are now ready to state our definition of $\iota$-feasibility:

**Definition 4.1.** Let $\sigma$ be a trace and $\iota$ be an annotation. Then $\sigma$ is $\iota$-feasible if:

- $\sigma = e$, or
- $\sigma = \iota'(n, v)$, where $\iota'$ is an $\iota$-feasible trace, and for all $m \in \mathbb{N}$, $\iota'(\text{endloc}(\sigma, m)) \land \text{enabled}(v)$ is satisfiable.

Note that the condition for extending an $\iota$-feasible path $\sigma$ by an action $(n, v)$ depends only on the annotations at the end locations of each thread, rather than on the states in $\sigma^*$.

**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 $\langle 0, u_1 \rangle \langle 0, u_2 \rangle \langle 0, u_3 \rangle$ has counter = 0, so $(0, u_3)$ is not enabled. However, assuming that $\iota(u_2) = \text{counter} \geq 0 \lor \text{lock1} = 1$, $\iota(u_1) = \text{counter} \geq 0$, and $C = \{\text{counter} \geq 0, \text{lock1} = 0, \text{lock2} = 0\}$ (which implies $\iota^2(v) = \text{true}$, $\iota^2(u_1) = \neg \text{lock1} = 0$, and enabled$(u_1) = \text{counter} > 0$), this trace is $\iota$-feasible. To illustrate how $\iota$-feasibility depends on $\iota$, consider the infeasible trace $\langle 0, u_1 \rangle \langle 0, u_2 \rangle \langle 1, v_1 \rangle$. This trace is
ι-feasible if \( \iota(u_2) = \text{counter} \geq 0 \land \text{lock} = 1 \), and ι-infeasible if \( \iota(u_2) = \text{counter} \geq 0 \land \text{lock} = 1 \land \text{lock} = 1 \).

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 \( \sigma \) be a trace, \( i \) be an annotation, \( u \) and \( v \) be locations, and \( x \) be a variable. Then \( \sigma \) is an ι-feasible witness of the data flow \( u \rightarrow^x v \) if \( \sigma \) is ι-feasible and witnesses the data flow \( u \rightarrow^x v \) (that is, \( \sigma \) 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 \( i \), if \( \sigma \) 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 \( i \) be an annotation, \( u, v \) be locations, and \( x \) be a variable. Let \( \sigma \) 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 \( \sigma, \sigma|_N \) is an ι-feasible trace, by induction on \( \sigma \).

The base case is obvious. For the inductive step, let \( \sigma(n,v) \) be an ι-feasible trace, and assume that \( \sigma|_N \) is ι-feasible. If \( n \not\in N \), then \( \sigma(n,v)|_N = \sigma|_N \), and the induction hypothesis is immediate from the induction hypothesis.

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

- \( m \in N \): then \( \text{endloc}(\sigma|m,n) = \text{endloc}(\sigma,m) \), and the fact that \( \check{i}^x(\text{endloc}(\sigma|m,n)) \land \text{enabled}(v) \) is satisfiable follows from the fact that \( \sigma(n,v) \) is ι-feasible.
- \( m \not\in N \): then \( \text{endloc}(\sigma|m,n) = \text{initloc} \). Since \( \sigma \) is finite, only finitely many threads execute actions in \( \sigma \), so there exists a thread \( i \in \mathbb{N} \) that does not execute actions in \( \sigma \). Since \( \sigma(n,v) \) is ι-feasible, \( \check{i}^x(\text{endloc}(\sigma,i)) \land \text{enabled}(v) \) is satisfiable. Since \( \text{endloc}(\sigma,i) = \text{initloc} \), we are done.

Now, we must prove the property of being a witness is preserved by projections. Let \( u, v \) be locations and \( x \) be a variable, and let \( \sigma \) be 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 \( \pi \), and a sequence of actions \( \rho \) such that \( \sigma = \pi(n,v) $\rho$ such that \( x \in \text{mod}(u) \), \( x \) is not modified by any action along \( \rho \), and \( v = \text{endloc}(\sigma,m) \). 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 \( u, v \in \text{Loc} \) and \( x \in \text{Var} \), we write \( u \leadsto^x_v \) 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 \( \text{enabled}(u, \varphi) \) which holds iff \( \varphi \land \text{enabled}(u) \) is satisfiable. Additionally, two auxiliary relations are used:

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

Since the set of locations and the set of variables are finite, \( \text{coreachable}, \text{mayReach}, \leadsto \), all must be finite. As a result, we may compute all members of \( \leadsto \) 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 \( u, v \in \text{Loc} \) and \( x \in \text{Var} \). There exists an ι-feasible trace that witnesses data flow the \( u \rightarrow^x v \) if and only if there exists a single-threaded witness, or if \( u \leadsto^x v \) belongs to the least fixedpoint 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.
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 $P^1$ with only intra-thread (sequential) data flow edges. It then computes an inductive annotation $\iota_1$ for $P^1$ as discussed in Section 3. This annotation $\iota_1$ is used as input to the interference analysis of Section 4, which computes the set of $\iota_1$-feasible data flow edges and adds them to $P^1$ to obtain a DFG $P^2$. After adding these edges, a new (possibly weaker) annotation is computed that is inductive for $P^2$. This process continues until a fixed point is reached; that is, until we reach some $k$ such that $P^k = P^k_{k+1}$. At this point, $P^k$ represents the program $P$ and $\iota_k$ is inductive for $P^k$, and therefore $\iota_k$ 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 $\text{unit}$.
- 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 $\text{assume}(q)$ occurs in $P$.
- Invariants($\langle \text{Loc, DF} \rangle$) computes an annotation that is inductive for the DFG $\langle \text{Loc, DF} \rangle$ using a modification of the standard worklist algorithm, as discussed in Section 3.
- AbstractAnnotation($\iota, C$) computes an observable formula $\iota^C(v)$ 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($P, \iota^C$) computes the relation $\rightsquigarrow_c$ 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 $u \rightarrow X$ from some arbitrary vertex $u$ to $X$ represents four edges: $u \rightarrow \text{initLoc}, u \rightarrow \text{lock}1, u \rightarrow \text{lock}2$ and $u \rightarrow v_4$. The vertex $\text{initLoc}$ is the initial location of the program where every thread begins its execution, and which has $u_2$ and $v_1$ as its control flow successors. Its action is to assume the condition of the initial state, as in:

$\text{assume}(\text{counter} = 0 \land \text{lock}1 = 0 \land \text{lock}2 = 0 \land \text{batch} \geq 0)$

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 $\iota_1$, $\iota_2$, and $\iota_3$ represent the annotation of the corresponding location in the first, second, and third rounds. Note that there is no edge from $v_5$ to $v_3$. This is very important for proving the assertion at $v_6$. Since the invariant at $v_3$ always remains $\text{counter} = 0$, 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 $v_3$ has to go through a counter increment ($u_1$) 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 $u_1$ to $v_6$, they cannot be witnesses for a data flow edge from $v_3$ to $v_6$.

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 $\iota$ and a DFG $P^k$ such that $P^k$ represents $P$, and for every reachable state $s$ of $P$ and thread $n$, $s[n] \models \iota(\text{initLoc}(n))$.

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

<table>
<thead>
<tr>
<th>$\text{initLoc}$</th>
<th>$\iota_1$</th>
<th>$\iota_2$</th>
<th>$\iota_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$u_2$</td>
<td>counter=0</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>counter=0</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
<tr>
<td>$u_4$</td>
<td>counter=0</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
<tr>
<td>$u_5$</td>
<td>counter=0</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
<tr>
<td>$u_6$</td>
<td>false</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
<tr>
<td>$u_7$</td>
<td>false</td>
<td>counter&gt;0</td>
<td>counter&gt;0</td>
</tr>
</tbody>
</table>
of COARSEN implies that \( \tau \) is inductive for \( P^2 \) and every \( \tau \)-feasible trace of \( P \) is represented in \( P^3 \). We first prove that every feasible trace \( \tau \) of \( P \) is \( \tau \)-feasible by induction on \( \tau \).

The base case is trivial, since \( \epsilon \) is \( \tau \)-feasible for any \( \epsilon \). For the induction step, assume that \( \tau(n, v) \) is a feasible trace and \( \tau \) is \( \tau \)-feasible; we must prove that \( \tau(n, v) \) is \( \tau \)-feasible.

Since \( \tau(n, v) \) is feasible, there must exist some \( s \in \tau^* \) and \( s' \in \text{State} \) such that \( (s, s') \in A\{n, v\} \). By the definition of \( \text{enabled}(v) \), we have that \( s \models \text{enabled}(v) \). We note that for a formula \( \varphi \) with free variables in \( GV \), the meaning of \( s \models \varphi \) is unambiguous since \( s_{\text{num}} \) assigns a single value to each global variable \( x \in GV \); moreover, we note that if \( \varphi \)'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 \( \tau \) is \( \tau \)-feasible, \( \tau \) is represented by the DFG \( \langle \text{Loc}, DF \rangle \). Since \( \epsilon \) is inductive for \( \langle \text{Loc}, DF \rangle \) and \( s \in \tau^* \), we have that \( s[m] \models \epsilon(s_{\text{loc}}(m)) \) by Corollary 3.5.

It follows from the definition of \( \epsilon^* \) that \( s \models \epsilon^*(s_{\text{loc}}(m)) \) and thus \( s \models \epsilon^*(s_{\text{loc}}(m)) \land \text{enabled}(v) \). This holds for all \( m \), \( \tau(n, v) \) is \( \epsilon \)-feasible (noting that \( s_{\text{loc}} \) = \( \epsilon_\text{loc} (\tau, m) \)).

Since every feasible trace is an \( \epsilon \)-feasible trace, and every \( \epsilon \)-feasible trace is represented by \( P^2 \), every feasible trace is represented by \( P^3 \), and so \( P \) is represented by \( P^2 \). Finally, let \( s \) be a reachable thread state and let \( n \in \mathbb{N} \) be a thread. Then there exists some feasible trace \( \tau \) such that \( s \in \tau^* \). Since \( \tau \) is \( \epsilon \)-feasible (by the above argument), it follows that \( \tau \) is represented by \( P^2 \). Finally, since \( \tau \) is represented by \( P^3 \) and \( \epsilon \) is inductive for \( P^2 \), we have that \( s[n] \models \epsilon(s_{\text{loc}}(n)) \) 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 of 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 \( u \rightarrow X \), relationships between variables in \( X \) can be maintained.

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

\[
\begin{align*}
\text{VAL}(u, X) &= \{ e' : \exists e \in \text{OUT}(u), \forall x \in X. e(x) = e'(x) \} \\
\text{IN}(v) &= \bigcup_{X \in P} \{ e \in \text{TE}(v) \rightarrow X. e \in \text{DF} \text{AL}(u, X) \} \\
\text{OUT}(v) &= \begin{cases} \\
\text{TE}(v) \quad &\text{if } v = \text{uninit} \quad \text{otherwise}
\end{cases}
\end{align*}
\]

Using the collecting semantics as a guideline, it is straightforward to define inductive invariants for relational DFGs. The inference 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:

\[
\text{mod}(v) = \{ X \in P : \text{mod}(v) \cap X \neq \emptyset \}
\]

By re-instantiating the interference analysis in Figure 3 with \( mod \) 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 \( mod(v) \) is generally larger than \( \text{mod}(v) \) for \( v \in \text{Loc} \). For example, if \( \text{batch} \) and \( \text{counter} \) are grouped together in the relational DFG construction for Figure 1, then the interference analysis will infer the \{batch, counter\}-labeled edges \( \text{v} \rightarrow \text{v} \) and \( \text{v} \rightarrow \text{v} \). With these edges present, the invariant that \( \text{counter} \geq 0 \) at \( v \) can no longer be proved, because the inductiveness condition for annotations implies that there is no lower bound for \( \text{counter} \) at \( v \). In our experiments in Section 6, there are cases where 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 bddhdb[33] to perform the interference analysis described in Section 4. Currently, DUET accepts three types of inputs: (1) C programs using pathshears 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 CPProver project) [1].

\(^3\)For more information on this tool, see http://duet.cs.toronto.edu
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 sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast. For the benchmark suite used in Section 6.2, aliasing is sound under the assumption that pointer-typed expressions are never cast.

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 $Var$,

// $P$ is a disjoint set data structure

$P \leftarrow \{ \{x\} : x \in Var\}$

for $v \in Loc$ do

if $v$ is an assignment statement then

$vs \leftarrow mod(v) \cup ref(v)$

if $|vs| = 2 \land (vs \subseteq LV \lor vs \subseteq GV)$ then

Merge the partitions of each $x \in vs$

end if

else if $v = assume(p)$ then

if $vars(v) \subseteq LV \lor vars(v) \subseteq GV$ then

Merge the partitions of each $x \in vars(v)$

end if

end if

end for

return $P$

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

<table>
<thead>
<tr>
<th>Device Drivers</th>
<th>#assertions</th>
<th>DUET: Interval Analysis</th>
<th>DUET: Octagon Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>i8xx_tco</td>
<td>90</td>
<td>75</td>
<td>71</td>
</tr>
<tr>
<td>ib700udt</td>
<td>75</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>machzwd</td>
<td>87</td>
<td>73</td>
<td>67</td>
</tr>
<tr>
<td>mixcomd</td>
<td>91</td>
<td>72</td>
<td>74</td>
</tr>
<tr>
<td>pcwd</td>
<td>240</td>
<td>147</td>
<td>145</td>
</tr>
<tr>
<td>pcwd_pci</td>
<td>204</td>
<td>187</td>
<td>188</td>
</tr>
<tr>
<td>sabc00xudt</td>
<td>91</td>
<td>77</td>
<td>69</td>
</tr>
<tr>
<td>sc820_wdt</td>
<td>85</td>
<td>71</td>
<td>65</td>
</tr>
<tr>
<td>sc1200udt</td>
<td>77</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>smnc37b787_wdt</td>
<td>93</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>w83977f_wdt</td>
<td>92</td>
<td>78</td>
<td>72</td>
</tr>
<tr>
<td>w83977f_wdt</td>
<td>101</td>
<td>90</td>
<td>82</td>
</tr>
<tr>
<td>wdt</td>
<td>99</td>
<td>88</td>
<td>86</td>
</tr>
<tr>
<td>wdt977</td>
<td>88</td>
<td>77</td>
<td>75</td>
</tr>
<tr>
<td>wdt_pci</td>
<td>84</td>
<td>67</td>
<td>66</td>
</tr>
<tr>
<td>total</td>
<td>1597</td>
<td>1312</td>
<td>1277</td>
</tr>
</tbody>
</table>

### 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, 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-inline 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:

```c
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 never...

---

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.
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 `ib700wxdt` 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.

In the DCD algorithm, the system is tested under 4 rounds of scheduling to look for a counterexample, 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 where assertions could not be proved safe. 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).

<table>
<thead>
<tr>
<th>Device Drivers</th>
<th>#programs</th>
<th>LI</th>
<th>DUET: Octagon Analysis</th>
<th>DUET: Interval Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>safe</td>
<td>unsafe</td>
<td>unknown</td>
</tr>
<tr>
<td><code>ib700wxdt</code></td>
<td>181</td>
<td>109</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><code>machzwd</code></td>
<td>255</td>
<td>56</td>
<td>24</td>
<td>94</td>
</tr>
<tr>
<td><code>mixcomwd</code></td>
<td>178</td>
<td>103</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td><code>pcwd</code></td>
<td>100</td>
<td>81</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>sbc60xxwdt</code></td>
<td>174</td>
<td>92</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td><code>sci200wudt</code></td>
<td>247</td>
<td>138</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><code>sc520_wdt</code></td>
<td>186</td>
<td>15</td>
<td>23</td>
<td>97</td>
</tr>
<tr>
<td><code>smc3/7b787_wdt</code></td>
<td>340</td>
<td>154</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><code>w33877f_wdt</code></td>
<td>230</td>
<td>15</td>
<td>23</td>
<td>97</td>
</tr>
<tr>
<td><code>w33877f_wdtd</code></td>
<td>389</td>
<td>147</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><code>wdt</code></td>
<td>230</td>
<td>109</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td><code>wdt977</code></td>
<td>351</td>
<td>139</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td><code>wdt_pci</code></td>
<td>217</td>
<td>10</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>3416</strong></td>
<td><strong>1382</strong></td>
<td><strong>248</strong></td>
<td><strong>292</strong></td>
</tr>
</tbody>
</table>

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.

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. All benchmarks were run with a timeout of 5 minutes. The `unknown` column for LI refers to the instances where assertions could not be proved safe, but **DUET** 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.
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.

<table>
<thead>
<tr>
<th>Device Drivers</th>
<th>#programs</th>
<th>DCD</th>
<th>DUET: Octagon Analysis</th>
<th>DUET: Interval Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>ib700wdt</td>
<td>132</td>
<td>safe 10</td>
<td>unsafe 102</td>
<td>timeout 20</td>
</tr>
<tr>
<td>mixcomwd</td>
<td>138</td>
<td>safe 9</td>
<td>unsafe 108</td>
<td>timeout 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>safe 16</td>
<td>unsafe 113</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>safe 16</td>
<td>unsafe 118</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

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.
puting 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