Inductive Data Flow Graphs

Azadeh Farzan   Zachary Kincaid
University of Toronto

Andreas Podelski
University of Freiburg

Abstract

The correctness of a sequential program can be shown by the annotation of its control flow graph with inductive assertions. We propose *inductive data flow graphs*, data flow graphs with incorporated inductive assertions, as the basis of an approach to verifying concurrent programs. An inductive data flow graph accounts for a set of dependencies between program actions in interleaved thread executions, and therefore stands as a representation for the set of concurrent program traces which give rise to these dependencies. The approach first constructs an inductive data flow graph and then checks whether all program traces are represented. The size of the inductive data flow graph is polynomial in the number of data dependencies (in a sense that can be made formal); it does not grow exponentially in the number of threads unless the data dependencies do. The approach shifts the burden of the exponential explosion towards the check whether all program traces are represented, i.e., to a combinatorial problem (over finite graphs).

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness Proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Concurrency, Verification, Static Analysis

1. Introduction

The success of the two main approaches to algorithmic verification is well established for their intended target domains:

- static analysis for the verification of sequential programs [8],
- model checking for the verification of finite-state concurrent protocols [7].

This paper addresses the algorithmic verification of concurrent programs. Considerable progress has been made with approaches that, depending on terminology, extend static analysis to concurrent control or extend model checking to general data domains [3, 10–12, 16, 21, 26]. Each of these approaches provides a different angle of attack to circumvent the same fundamental issue: *the space required to prove the correctness of a concurrent program grows exponentially in the number of its threads* (in at least some practical examples).

We propose *inductive data flow graphs (iDFGs)*, data flow graphs with incorporated inductive assertions, as the basis of an approach to verifying concurrent programs. An iDFG accounts for a set of dependencies between data operations in interleaved thread executions, and therefore stands as a representation for the set of concurrent program traces which give rise to these dependencies. The approach first constructs an iDFG and then checks whether all program traces are represented. The size of an iDFG is polynomial in the number of data dependencies (in a sense that can be made formal); it does not grow exponentially in the number of threads unless the data dependencies do. Intuitively, this succinctness is possible because iDFGs represent only the *data flow* of the program, and abstract away control features that are irrelevant to the proof. The approach shifts the burden of the exponential explosion towards the check whether all program traces are represented, which is a combinatorial problem (over finite graphs).

There are several directions in which one can explore the practical potential of the approach. This is not the focus of this paper. The focus in this paper is to introduce the approach and to investigate its formal foundation.

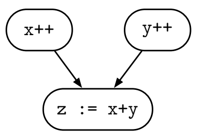
We will next explain in what sense our approach relies on the respective strength of both static analysis and model checking.

In static analysis, techniques have been developed that are successful in finding solutions to fixpoint equations, namely by solving them over *abstract domains*. In the settings where the elements of the abstract domain denote state predicates, static analysis amounts to establishing a Floyd-Hoare annotation of the control flow graph of the program. The validity of the fixpoint equation translates to the inductiveness of the annotation, i.e., the validity of the Hoare triple at each node of the control flow graph. To establish the validity (of the solution for the fixpoint equation/of the Hoare triples), the Floyd-Hoare annotation of the control flow graph has to be stored. This becomes an obstacle for the naive approach to apply static analysis to a concurrent program. The naive approach is to *flatten* the concurrent program, i.e., transform it to a non-deterministic sequential program whose executions include *all* interleaved executions of the concurrent program. The size of the control flow graph of the flattened program grows exponentially with the number of threads.

One motivation behind the work in this paper is the question whether static analysis—with the techniques that are well established for sequential programs [4]—can be put into work for concurrent programs. Our approach does not aim at improving those techniques (for constructing abstract domains, widening, strengthening loop invariants, etc.) but, instead, relies on them. The question is whether the output of the static analysis applied to interleaved executions of the threads of a concurrent program can be assembled (and used) in a space-efficient way. The answer we propose lies in iDFGs.

In model checking, techniques have been developed that are successful for the exhaustive exploration of the finite, though exponentially large state space of concurrent protocols. Model checking is a combinatorial problem that is "easy" in the sense that it can be reduced to search, and therefore can be solved in PSPACE. Our approach relies on the efficiency of space exploration techniques as used in model checking for checking whether all program traces are represented by a given candidate iDFG proof. The check is a combinatorial problem which is directly related to model checking, and inherits its theoretical complexity bound; i.e., the check can be implemented in polynomial space (polynomial in the number of threads of the given concurrent program).

We will next explain how the concept of *data dependencies* in connection with *Hoare triples* leads to iDFGs.



Assume trace τ is composed of three actions `a1: x++`, `a2: y++`, and `a3: z:=x+y` (possibly from different threads of a concurrent program). The two edges in the graph to the right express the data dependences in τ. Conceptually, the graph represents the set of all traces that preserve the data dependences expressed by its edges. Such traces contain the three actions in an order that complies with the two edges (i.e., `a1` and `a2` must occur before `a3`); they can contain additional actions as long as these actions do not modify the value of `x`, `y`, or `z`. All such traces have the same end state, which formally means that they satisfy the same set of pre/postcondition pairs. Now, in the setting of verification where correctness is defined by one particular pre/postcondition pair, we would like to have a more targeted sense of data dependencies (and thus a means to define a more targeted set of traces).

As an example, let us take the pre/postcondition pair that defines the correctness of the trace τ in the Hoare triple:

$$\{\texttt{x} \geq 0 \wedge \texttt{y} \geq 0\}\ \texttt{x++}\ ;\ \texttt{y++}\ ;\ \texttt{z:=x+y}\ \{\texttt{z} > 0\}.$$

The *essence* of the correctness proof for τ consists of the three Hoare triples below.

$$\begin{array}{rcl} \{\texttt{x} > 0 \wedge \texttt{y} > 0\} & \texttt{z:=x+y} & \{\texttt{z} > 0\} \\ \{\texttt{y} \geq 0\} & \texttt{y++} & \{\texttt{y} > 0\} \\ \{\texttt{x} \geq 0\} & \texttt{x++} & \{\texttt{x} > 0\} \end{array} \quad (1)$$

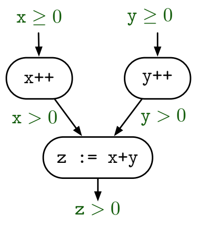
In addition to these essential Hoare triples, the correctness proof for τ requires the following *stability* Hoare triples, which state the irrelevance of an action with respect to an assertion:

$$\begin{array}{rcl} \{\texttt{y} \geq 0\} & \texttt{x++} & \{\texttt{y} \geq 0\} \\ \{\texttt{x} > 0\} & \texttt{y++} & \{\texttt{x} > 0\} \end{array}$$

We obtain the two Hoare triples below by taking the sequential composition of Hoare triples that appear above.

$$\begin{array}{rcl} \{\texttt{x} \geq 0\} & \texttt{x++}\ ;\ \texttt{y++} & \{\texttt{x} > 0\} \\ \{\texttt{y} \geq 0\} & \texttt{x++}\ ;\ \texttt{y++} & \{\texttt{y} > 0\} \end{array}$$

To obtain a correctness proof for τ, we apply the conjunction rule to these two Hoare triples and then sequentially compose with the Hoare triple for `z:=x+y` in (1).



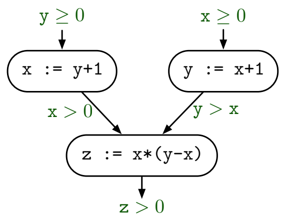
The iDFG to the right is constructed from the correctness proof for the trace τ. The annotation of edges with assertions corresponds to the three Hoare triples in (1) which form the essence of the correctness proof.

We consider this iDFG to be a representation the set of all traces that have the three Hoare triples in (1) for the essence of the correctness proof (all other Hoare triples in the correctness proof state just the stability of an assertion under an action). By definition, all traces in this set are correct. We next give examples of traces in the set.

If we permute the actions `a1` and `a2` in the trace τ, the resulting trace τ' lies in the set: the essence of the correctness proof for τ' is the three Hoare triples in (1); the other Hoare triples in the correctness proof for τ' state the stability of $\texttt{x} \geq 0$ under the action `y++` and the stability of $\texttt{y} > 0$ under the action `x++`.

If we add actions to the trace τ (or to the trace τ') and the assertion at each position with an added action is stable under the added action, then the new trace lies in the set: the essence of its correctness proof are the Hoare triples in (1); the new Hoare triples in the correctness proof are stability Hoare triples.

The edges in an iDFG explicitly formulate ordering constraints on the actions in a trace (as in the example above). We will next illustrate that the assertion labels of edges implicitly define additional ordering constraints.



The iDFG shown to the right can be constructed from the correctness proof for the trace composed of the actions `b1: x:=y+1`, `b2: y:=x+1`, and `b3: z:=x*(y-x)` (we use the same pre/postcondition pair as above). The assertions labeling the two edges are $\texttt{x} > 0$ and $\texttt{y} > \texttt{x}$. The assertion $\texttt{x} > 0$ is stable under the action `b2` (the Hoare triple $\{\texttt{x} > 0\}$ `y:=x+1` $\{\texttt{x} > 0\}$ holds). However, the assertion $\texttt{y} > \texttt{x}$ is not stable under the action `b1` (the Hoare triple $\{\texttt{y} > \texttt{x}\}$ `x:=y+1` $\{\texttt{y} > \texttt{x}\}$ does not hold). This means that the action `b2` may come after the action `b1`, but not vice versa. That is, the trace `b1.b2.b3` lies in the set of traces represented by the iDFG, but the trace `b2.b1.b3` does not. Note that the trace `b1.b2.b3` is correct wrt the specification but the trace `b2.b1.b3` is not.

We use iDFGs as the basis of an algorithm to verify concurrent programs (see also Figure 3). The algorithm iteratively picks a program trace, applies a static analysis to the program trace, uses the output of the static analysis to *construct* an iDFG, *merges* the new iDFG with the one constructed so far, and *checks* whether the set of traces that is represented by the resulting iDFG includes all program traces. If this is the case, the iteration stops; the program is proven correct.

In Section 2 we will see examples in the context of concurrent programs. Sections 3 to 7 then present the individual conceptual and technical contributions of our work:

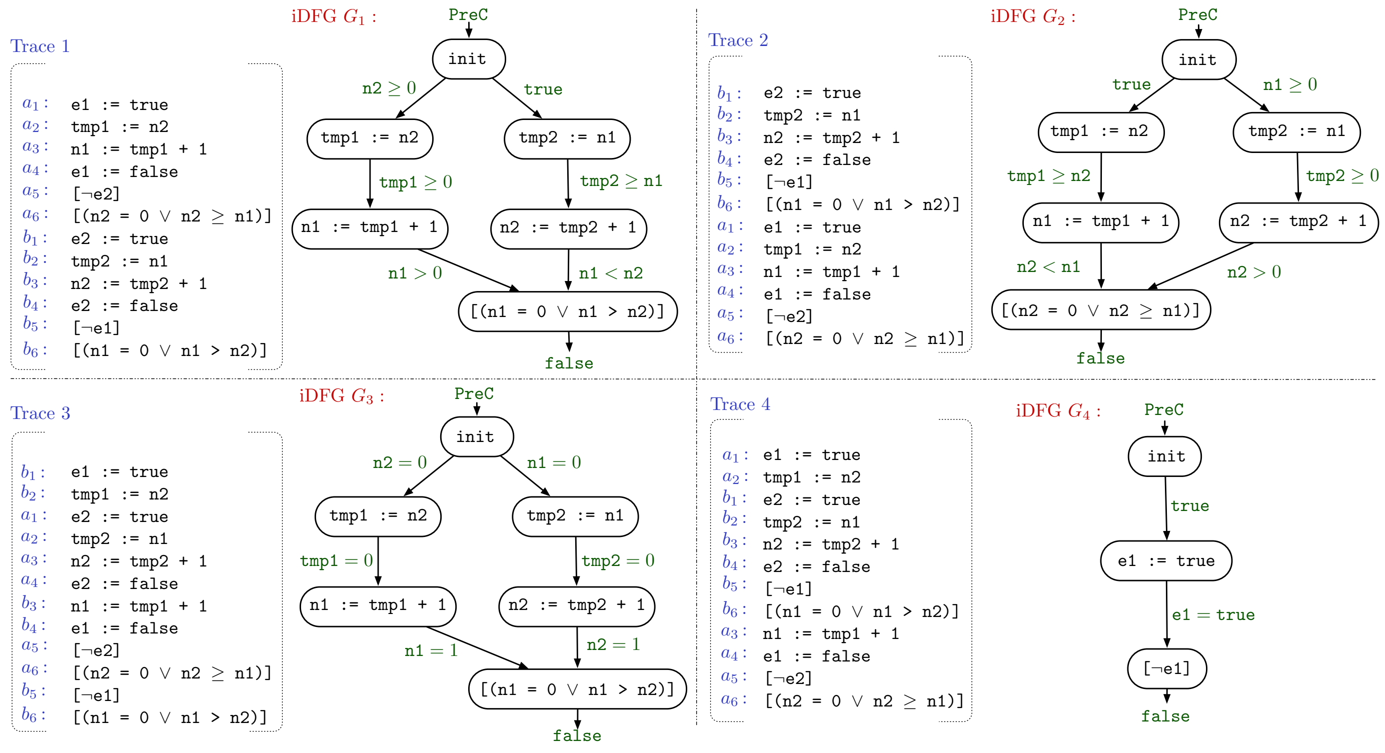
- the concept of iDFGs as a representation of a set of interleavings of thread executions, and the characterization of program correctness by an iDFG (Section 3),
- the reduction of the *check* of program correctness, given a program and an iDFG, to a combinatorial problem in PSPACE (Section 4),
- the formalization of a measure of data complexity and the proof that iDFGs can be polynomial in this measure (Section 5),
- a verification algorithm (Section 6), and
- the proof that the algorithm constructs a polynomial-size iDFG under the assumption that the underlying static analysis provides the required inductive assertions (Section 7).

2. Examples

In this section, we use a few simple examples of concurrent programs to illustrate iDFGs.

Lamport's Bakery Algorithm

The code below implements a version of Lamport's mutual exclusion protocol for two threads [23]. The integer variables `n1` and `n2`



Trace 1
```
a1:  e1 := true
a2:  tmp1 := n2
a3:  n1 := tmp1 + 1
a4:  e1 := false
a5:  [¬e2]
a6:  [(n2 = 0 ∨ n2 ≥ n1)]
b1:  e2 := true
b2:  tmp2 := n1
b3:  n2 := tmp2 + 1
b4:  e2 := false
b5:  [¬e1]
b6:  [(n1 = 0 ∨ n1 > n2)]
```

Trace 2
```
b1:  e2 := true
b2:  tmp2 := n1
b3:  n2 := tmp2 + 1
b4:  e2 := false
b5:  [¬e1]
b6:  [(n1 = 0 ∨ n1 > n2)]
a1:  e1 := true
a2:  tmp1 := n2
a3:  n1 := tmp1 + 1
a4:  e1 := false
a5:  [¬e2]
a6:  [(n2 = 0 ∨ n2 ≥ n1)]
```

Trace 3
```
b1:  e1 := true
b2:  tmp1 := n2
a1:  e2 := true
a2:  tmp2 := n1
a3:  n2 := tmp2 + 1
a4:  e2 := false
b3:  n1 := tmp1 + 1
b4:  e1 := false
a5:  [¬e2]
a6:  [(n2 = 0 ∨ n2 ≥ n1)]
b5:  [¬e1]
b6:  [(n1 = 0 ∨ n1 > n2)]
```

Trace 4
```
a1:  e1 := true
a2:  tmp1 := n2
b1:  e2 := true
b2:  tmp2 := n1
b3:  n2 := tmp2 + 1
b4:  e2 := false
b5:  [¬e1]
b6:  [(n1 = 0 ∨ n1 > n2)]
a3:  n1 := tmp1 + 1
a4:  e1 := false
a5:  [¬e2]
a6:  [(n2 = 0 ∨ n2 ≥ n1)]
```

Figure 1. Four traces of the Bakery algorithm and their corresponding iDFGs. The four iDFGs constitute a proof of the correctness of the mutual exclusion property. Correctness here means: every trace that violates mutual exclusion is infeasible, i.e., has postcondition *false*. The precondition `PreC` is $\texttt{n1} = \texttt{n2} = 0 \wedge \texttt{e1} = \texttt{e2} = \texttt{false}$. The action `init` is a dummy label of the initial node in an iDFG.

(the *tickets*) are initially set to 0, the boolean variables `e1` and `e2` to *false*. We use the notation `[exp]` for *assume* statements ("`if(exp) then skip else block`"). Thread B cannot enter its critical section unless either $\texttt{n1} = 0$ (Thread A has not asked to enter the critical section) or $\texttt{n1} < \texttt{n2}$ (Thread A has asked but it has done so after Thread B). Symmetrically for Thread A, except if there is a tie ($\texttt{n1} = \texttt{n2}$), Thread A has priority over Thread B to enter its critical section first. We break the ticket acquisition statement (`n1 := n2 + 1`) into two statements `tmp1 := n2` and `n1 := tmp1 + 1` to reflect the possibility of the update happening non-atomically (we treat individual program statements as atomic). The flag `e1` (initially `false`) is used to communicate that Thread A is *entering* (that is, it is in the process of acquiring its ticket). The flag `e2` is used similarly for Thread B.

Thread A
```
a1:  e1 := true
a2:  tmp1 := n2
a3:  n1 := tmp1 + 1
a4:  e1 := false
a5:  [¬e2]
a6:  [(n2 = 0 ∨ n2 ≥ n1)]
     // critical section
a7:  n1 := 0
```

Thread B
```
b1:  e2 := true
b2:  tmp2 := n1
b3:  n2 := tmp2 + 1
b4:  e2 := false
b5:  [¬e1]
b6:  [(n1 = 0 ∨ n1 > n2)]
     // critical section
b7:  n2 := 0
```

We wish to verify that this protocol guarantees mutual exclusion: only one thread at a time can be in its critical section. Each trace leading to a state where both threads are in their critical sections (after executing a_6 respectively b_6) must be infeasible (i.e., there exists no initial state that has an execution along the trace) or, equivalently, must have the postcondition `false`.

Trace 1 in Figure 1 is a run of Bakery where first Thread A runs until it enters its critical section and then Thread B runs until it enters its critical section. The iDFG G_1 in Figure 1 expresses the essence of the proof that Trace 1 is infeasible. This graph should be read from bottom to top, and should be thought of as backwards proof argument, along the lines of the following dialogue between *Student* and Teacher:

Why is Trace 1 infeasible? – Thread B can not execute the statement b_6. – *Why not?* – When Thread B tries to execute b_6, $\texttt{n1} > 0$ (Thread A has a ticket) and $\texttt{n1} < \texttt{n2}$ (Thread A's ticket is smaller than Thread B's ticket).

The dialogue then bifurcates:

- *Why is* $\texttt{n1} > 0$ *at* b_6*?* – Thread A executes `n1:=tmp1+1` at a_3 when $\texttt{tmp1} \geq 0$. – *But what about everything that happens between* a_3 *and* b_6*?* – Irrelevant. Nothing between a_3 and b_6 can change the fact that $\texttt{n1} > 0$. – *Why is* $\texttt{tmp1} \geq 0$ *at* a_3*?* – ...
- *Why is* $\texttt{n1} < \texttt{n2}$ *at* b_6*?* – ...

The iDFG G_1 in Figure 1 keeps account of the relevant details of the trace (e.g., the edge labeled $\texttt{n1} > 0$ represents that a_3 must be executed before b_6, and nothing executed between a_3 and b_6 may change the fact that $\texttt{n1} > 0$). It abstracts away irrelevant details of the trace (e.g., the ordering between a_2 and b_2). The iDFG represents the set of traces which are infeasible for essentially the same reason as Trace 1 (i.e., which have essentially the same Hoare triples in the proof that the postcondition is `false`). The traces in the set can be obtained from Trace 1 by reordering actions or by inserting actions that are irrelevant for the proof (they leave the corresponding assertion stable). Informally, the set contains *all traces where Thread B enters its critical section while Thread A has a smaller (nonzero) ticket.*

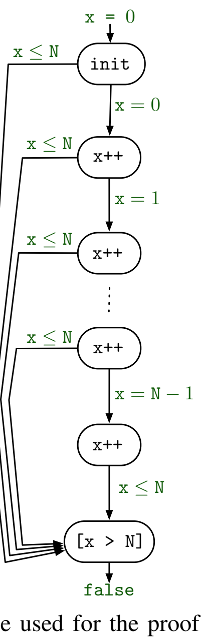
Every trace that violates mutual exclusion falls into one of four scenarios, each scenario being exemplified by a trace in Figure 1. The scenarios are: (Trace 1) Thread B attempts to enter its critical section when Thread A has a smaller ticket, (Trace 2) the symmetric scenario where Thread A attempts to enter its critical section when Thread B has a smaller ticket, (Trace 3) Thread B attempts to enter its critical section when Thread A and B have the same ticket (recalling that Thread A has priority in this situation), (Trace 4) Thread B enters its critical section without waiting for Thread A to receive its ticket. The iDFGs $G_1,\dots, G_4$ express the essence of the proof of the infeasibility of each trace that falls into the corresponding scenario. Together, they form a proof of the correctness of Bakery. This is because each trace of Bakery falls into one of the four scenarios; formally, it lies in one of the four sets represented by $G_1,\dots, G_4$ (which one can check with the algorithm of Section 4). Note that there is no need for a new iDFG for the trace symmetric to Trace 4. The corresponding scenario is accounted for by the scenario of Trace 3. This asymmetry is due to the asymmetry of the protocol (Thread A has priority over Thread B if they have the same ticket).

Increment Example

We use a simple program with a parametrized number of threads to illustrate the exponential succinctness of data flow graphs. The program has one global variable `x` (initially `0`) and is composed of a thread that asserts $\mathtt{x} \leq 0$ and a parametrized number N of threads that each increment `x` once.

```
Thread 0                    Thread 1           Thread N
a_0 :  assert(x ≤ N)        a_1 :  x++    ···   a_N :  x++
```

The program is *safe* (the `assert` never fails) if $\mathtt{x} \leq \mathtt{N}$ is an invariant. Formally, the correctness of the program means that each trace containing the assume statement `[x>N]` is infeasible, i.e., has postcondition `false`. The iDFG shown on the right is a proof of the correctness of the program. Its size *linear* in N. As one can check algorithmically (Section 4), the set of traces it represents contains all program traces. Thus, it is a proof of the correctness of the program.



The program is based on an example that was used in [28] to illustrate the need for auxiliary variables in compositional proof rules. The program is a challenge for many existing approaches. We ran experiments, for example, with the tool THREADER [16], which generates Owicki-Gries type proofs and rely-guarantee type proofs, and with the tool SLAB [11], which uses abstraction-refinement using Craig interpolation with slicing. In both tools, the space used for the proof grows exponentially in N.

If we consider the increment actions of each thread as pairwise distinct (i.e., the program has N actions instead of just one), then we can give a correctness proof in the form of an iDFG of size *quadratic* in N. The graph can be depicted as an $(N+1)\times N$ matrix of vertices. Each column i contains N vertices labeled with the increment action of the i-th thread (plus the vertex labeled `[x>N]`). Each vertex in row j has an edge to every vertex in row $j + 1$. The set represented by this iDFG thus contains traces with more than one occurrences of the increment action of the same thread (for example, the traces corresponding to a column). These traces do not correspond to a path in the control flow graph but they are correct. This illustrates the point that a set of correct traces can represented more compactly if it is larger.

Ticket Algorithm

In the parametrized version of the ticket algorithm [1] depicted below, the statement at line $\ell_{1,i}$ (for $i \in [1, N]$) indicates an atomic operation that increments `t` after assigning its current value to $\mathtt{m_i}$. Initially `s = t = 0`. The algorithm is a challenge for many existing approaches to concurrent program verification in that, as with the increment example, the space used for the proof will grow exponentially in N.

```
Thread 1                                   Thread N
ℓ_{1,1} :  m_1 := t++                      ℓ_{N,1} :  m_N := t++
ℓ_{1,2} :  [m_1 ≤ s]                       ℓ_{N,2} :  [m_N ≤ s]
           // critical section      ···               // critical section
ℓ_{1,3} :  s := s + 1                      ℓ_{N,3} :  s := s + 1
```

We can give a correctness proof in the form of an iDFG of *quadratic* size in N. It is related to the one for the increment example. We present its construction (for $N = 3$) as an example in Section 6.

3. Inductive Data Flow Graphs (iDFGs)

We will first introduce the notation needed to define the correctness of programs. We then introduce the notion of inductive data flow graphs and use it to characterize program correctness.

To abstract away from the specifics of a programming language and from the specifics of a program analysis and verification environment, we use a very general setup. A program is given as an edge-labeled graph $\mathcal{P} = \langle \mathsf{Loc}, \delta\rangle$ (the *control flow graph*) where the edge labels are taken from the set Actions (a given set of *actions*); i.e., $\delta \subseteq \mathsf{Loc}\times\mathsf{Actions}\times\mathsf{Loc}$. We say that the edge between the two vertices (*locations*) ℓ and ℓ' is labeled by the action a if $(\ell, a, \ell') \in \delta$.

The program $\mathcal{P} = \langle \mathsf{Loc}, \delta\rangle$ can be defined as the *parallel composition* of a number N of programs $\mathcal{P}_1 = \langle \mathsf{Loc}_1, \delta_1\rangle, \dots, \mathcal{P}_N = \langle \mathsf{Loc}_N, \delta_N\rangle$ (we will say that $\mathcal{P}_1, \dots, \mathcal{P}_N$ are the *threads* of the program $\mathcal{P}$). Its set of locations is the Cartesian product of the sets of thread locations, i.e., $\mathsf{Loc} = \mathsf{Loc}_1 \times\cdots\times \mathsf{Loc}_N$. Each edge (ℓ_i, a_i, ℓ'_i) in the i-th thread $\mathcal{P}_i$ gives rise to an edge (ℓ, a_i, ℓ') in the program $\mathcal{P}$; the edge is labeled by the same action a_i and goes from the location $\ell = (\ell_1, \dots, \ell_i, \dots, \ell_N)$ to the location $\ell' = (\ell_1, \dots, \ell'_i, \dots, \ell_N)$ (i.e., only the i-th component can change). In algorithms that take $\mathcal{P}_1, \dots, \mathcal{P}_N$ as the input, the control flow graph for the program $\mathcal{P}$ is generally not constructed explicitly.

We assume a set Φ of *assertions*. Each assertion φ is a first-order logic formula over a given vocabulary that includes a set Var of variables (the *program variables*). We assume that the set of assertions comes with a binary relation, the *entailment* relation.

Each action a comes with a binary relation between assertions (the set of its *precondition/postcondition pairs*). We say that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ a\ \{\varphi_{\mathsf{post}}\}$ is valid if the binary relation for the action a holds between the assertions φ_{pre} and φ_{post}.

It is useful to suppose that we have actions that correspond to *assume* statements. That is, for every assertion ψ we have an action $[\psi]$ such that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ [\psi]\ \{\varphi_{\mathsf{post}}\}$ is valid if the assertion φ_{post} is entailed by the conjunction $\psi \wedge \varphi_{\mathsf{pre}}$.

A *trace* τ is a sequence of actions, say $\tau = a_1 \dots a_n$. We extend the validity of Hoare triples to traces in the canonical way. The Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid for the empty trace ε if φ_{pre} entails φ_{post}. It is valid for the trace $\tau = a_1 \dots a_n$ if each of the Hoare triples $\{\varphi_{\mathsf{pre}}\}\ a_1\ \{\varphi_1\}, \dots, \{\varphi_{n-1}\}\ a\ \{\varphi_{\mathsf{post}}\}$

is valid. Later, when we formulate algorithms, we will abstract away from the specific procedure (static analysis, interpolant generation, ...) used to construct the sequence of intermediate assertions $\varphi_1, \ldots, \varphi_{n-1}$.

To define the correctness of the program $\mathcal{P}$, we need define its set of *program traces*. We assume that the control flow graph of the program $\mathcal{P}$ comes with an *initial* location ℓ_0 and a set F of *final* locations. We say that the trace τ is a program trace of $\mathcal{P}$ if τ labels a path between the initial and a final location. Not every such path corresponds to a possible execution of the program. The set of program traces is generally not prefix-closed.

We say that the program $\mathcal{P}$ is *correct* wrt. to the pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid for every program trace τ of $\mathcal{P}$. We will later characterize program correctness in terms of the notion that we introduce next.

Definition 3.1 (Inductive Data Flow Graph, iDFG) An *inductive data flow graph* (iDFG)

$$G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$$

consists of a vertex-labeled edge-labeled graph $\langle V, E \rangle$, an assertion φ_{pre} (the *precondition*), an assertion φ_{post} (the *postcondition*), a vertex $v_0 \in V$ (the *initial vertex*), and the subset of vertices $V_{\mathsf{final}} \subseteq V$ (the set of *final vertices*).

- Vertices are labeled by actions. We use $\mathtt{act}(v)$ to denote the action that labels v. The initial vertex v_0 has a dummy label: the special action `init` which has the same Hoare triples as `skip`, i.e., $\{\varphi\}$ `init` $\{\varphi'\}$ holds if φ entails φ'.
- Edge are labeled by assertions, i.e., $E \subseteq V \times \Phi \times V$. We require that the initial vertex v_0 has no incoming edges. We will use the notation $v \xrightarrow{\varphi} v'$ to denote an edge from v to v' labeled by the assertion φ.
- The labeling of edges with assertions is *inductive*, i.e., for every vertex v labeled with, say, the action a, the Hoare triple $\{\psi\}\ a\ \{\psi'\}$ holds for assertions ψ and ψ' chosen as follows.
  - If v is the initial vertex (i.e., $v = v_0$), then ψ is the precondition φ_{pre}; otherwise, ψ is the conjunction of the assertions labeling the incoming edges to v.
  - If v is a final vertex (i.e., $v \in V_{\mathsf{final}}$), then ψ' is the conjunction of the postcondition φ_{post} and the assertions labeling the outgoing edges of v; otherwise ψ' is just the conjunction of the assertions labeling the outgoing edges of v. ⌟

Remark 3.2 In the special case of an iDFG G where the initial vertex is also a final vertex ($v_0 \in V_{\mathsf{final}}$), Definition 3.1 implies that the precondition of G entails its postcondition ($\varphi_{\mathsf{pre}} \models \varphi_{\mathsf{post}}$). ⌟

We will use an iDFG G as a representation of a set of traces $[\![G]\!]$ (the *denotation* of G). The set $[\![G]\!]$ consists of those traces τ for which G can be used—in a prescribed way—to justify the correctness of τ (in this paper, whenever the context is given by an iDFG G, the term *correctness* refers to the pre/postcondition pair of G). We will next use an example to explain how one uses an iDFG G to justify the correctness of a trace τ, i.e., how one derives that $\tau \in [\![G]\!]$.

Consider the iDFGs $G_0, \ldots, G_4$ in Figure 2 (which all have the same precondition, $\varphi_{\mathsf{pre}} = \mathtt{x} \geq 0 \land \mathtt{y} \geq 0$, but different postconditions), the trace τ_0 = x++.y++.z:=x+y, and its prefix traces τ_1 = x++.y++, τ_2 = x++, and $\tau_3 = \varepsilon$. We will derive $\tau_0 \in [\![G_0]\!]$ via $\tau_3 \in [\![G_3]\!]$, $\tau_2 \in [\![G_1]\!]$, and $\tau_1 \in [\![G_1]\!]$ in one derivation branch and via $\tau_3 \in [\![G_4]\!]$, $\tau_2 \in [\![G_4]\!]$, and $\tau_1 \in [\![G_2]\!]$ in the other.

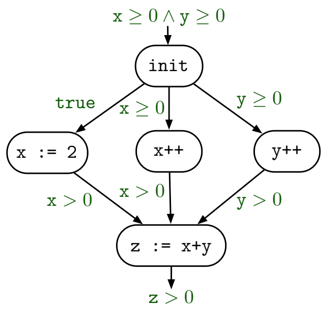
By Remark 3.2, the precondition of G_3 entails its postcondition, so we can use G_3 to justify the *correctness* of the empty trace wrt. the specification of G_3 (i.e., the validity of $\{\varphi_{\mathsf{pre}}\}\ \varepsilon\ \{\mathtt{x} \geq 0\}$). Thus, $\tau_3 \in [\![G_3]\!]$.

The iDFG G_1 has the postcondition $\mathtt{x} > 0$. The trace τ_2 is of the form $\tau_2 = \tau_3$.x++. We have already used G_3 to justify that τ_3 has the postcondition $\mathtt{x} \geq 0$. Since, by the inductiveness of G_1, the Hoare triple $\{\mathtt{x} \geq 0\}$ `x++` $\{\mathtt{x} > 0\}$ is valid, we can use G_1 to justify the correctness of τ_2 (wrt. the postcondition $\mathtt{x} > 0$). Thus, $\tau_2 \in [\![G_1]\!]$.

The trace τ_1 is of the form $\tau_1 = \tau_2$.y++. We have already used G_1 to justify that τ_2 has the postcondition $\mathtt{x} > 0$. The iDFG G_1 has the postcondition $\mathtt{x} > 0$ which is *stable* under the action `y++`, i.e., $\{\mathtt{x} > 0\}$ `y++` $\{\mathtt{x} > 0\}$ is valid. Taken together, this means that we can use G_1 also to justify the correctness of τ_1 (wrt. the postcondition $\mathtt{x} > 0$). Thus, $\tau_1 \in [\![G_1]\!]$.

We use G_2 to justify the correctness of τ_1 (now wrt. the postcondition $\mathtt{y} > 0$). In three steps that are similar but not symmetric to the previous one, we derive (1) $\varepsilon \in [\![G_4]\!]$ (by Remark 3.2), (2) $\tau_2 \in [\![G_4]\!]$ (because $\tau_2 = \varepsilon$.x++ and the postcondition of G_4 is stable under the action `x++`, i.e., $\{\mathtt{y} \geq 0\}$ `y++` $\{\mathtt{y} \geq 0\}$), and (3) $\tau_1 \in [\![G_2]\!]$ ($\tau_1 = \tau_2$.y++ and, by the inductiveness of G_1, $\{\mathtt{y} \geq 0\}$ `x++` $\{\mathtt{y} > 0\}$ is valid).

The trace τ_0 is of the form $\tau_0 = \tau_1$.`z:=x+y`. We can use G_1 and G_2 *together* to justify that τ_1 has the postcondition $\mathtt{x} > 0 \land \mathtt{y} > 0$ which is the *conjunction* of the postconditions of G_1 resp. G_2. Since, by the inductiveness of G_0, $\{\mathtt{x} > 0 \land \mathtt{y} > 0\}$ `z:=x+y` $\{\mathtt{z} > 0\}$ is valid, we can use G_0 to justify the correctness of τ_0 (wrt. the postcondition $\mathtt{z} > 0$). Thus, $\tau_0 \in [\![G_0]\!]$.



In the derivation above, the iDFGs G_1 and G_2 corresponding to the two incoming edges to the final node of G_0 are treated in *conjunction*: one derives that the prefix τ_1 of τ_0 lies in $[\![G_1]\!]$ *and* in $[\![G_2]\!]$. We next illustrate that the notion of iDFGs also provides a concept of *disjunction*. In the iDFG G_0' to the right, two of the three incoming edges to the final node are labeled with the same assertion, namely $\mathtt{x} > 0$. The two iDFGs corresponding to the two edges are G_1 and G_1', where G_1 is as before and G_1' is the iDFG whose final node is the new node (labeled with the action `x:=2`). Both, G_1 and G_1', have the postcondition $\mathtt{x} > 0$. They are treated in *disjunction*: in order to derive $\tau_0 \in [\![G_0']\!]$, one can derive that τ_1 lies in $[\![G_1]\!]$ *or* in $[\![G_1']\!]$. As a consequence, we can derive not only $\tau_0 \in [\![G_0']\!]$ (via $\tau_1 \in [\![G_1]\!]$) but also $\tau_0' \in [\![G_0']\!]$ where τ_0' = `x:=2.y++.z:=x+y` (via $\tau_1' \in [\![G_1']\!]$ where τ_1' = `x:=2.y++`).

The examples above illustrate how one can use an iDFG G to justify the correctness of a trace τ and derive $\tau \in [\![G]\!]$. The definition below generalizes the examples.

Definition 3.3 (Denotation of an iDFG, $[\![G]\!]$) We define when a trace τ lies in the denotation of an iDFG G, formally

$$\tau \in [\![G]\!]$$

by induction over the construction of the trace τ. The empty trace $\tau = \varepsilon$ lies in $[\![G]\!]$ iff $v_0 \in V_{\mathsf{final}}$. The trace $\tau = \tau'.a$ obtained by attaching the action a at the end of the trace τ' lies in the denotation of an iDFG G of the form

$$G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$$

if either:

- the postcondition of G is *stable* under the action a and the trace τ' lies in the denotation of G, i.e.,

$$\{\varphi_{\mathsf{post}}\}\ a\ \{\varphi_{\mathsf{post}}\} \text{ and } \tau' \in [\![G]\!]$$

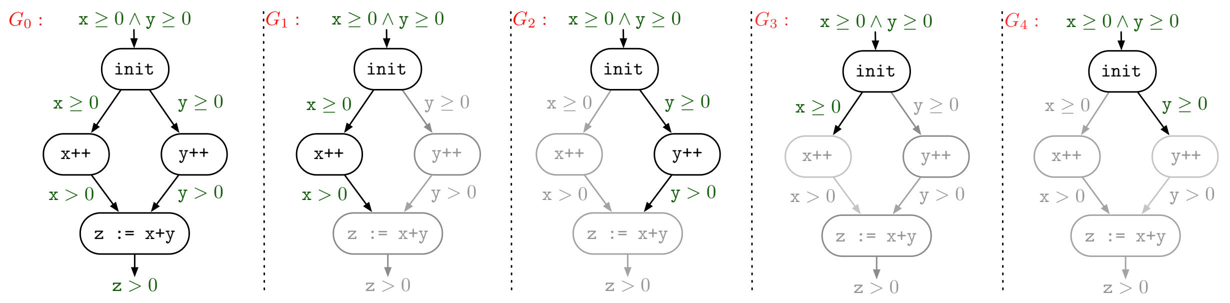


Figure 2. Example iDFGs used to illustrate Definition 3.3. The five iDFGs differ in the postcondition φ_{post} and the set of final vertices V_{final}. In each iDFG, V_{final} consists of the vertex with a *dangling* outgoing edge (the edge has no target or its target lies in the gray part of the graph), and φ_{post} is the assertion labeling that edge. For example, the final vertex of G_1 is the vertex labeled with the action `x++` and φ_{post} is $\mathtt{x} > 0$; the final node of G_3 is the initial vertex and φ_{post} is $\mathtt{x} \geq 0$.

or

- one of the final vertices $v_{\mathsf{f}} \in V_{\mathsf{final}}$ is labeled by the action a, and:
  - the assertions $\varphi_1, \ldots, \varphi_n$ are the labels of the incoming edges of the final vertex v_{f}, i.e.,
  $$\{\varphi_1, \ldots, \varphi_n\} = \{\psi \mid v \xrightarrow{\psi} v_{\mathsf{f}}\},$$
  - for each $i = 1, \ldots, n$, the trace τ' lies in the denotation of the iDFG G_i which we obtain from G by taking the assertion φ_i for the postcondition and the set of vertices V_i (defined below) for the set of final vertices, i.e.,
  $$G_i = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_i, v_0, V_i \rangle$$
  where V_i is the set of vertices that have an outgoing edge labeled with the assertion φ_i to the final vertex v_{f}, i.e.,
  $$V_i = \{v \mid v \xrightarrow{\varphi_i} v_{\mathsf{f}}\}. \quad \lrcorner$$

Remark 3.4 An alternative, equivalent definition of the denotation of an iDFG is based on fixpoints. Given the iDFG G as above, we define a monotone function

$$F : (V \to 2^{\mathsf{Actions}^*}) \to (V \to 2^{\mathsf{Actions}^*})$$

over the complete lattice $V \to 2^{\mathsf{Actions}^*}$ as follows:

$$F(L)(v_0) = \{\varepsilon\}$$

and for all $v \neq v_0$,

$$F(L)(v) = \bigcap_{\{\varphi \mid \exists u \in V. u \xrightarrow{\varphi} v\}} \bigcup_{\{u \mid u \xrightarrow{\varphi} v\}} L(u).Stable(\varphi)^*.\mathsf{act}(v)$$

where

$$Stable(\varphi) = \{a \in \mathsf{Actions} \mid \{\varphi\}\ a\ \{\varphi\}\}$$

We use $\widehat{L}$ to denote the least fixpoint of F. The denotation of G can be equivalently defined as below.

$$[\![G]\!] = \bigcup_{v \in V_{\mathsf{final}}} \widehat{L}(v).Stable(\varphi_{\mathsf{post}})^* \quad \lrcorner$$

The following observation, a direct consequence of Definition 3.3, relates the correctness of a trace with the denotation of an iDFG.

Remark 3.5 Let $G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$ be an iDFG. Then for any $\tau \in [\![G]\!]$, the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ holds. $\lrcorner$

Since we defined that the program $\mathcal{P}$ is correct wrt. the pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid for every program trace, the above observation gives immediately rise to a characterization of program correctness.

Theorem 3.6 (Program correctness via iDFG)**.** A program $\mathcal{P}$ is correct wrt. the precondition φ_{pre} and the postcondition φ_{post} if there exists an iDFG G with precondition φ_{pre} and postcondition φ_{post} such that every program trace of the program $\mathcal{P}$ lies in the denotation of the iDFG G:

$$\{\text{program traces of } \mathcal{P}\} \subseteq [\![G]\!].$$

We say that G is a proof for $\mathcal{P}$ if the above inclusion holds. In the next sections, we will investigate how one can algorithmically check the inclusion for a given program and a given iDFG, and how one can construct an iDFG.

4. Checking iDFGs

In order to show that a given iDFG G is a proof for a given program $\mathcal{P}$, we need to check the condition that every program trace of $\mathcal{P}$ lies in the denotation of G. We will reduce this check to an inclusion problem between two automata.

The set of program traces of the program $\mathcal{P}$ is the set of labelings of paths between the initial location ℓ_0 and a final location $\ell \in F$ in the control flow graph of the program $\mathcal{P}$ (as mentioned previously, not every such path corresponds to a possible execution of the program). Thus, the set of program traces is a *regular language* over the alphabet Actions, the set of actions. It is recognized by the program $\mathcal{P}$ viewed as an automaton. Its set of states is Loc, the set of program locations. Its transition are the edges $\ell_1 \xrightarrow{a} \ell_2$ (state ℓ_1 goes to state ℓ_2 upon reading letter a). The initial state is the initial location ℓ_0 and the set of final states is the set F of final locations. We use $\mathcal{L}(\mathcal{A})$ to denote the language recognized by the automaton $\mathcal{A}$. We thus have

$$\{\text{program traces of } \mathcal{P}\} = \mathcal{L}(\mathcal{P}).$$

We will next transform an iDFG G into an *alternating finite automaton*. Alternating finite automata [5, 6] or AFA are a generalization of NFA. This generalization is easily understood if one views the generalization from DFA to NFA as follows. The value of the deterministic successor function is generalized from a single successor state to a disjunction of states, and the initial state is generalized to be a disjunction of states. The NFA associates with any word a Boolean expression (in fact, a disjunction of states) obtained by repeatedly rewriting the initial formula using the successor function, and accepts if that Boolean expression evaluates to *true* under the assignment that sends final states to *true* and non-final states to *false*. The generalization to AFA is then very natural: one allows a general Boolean expression over states for the image of the successor function and for the initial condition.

From data flow graph G to alternating finite automaton $\mathcal{A}_G$. Given the data flow graph $G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}}\rangle$, we define the AFA $\mathcal{A}_G = \langle \Sigma, Q, \delta, q_0, Q_{\mathsf{final}}\rangle$ where

- $\Sigma = \mathsf{Actions}$
- $Q = \{(\psi, v) \mid \exists v' \in V.\ v \xrightarrow{\psi} v'\} \cup \{(\varphi_{\mathsf{post}}, v_{\mathsf{f}}) \mid v_{\mathsf{f}} \in V_{\mathsf{final}}\}$
- $\delta((\psi, v), a) = \mathit{skip}((\psi, v), a) \vee \mathit{step}((\psi, v), a)$ where

$$\mathit{skip}((\psi, v), a) = \begin{cases} (\psi, v) & \text{if } \{\psi\}\ a\ \{\psi\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{step}((\psi, v), a) = \begin{cases} \bigwedge_{\{\varphi \mid v' \xrightarrow{\varphi} v\}} \bigvee_{\{v' \mid v' \xrightarrow{\varphi} v\}} (\varphi, v') & \text{if } \mathsf{act}(v) = a \\ \mathit{false} & \text{otherwise} \end{cases}$$

- $q_0 = \bigvee_{v_{\mathsf{f}} \in V_{\mathsf{final}}} (\varphi_{\mathsf{post}}, v_{\mathsf{f}})$
- $Q_{\mathsf{final}} = \{(\psi, v) \in Q \mid v = v_0\}$.

The observation below is immediate by the fact that the construction of $\mathcal{A}_G$ mimics the definition of the denotation of G. We write L^{rev} for the reversal of the language L.

Remark 4.1 The set of traces denoted by G is the reverse of the language recognized by $\mathcal{A}_G$, i.e., $[\![G]\!] = \mathcal{L}(\mathcal{A}_G)^{\mathit{rev}}$. ⌟

We can now reformulate Theorem 3.6 from the previous section. The check whether a given language is included in the reversal of the language of a given AFA can bypass the construction of the reversal of the AFA (because we can reverse the program traces instead).

Theorem 4.2 (Checking program correctness via an iDFG)**.** A program $\mathcal{P}$ is correct wrt. the precondition φ_{pre} and the postcondition φ_{post} if there exists an iDFG G with precondition φ_{pre} and postcondition φ_{post} such that the language inclusion

$$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A}_G)^{\mathit{rev}}$$

holds for the program automaton of $\mathcal{P}$ and the alternating finite automaton $\mathcal{A}_G$ derived from the iDFG G.

The theorem above expresses that we can shift the burden of having to deal with the exponential size of the control flow graph of a concurrent program $\mathcal{P}$ defined by the parallel composition of N threads to a combinatorial problem over finite graphs (and thus to a place where *it hurts less*, in comparison to a setting where one has to deal with data). We can view the problem as a finite-state model checking problem since it is equivalent to the satisfaction of a linear time property (a regular safety property defined by the AFA) by a finite-state model (the control flow graph of $\mathcal{P}$). It is a classical result that this problem can be solve in polynomial space.

Theorem 4.3 (PSPACE)**.** The problem of checking whether an iDFG G is a proof for a program $\mathcal{P}$ given as the parallel composition of N threads, (i.e., checking whether all program traces of $\mathcal{P}$ are included in the denotation of G) is PSPACE (in the number N of threads).

5. Succinctness of iDFGs

In this section, we justify our claim that iDFGs are succinct proof objects. We introduce *localized proofs* in order to define a measure of the difficulty of proving that given program $\mathcal{P}$ satisfies a specification $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$; we call this measure the *data complexity* of $\mathcal{P}$ (with respect to $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$). We prove that if there exists a "small" proof of correctness for a program (i.e., if the program has low data complexity), then there exists a small iDFG proof.

It is easy to construct an iDFG proof from a control flow graph with a Floyd annotation by replacing edges with vertices, and vertices with edges. This construction proves the completeness of our proof rule (relative to the completeness of Floyd/Hoare proofs), but it leaves something to be desired: for a concurrent program $\mathcal{P}$, the iDFG resulting from this construction is of the same size as the control flow graph for $\mathcal{P}$, which is exponential in the number of threads in $\mathcal{P}$. In the rest of this section, we develop a characterization of when it is possible to prove that smaller iDFG proofs exist.

Consider the increment example from Section 2. There is an intuitive sense in which it is easy to prove that this program is correct because, although the size of the control flow graph for increment is exponential in the number of threads, the number of distinct assertions in the labeling of the control flow graph as required by a Floyd/Hoare proof is linear (the assertions appearing in the Floyd annotation are the ones of the form $x = i$, for $i \in [1, N]$). Following this example, a first attempt at defining a measure of the inherent difficulty of proving a program correct w.r.t. a specification may be "the minimal number of assertions in a Floyd/Hoare proof of the property." This definition fails a natural requirement for such a measure, which is that the if the threads have no shared data (disjoint parallelism), then their parallel composition should have a small proof. It is for this reason that we introduce the concept of *localized proofs*. We first explain the intuition behind localized proofs, and then give their formal definition.

Localized proofs are a way of exposing "how compositional" a Floyd/Hoare proof is, while avoiding syntactic issues common to practical compositional proof methods such as auxiliary variables in Owicki-Gries or Rely/Guarantee. A localized proof essentially splits each assertion in the Floyd annotation of the control flow graph of $\mathcal{P}$ into a global assertion and a set of local assertions (local for a thread). The total number of distinct assertions (both global and local) can then be seen as a measure of the inherent complexity of the proof. The intuition behind this view comes from the idea of transforming a succinct proof (in some other framework, e.g., Owicki-Gries) to a Floyd/Hoare proof for the product program. In performing this transformation, assertions from the succinct proof will be replicated several times: an assertion attached to the control location ℓ of some thread $\mathcal{P}_i$ will appear as a conjunct in each assertion attached to a location of the control flow graph of $\mathcal{P}$ where thread $\mathcal{P}_i$ is at ℓ, i.e., a location of the form $(\ell_1, \ldots, \ell_i, \ldots, \ell_N)$ where $\ell_i = \ell$.

Definition 5.1 (Localized proof) Given a concurrent program $\mathcal{P}$ defined by the parallel composition of N threads $\mathcal{P}_1, \ldots, \mathcal{P}_N$ and a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, a *localized proof*

$$\langle \iota_0, \iota_1, \ldots, \iota_N\rangle$$

is as a tuple of annotations (i.e., mappings from locations of the control flow graph of $\mathcal{P}$ to assertions) where each assertion $\iota_0(\ell)$ ("the global assertion at ℓ") may refer to global and local variables but each assertion $\iota_i(\ell)$ may refer only to the local variables of thread $\mathcal{P}_i$.

1. If the edge (ℓ, a, ℓ') labeled by the action a from the location ℓ to the location ℓ' in the control flow graph of $\mathcal{P}$ is induced by the edge (ℓ_k, a, ℓ'_k) of the thread $\mathcal{P}_k$ (i.e., if $\ell = (\ell_1, ..., \ell_N)$ and $\ell' = (\ell'_1, ..., \ell'_N)$ then $\ell_i = \ell'_i$ for $i \neq k$), then the Hoare triple
$$\{\iota_0(\ell) \wedge \iota_k(\ell)\}\ a\ \{\iota_0(\ell') \wedge \iota_k(\ell')\}$$
is valid and for all $i > 0$ different from k, the local assertion is the same for location ℓ and location ℓ'; i.e., $\iota_i(\ell) = \iota_i(\ell')$ for $i \neq k$.

2. The assertions at the initial location are entailed by the precondition; i.e., $\varphi_{\mathsf{pre}} \models \iota_i(\ell_0)$ for $i = 0, 1, \dots, N$
3. Every final location is annotated with the same assertions and these assertions entail the postcondition; i.e., there exists $\varphi_0, \varphi_1, \dots, \varphi_N \in \Phi$ such that for all final locations ℓ and all i, $\iota_i(\ell) = \varphi_i$, and also $\varphi_0 \wedge \varphi_1 \wedge \cdots \wedge \varphi_N \models \varphi_{\mathsf{post}}$. ⌟

A localized proof can be viewed as a particular presentation of the Floyd/Hoare proof that labels each location ℓ with the conjunction of the global assertion and all local assertions for location ℓ.

In a degenerate case of a localized proof, the local annotations $\iota_1, \cdots, \iota_N$ are all trivial; i.e., $\iota_i(\ell) = \mathit{true}$ for each location ℓ of $\mathcal{P}$ and for $i = 1, \dots, N$. In this case, the whole assertion for location ℓ in a Floyd/Hoare proof is put into the global assertion $\iota_0(\ell)$ of the localized proof.

The size of a localized proof $\iota = \langle \iota_0, \iota_1, \dots, \iota_N \rangle$ is the number of distinct assertions appearing in ι. Formally,

$$\mathsf{size}(\iota) = \sum_{i \in [1,N]} |\mathsf{Actions}_i| \cdot |\mathrm{rng}(\iota_0)| \cdot |\mathrm{rng}(\iota_i)|$$

where $\mathrm{rng}(\iota)$ denotes the range of the annotation ι, and $\mathsf{Actions}_i$ is the set of actions in the i-th thread $\mathcal{P}_i$.

Example 5.2 Consider the following simple example program, consisting of N threads, which each increment a local variable $\mathtt{t_i}$ before storing that value in a global variable x:

Thread 1: $\mathtt{t_1}$++; x:=$\mathtt{t_1}$

⋮

Thread N: $\mathtt{t_N}$++; x:=$\mathtt{t_N}$

In the Hoare proof that this program satisfies the specification $\varphi_{\mathsf{pre}} : \mathtt{x} = \mathtt{t_1} = \ldots = \mathtt{t_N} = 0 / \varphi_{\mathsf{post}} : \mathtt{x} = 1$, a distinct assertion is required for each combination of values for the $\mathtt{t_i}$ and x variables (except the combination where x = 1 and each $\mathtt{t_i}$ = 0). The total number of such assertions is $2^{N+1}-1$. In the *localized* Hoare proof, ι_i assigns $\mathtt{t_i}$ = 1 to every location after thread i executes $\mathtt{t_i}$++ and $\mathtt{t_i}$ = 0 to every other location. The global annotation ι_0 assigns x = 1 to every location where some thread has executed x:=$\mathtt{t_i}$ and x = 0 to every other location. The size of this localized proof is $8N$. ⌟

Definition 5.3 (Data complexity) Given a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, the *data complexity* of a program $\mathcal{P}$ is the minimum size of a localized proof that $\mathcal{P}$ satisfies the specification $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$. ⌟

We can now state the main result of this section.

Theorem 5.4 (Size of iDFG proofs)**.** Given the pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, if a proof for the program $\mathcal{P}$ exists, then there exists a proof for $\mathcal{P}$ in the form of an iDFG whose size is polynomial in the data complexity of $\mathcal{P}$.

Proof. Let $\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}$ be a specification, $\mathcal{P} = \langle \mathsf{Loc}, \delta \rangle$ be a program obtained as the parallel composition of N threads $\mathcal{P}_1, \dots, \mathcal{P}_N$, and $\iota = \langle \iota_0, \iota_1, \dots, \iota_N \rangle$ be a localized proof of minimum size that $\mathcal{P}$ satisfies $\varphi_{\mathsf{pre}} / \varphi_{\mathsf{post}}$. We construct an iDFG

$$G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$$

that proves that $\mathcal{P}$ satisfies the specification $\varphi_{\mathsf{pre}} / \varphi_{\mathsf{post}}$ as follows:

First we define the set of vertices of G. There are three different classes of vertices: the *initial* vertex, *regular* vertices, and *final* vertices. As always, we use v_0 to denote the initial vertex. The regular vertices are defined as follows:

$$V_{\mathsf{regular}} = \{\langle a, \iota_0(\ell), \iota_i(\ell) \rangle \mid i \in [1,N], \exists \ell'. (\ell, a, \ell') \in \delta_i\},$$

where for any $i \in [1,N]$, δ_i denotes the set of edges due to thread i. Letting F denote the set of final locations of $\mathcal{P}$, the final vertices are defined

$$V_{\mathsf{final}} = \{\langle a, \iota_0(\ell), \iota_1(\ell), \dots, \iota_N(\ell) \rangle \mid (\ell, a, \ell') \in \delta, \ell' \in F\}.$$

The set of all vertices may then be defined as

$$V = \{v_0\} \cup V_{\mathsf{regular}} \cup V_{\mathsf{final}}.$$

Now we construct the edges of G. We introduce two auxiliary functions for this purpose: the function in maps each vertex v to a set of assertions that will label the incoming edges to v, and the function pre maps each vertex to an assertion that can be thought of as its precondition. Formally,

$$\begin{aligned} \mathsf{in}(v_0) &= \emptyset & \mathsf{in}(\langle a, \varphi, \varphi_i \rangle) &= \{\varphi, \varphi_i\} \\ \mathsf{in}(\langle a, \varphi_0, \varphi_1, \dots, \varphi_N \rangle) &= \{\varphi_0, \varphi_1, \dots, \varphi_N\} \\ \mathsf{pre}(v_0) &= \varphi_{\mathsf{pre}} & \mathsf{pre}(\langle a, \varphi, \varphi_i \rangle) &= \varphi \wedge \varphi_i \\ \mathsf{pre}(\langle a, \varphi_0, \varphi_1, \dots, \varphi_N \rangle) &= \varphi_0 \wedge \varphi_1 \wedge \cdots \wedge \varphi_N \end{aligned}$$

The set of edges is the defined as

$$E = \{u \xrightarrow{\varphi} v \mid u \in V \setminus V_{\mathsf{final}}, v \in V, \varphi \in \mathsf{in}(v), \{\mathsf{pre}(u)\}\ \mathsf{act}(u)\ \{\varphi\} \text{ holds}\}.$$

This completes our definition of G. It is easy to check that G is well defined (that is, that the labeling of G is inductive). We now argue that the size of G is polynomial in the size of ι. The number of regular vertices in G is at most

$$\mathsf{size}(\iota) = \sum_{i \in [1,N]} |\mathsf{Actions}_i| \cdot |\mathrm{rng}(\iota_0)| \cdot |\mathrm{rng}(\iota_i)|.$$

This is also the maximum number of final vertices of G, since the definition of localized proofs guarantees that for any $a \in \mathsf{Actions}_i$ and any two final vertices labeled with a, $\langle a, \varphi_0, ..., \varphi_N \rangle, \langle a, \varphi_0', ..., \varphi_N' \rangle \in V_{\mathsf{final}}$, we must have $\varphi_j = \varphi_j'$ except when $j = 0$ or $j = i$. Thus, total number of vertices is at most $1 + 2 \cdot \mathsf{size}(\iota)$. Each pair of vertices may have at most two edges between them, so the total number of edges must be at most

$$2 \cdot |V|^2 \leq 2 \cdot (1 + 2 \cdot \mathsf{size}(\iota))^2,$$

which is polynomial in $\mathsf{size}(\iota)$.

It remains to show that every program trace τ of $\mathcal{P}$ belongs to $[\![G]\!]$. We prove the following result:

Lemma 5.5. For any path $\ell_0 a_0 \cdots a_{n-1} \ell_n$ in $\mathcal{P}$ (starting at the entry vertex ℓ_0), there exists $u_0, u_1, \dots, u_N \in V \setminus V_{\mathsf{final}}$ such that for all $i = 0, 1, \dots, N$,

$$a_0 \cdots a_{n-1} \in [\![G/(u_i, \iota_i(\ell_n))]\!]$$

where, for any vertex $v \in V$ and assertion $\varphi \in \Phi$,

$$G/(v, \varphi) = \langle V, E, \varphi_{\mathsf{pre}}, \varphi, v_0, \{v\} \rangle$$

is the iDFG obtained from G by making v the final vertex and φ the postcondition.

Before we prove this lemma, we show why it implies that every trace of $\mathcal{P}$ belongs to $[\![G]\!]$. Suppose τ is a trace of $\mathcal{P}$. Then there exists a path $\ell_0 a_0 \cdots a_{n-1} \ell_n a_n \ell_{n+1}$ such that $a_0 \cdots a_n = \tau$ and ℓ_{n+1} is a final location. By the lemma, there exists some $u_0, u_1, \dots, u_N$ such that $a_0 \cdots a_{n-1} \in [\![G/(u_i, \iota_i(\ell_n))]\!]$ for all $i = 0, 1, \dots, N$. Let $u_{\mathsf{f}} = \langle a_n, \iota_0(\ell_n), \iota_1(\ell_n), \dots, \iota_N(\ell_n) \rangle$. Then it follows from the construction of E that $u_i \xrightarrow{\iota_i(\ell_n)} u_{\mathsf{f}} \in E$ for all $i = 0, 1, ..., N$, and thus that $\tau = a_0 \cdots a_{n-1} a_n \in [\![G]\!]$.

Now we prove the lemma by induction on paths. The base case is trivial: we may choose $u_0 = u_1 = \cdots = u_N$ to be the initial vertex v_0.

For the induction step, suppose that $\ell_0 a_0 \cdots a_{n-1} \ell_n a_n \ell_{n+1}$ is a path in $\mathcal{P}$ and that there exists $u'_0, u'_1, \cdots, u'_N$ such that $a_0 \cdots a_{n-1}$ belongs to $[\![G/(u'_i, \iota_i(\ell_n))]\!]$ for all $i \in [0, N]$. Let $k \in [1, N]$ be the thread which executes the last action a_n. We distinguish two cases:

For $i \neq k$, we may take $u_i = u'_i$. By the induction hypothesis, $a_0 \cdots a_{n-1} \in [\![G/(u'_i, \iota_i(\ell_n))]\!]$. By condition 1 of Definition 5.1, $\iota_i(\ell_n) = \iota_i(\ell_{n+1})$, so $a_0 \cdots a_{n-1} \in [\![G/(u'_i, \iota_i(\ell_{n+1}))]\!]$. Since $\iota_i(\ell_{n+1})$ is stable under the action a_n (since a_n is executed by thread $k \neq i$), it follows that $a_0 \cdots a_{n-1} a_n \in [\![G/(u'_i, \iota_i(\ell_{n+1}))]\!]$.

For $i = k$, we make take $u_i = \langle a_n, \iota_0(\ell_n), \iota_k(\ell_n) \rangle$. First, we note that we must have (by the definition of E)

$$u'_0 \xrightarrow{\iota_0(\ell_n)} u_i \in E \text{ and } u'_k \xrightarrow{\iota_k(\ell_n)} u_i \in E$$

Since (by the induction hypothesis), $a_0 \cdots a_{n-1}$ is in both $[\![G/(u'_0, \iota_0(\ell_n))]\!]$ and $[\![G/(u'_i, \iota_i(\ell_n))]\!]$, we have the desired result: $a_0 \cdots a_{n-1} a_n \in [\![G/(u_i, \iota_k(\ell_{n+1}))]\!]$. The same argument shows that u_i is also an adequate choice for u_0 (i.e., $a_0 \cdots a_{n-1} a_n \in [\![G/(u_i, \iota_0(\ell_{n+1}))]\!]$).

□

The proof shows that the number of nodes of G is linear in the data complexity. In the extreme case of *disjoint parallelism* (a concurrent program composed of N threads with no shared data) there exists a localized proof without global assertion (formally, each global assertion is *true*.) In this case, the iDFG constructed in the proof of Theorem 5.4 is essentially the collection of local proofs as in an Owicki-Gries style proof (the local proofs are connected via edges from the initial vertex v_0). Its vertices correspond to pairs consisting of an action and a local assertion. Hence, the number of its vertices is linear in N.

A special case of this theorem is a concurrent program in a parametrized number N of threads that has a localized proof where the number of global assertions (i.e., the size of $rng(\iota_0)$) grows linearly in N, and the number of actions and local assertions (i.e., the size of $rng(\iota_i)$) is constant. The increment example from Section 2 falls into this case. The iDFG constructed in the proof of Theorem 5.4 has $O(N)$ vertices (as does the one we constructed manually in Section 2). In Section 2, we also hinted at the iDFG (with $O(N^2)$ vertices) for the case where we rename apart the action `x++` in each thread. This case illustrates the motivation to use the number of actions as a parameter for the data complexity (which becomes quadratic in N in this case).

In the case of the Ticket algorithm, not only the number of global assertions, but also the number of local assertions grows linearly in N. Therefore, the data complexity is quadratic in N. Indeed, the iDFG we will construct in Section 6 has $O(N^2)$ vertices.

6. Verification Algorithm

In this section, we develop an iDFG-based algorithm for verifying concurrent programs. Given a program $\mathcal{P}$ and a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, the goal of this algorithm is to either:

- construct a proof for $\mathcal{P}$ in the form of an iDFG G, or
- return a counterexample, i.e., a program trace τ such that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ does not hold.

We use Figure 3 to explain the basic idea behind the algorithm. The algorithm starts by picking a program trace τ. If the trace does not satisfy the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$, then the program is not correct with respect to the given specification. If it does, then τ is abstracted into an iDFG G_τ.

The algorithm maintains an iDFG G that represents a set of traces proven correct (initially, this set is empty). The iDFG G

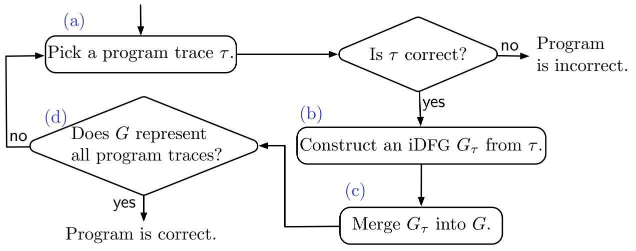

(a)
Pick a program trace τ.
Is τ correct?
no
Program is incorrect.
yes
(b)
Construct an iDFG G_τ from τ.
(c)
Merge G_τ into G.
(d)
Does G represent all program traces?
no
yes
Program is correct.

Figure 3. Verification algorithm based on inductive data flow graphs (iDFGs). Initially G is empty.

is updated by merging it with G_τ. If the resulting iDFG contains every program trace, then G is a proof for the program. If not, then the algorithm keeps generating traces and updating G until either a proof or a counterexample is found.

In component (d), we check whether an iDFG G is a proof for $\mathcal{P}$, which means checking the inclusion $\mathcal{L}(\mathcal{P}) \subseteq [\![G]\!]$; we discussed how to accomplish this in Section 4. The failure of the inclusion check means that there is a trace τ in the difference between the two sets, $\mathcal{L}(\mathcal{P}) \setminus [\![G]\!]$. By choosing such a trace in component (a), we ensure progress of the algorithm (i.e., after each iteration, G denotes a larger set of correct traces). In what follows, we discuss the remaining components (b) and (c). We will explain how an iDFG G_τ is constructed from a trace τ (Section 6.1) and how iDFG G is updated by merging it with G_τ (Section 6.2). Finally, we present the full algorithm in Section 6.3.

6.1 Constructing an iDFG from a trace

We will present an algorithm that, given a trace τ and a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ such that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ holds, constructs an iDFG G_τ (with the same pre/postcondition pair) whose denotation contains τ. By Remark 3.5, G_τ is a proof for the correctness of τ, i.e., for the validity of the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$.

This algorithm uses an auxiliary procedure $\mathsf{Interpolate}$. Given a trace τ which ends in the action a, i.e., $\tau = \tau'.a$ and which is correct for a given pair of assertions $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, i.e.,

$$\{\varphi_{\mathsf{pre}}\}\ \tau'.a\ \{\varphi_{\mathsf{post}}\},$$

the call of the auxiliary procedure $\mathsf{Interpolate}(\varphi_{\mathsf{pre}}, \tau, a, \varphi_{\mathsf{post}})$ returns an intermediate assertion ψ such that

$$\{\varphi_{\mathsf{pre}}\}\ \tau'\ \{\psi\} \text{ and } \{\psi\}\ a\ \{\varphi_{\mathsf{post}}\}.$$

The auxiliary procedure $\mathsf{Interpolate}$ is always applied to a finite program trace, which is effectively a very simple sequential program. This means that it can be implemented by one form or another of static analysis applied backwards to this sequential program (that consists just of the trace τ). One can leverage the power of existing static analysis methods such as apply a backwards transformer in some abstract domain, Craig interpolation, weakest precondition computation, among others to construct iDFGs [8].

Intuitively, the procedure $\mathsf{construct\text{-}idfg}(\tau, \varphi, \varphi')$ takes the trace τ and detects what actions in τ and what ordering constraints embodied in τ are irrelevant (for attaining the postcondition φ'). This is accomplished by leveraging the $\mathsf{Interpolate}$ procedure as follows:

- If $\mathsf{Interpolate}(\varphi, \tau', a, \varphi')$ returns the postcondition φ' (and thus, the postcondition φ' is stable wrt. the action a (i.e., $\{\varphi'\}\ a\ \{\varphi'\}$), and therefore a is irrelevant.
- Otherwise, if $\mathsf{Interpolate}(\varphi, \tau', a, \varphi')$ returns a proper conjunction $\psi_1 \wedge \cdots \wedge \psi_k$, then each assertion $\psi_1, \ldots$, or ψ_k can be

Algorithm construct-idfg$(\tau, \varphi, \varphi')$

```
Input: trace $\tau$ and a pair of assertions $(\varphi, \varphi')$ such that $\{\varphi\}\ \tau\ \{\varphi'\}$
Output: proof for $\tau$ in the form of an iDFG $G_\tau$
    (i.e., $G_\tau$ has the pre/postcondition pair $(\varphi, \varphi')$ and $\tau \in [\![G_\tau]\!]$)
  if $\tau = \varepsilon$ then
    return $G_\tau = \langle \{v_0\}, \emptyset, \varphi, \varphi', v_0, \{v_0\}\rangle$
  else
    $\tau = \tau'.a$ for some trace $\tau'$ and action $a$
    $\psi \leftarrow \mathsf{Interpolate}(\varphi, \tau', a, \varphi')$
    if $\psi = \varphi'$ then
      return construct-idfg$(\tau', \varphi, \varphi')$
    else
      $\psi = \psi_1 \wedge \cdots \wedge \psi_k$
      $v \leftarrow$ fresh vertex with label $a$ (i.e., $\mathsf{act}(v) = a$)
      parallel for $i = 1, \ldots, k$ do
        $\langle V_i, E_i, \varphi, \psi_i, v_0, V^i_{\mathsf{final}}\rangle \leftarrow$ construct-idfg$(\tau', \varphi, \psi_i)$
      end parallel for
      $V \leftarrow \{v\} \cup \bigcup_{i=1,\ldots,k} V_i$
      $E \leftarrow \bigcup_{i=1,\ldots,k}(E_i \cup \{u \xrightarrow{\psi_i} v \mid u \in V^i_{\mathsf{final}}\})$
      return $G_\tau = \langle V, E, \varphi, \varphi', v_0, \{v\}\rangle$
    end if
  end if
```

attained as postcondition for the prefix trace τ' *in parallel* (as opposed to in sequential order).

One possibility to maximize parallelism is to have Interpolate return formulae in conjunctive normal form. The particular strategy to break the intermediate assertions into conjuncts does not matter for the correctness of the overall algorithm.

Note that construct-idfg produces acyclic iDFGs without disjunctions (no vertex has multiple incoming edges labeled with the same assertion). Disjunctions and cycles will be produced by the *merge* procedure that we will discuss in Section 6.2.

Example 6.1 We will demonstrate the algorithm by applying it to Trace 1 of Bakery, pictured in Figure 1(a). The precondition is $\texttt{PreC} = \texttt{n1} = \texttt{n2} = 0 \wedge \neg\texttt{e1} \wedge \neg\texttt{e2}$ and the postcondition is $\texttt{false}$. We use τ to denote Trace 1 and τ_1 to denote the prefix of τ obtained by removing last action b_6.

For this example, we use the Craig interpolation feature of MathSAT5 [15] to implement Interpolate. The first call to Interpolate in construct-idfg yields the following interpolant:

$$\mathsf{Interpolate}(\texttt{PreC}, \tau_1, b_6, \texttt{false}) = \texttt{n1} > 0 \wedge \texttt{n1} < \texttt{n2}$$

Intuitively, the order in which the two assertions $\texttt{n1} > 0$ and $\texttt{n1} < \texttt{n2}$ are enforced is irrelevant, as long as both are enforced before b_6 is executed. This is reflected by how the execution of construct-idfg proceeds, namely by calling

$$\text{construct-idfg}(\tau_1, \texttt{PreC}, \texttt{n1} > 0)$$

and

$$\text{construct-idfg}(\tau_1, \texttt{PreC}, \texttt{n1} < \texttt{n2})$$

in parallel. The two calls correspond, respectively, to the left and the right branch of the iDFG (c) in Figure 1.

In order to illustrate how construct-idfg suppresses irrelevant actions, we will continue and follow the execution of the call

$$\text{construct-idfg}(\tau_1, \texttt{PreC}, \texttt{n1} > 0).$$

We use τ_2 to denote the prefix of τ_1 obtained by removing last action, which is `[¬e1]`. The action `[¬e1]` is irrelevant in the sense that it is not needed to enforce the postcondition $\texttt{n1} > 0$. This is established by construct-idfg when the next interpolant along this branch is computed:

$$\mathsf{Interpolate}(\texttt{PreC}, \tau_2, [\neg\texttt{e1}], \texttt{n1} > 0) = \texttt{n1} > 0$$

Since the computed interpolant is the same as the postcondition, construct-idfg enters the **then** branch of the conditional and proceeds to the call

$$\text{construct-idfg}(\tau_2, \texttt{PreC}, \texttt{n1} > 0).$$

The algorithm continues to suppress irrelevant actions in this way, going backwards along τ_2 until it comes to the first action under which $\texttt{n1} > 0$ is not stable, which is a_3 : `n1 := tmp1 + 1`. This action becomes the next vertex along the `n1>0` branch of the iDFG in Figure 1(a). ⌟

The essential property of construct-idfg is that it constructs an iDFG proof of the correctness of a trace with respect to a given pre/postcondition. This is expressed in the following lemma.

Lemma 6.2. Let τ be a trace that is correct with respect to the pre/postconditions φ/φ', and let $G_\tau = \text{construct-idfg}(\tau, \varphi, \varphi')$. Then $\tau \in [\![G_\tau]\!]$.

6.2 Merging iDFGs

Our merge operator $G_1 \mathbin{/\!\!\!\wedge\!\!\!\backslash} G_2$ can be thought of as a three step process: in the first step, we construct the disjoint union of G_1 and G_2; in the second step, *completion*, we saturate this iDFG by adding edges that do not violate the inductiveness property for iDFGs; in the third step, *reduction*, we collapse "equivalent" vertices.

We begin with a declarative definition for what it means for an iDFG to be complete (i.e., edge-saturated).

Definition 6.3 (Complete) An iDFG $G = \langle V, E, \varphi, \varphi', v_0, V_{\mathsf{final}}\rangle$ is *complete* if:

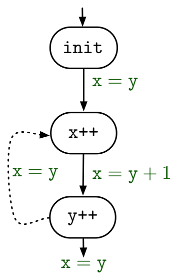
- For any $v \in V$ with such that $\{\mathsf{pre}(v)\}\ \mathsf{act}(v)\ \{\varphi'\}$ holds, $v \in V_{\mathsf{final}}$
- For any $u, v \in V$, and any φ such that v has an incoming edge labeled φ and $\{\mathsf{pre}(u)\}\ \mathsf{act}(u)\ \{\varphi\}$ holds, $u \xrightarrow{\varphi} v \in E$

where

$$\mathsf{pre}(v) = \begin{cases} \varphi & \text{if } v = v_0 \\ \bigwedge_{u \xrightarrow{\varphi} v \in E} \varphi & \text{otherwise} \end{cases}$$

⌟

Completion plays an essential role in the construction of iDFG proofs for programs. It is essential for producing proofs of programs that contain loops. Note that, as we mentioned in Section 6.1, the construct-idfg procedure does not introduce any disjunctions. Disjunctions are essential for capturing program behaviour produced by loops. The completion process can introduce these necessary disjunctions.



For example consider the following simple program and its specification: $\{$`x = y`$\}$ `while(*): x++; y++` $\{$`x = y`$\}$ The iDFG pictured to the right proves that this program satisfies the specification. The subgraph consisting only of the solid arrows can be generated by construct-idfg from a sample trace of the program `x++.y++`. The remaining dotted edge is added to the iDFG by the completion procedure (since `x++` has an incoming edge labeled $x = y$ and the Hoare triple $\{\mathsf{pre}(\texttt{y++})\}\ \texttt{y++}\ \{x = y\}$ holds). This generalizes the iDFG so that it proves the correctness of *any number* of iterations of the loop rather than a single iteration.

The following is a declarative definition for what it means for an iDFG to be reduced (i.e., vertex-minimal)

Definition 6.4 (Reduced) An iDFG $G = \langle V, E, \varphi, \varphi', v_0, V_{\mathsf{final}} \rangle$ is *reduced* if there exist no distinct $u, v \in V$ such that $\mathsf{act}(u) = \mathsf{act}(v)$ and the set of assertions labeling the incoming edges to u and v are the same (i.e., $\{\varphi \mid \exists w. w \xrightarrow{\varphi} v \in E\} = \{\varphi \mid \exists w. w \xrightarrow{\varphi} u \in E\}$). ⌟

There is an algorithm that, given an arbitrary iDFG G, constructs a reduced complete iDFG $rc(G)$ such that $[\![rc(G)]\!]$ contains $[\![G]\!]$. We call $rc(G)$ the *reduced completion of* G. This is stated formally in the following proposition:

Proposition 6.5. For every iDFG $G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$, there is a reduced, complete iDFG $rc(G)$ that can be computed from G in $O(|V| \cdot |E|)$ time and such that the precondition of $rc(G)$ is φ_{pre}, the postcondition of $rc(G)$ is φ_{post}, and $[\![G]\!] \subseteq [\![rc(G)]\!]$.

Proof. For any vertex v, we define

$$\mathsf{in}(v) = \{\varphi \mid \exists u. u \xrightarrow{\varphi} v \in G\}$$

We define $rc(G) = \langle V', E', \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V'_{\mathsf{final}} \rangle$, where

$$\begin{aligned} V' &= \{v_0\} \cup \{\langle \mathsf{act}(v), \mathsf{in}(v) \rangle \mid v \in V \setminus \{v_0\}\} \\ E' &= \{\langle a, \Psi \rangle \xrightarrow{\varphi} \langle a', \Psi' \rangle \mid \varphi \in \Psi' \wedge \{\bigwedge \Psi\}\ a\ \{\varphi\}\} \\ &\quad \cup \{v_0 \xrightarrow{\varphi} \langle a', \Psi' \rangle \mid \varphi \in \Psi' \wedge \{\varphi_{\mathsf{pre}}\}\ a\ \{\varphi\}\} \end{aligned}$$

and

$$V'_{\mathsf{final}} = \begin{cases} \{v_0\} \cup \{\langle \mathsf{act}(v), \mathsf{in}(v) \rangle \mid v \in V_{\mathsf{final}}\} & \text{if } \varphi_{\mathsf{pre}} \Rightarrow \varphi_{\mathsf{post}} \\ \{\langle \mathsf{act}(v), \mathsf{in}(v) \rangle \mid v \in V_{\mathsf{final}}\} & \text{otherwise} \end{cases}$$

It is easy to check that $rc(G)$ is well-defined. The machinery required to prove that $[\![G]\!] \subseteq [\![rc(G)]\!]$ is presented in Section 7. □

Finally, we describe our merge operation. The merge operator $G_1 \mathbin{⩕} G_2$ functions by forming the disjoint union of G_1 and G_2, and then taking the reduced completion of the resulting iDFG. Formally, we define merge as follows:

Definition 6.6 Given two iDFGs $G_1 = \langle V_1, E_1, \varphi, \varphi', v_0, V^1_{\mathsf{final}} \rangle$ and $G_2 = \langle V_2, E_2, \varphi, \varphi', v_0, V^2_{\mathsf{final}} \rangle$, their *merge*, $G_1 \mathbin{⩕} G_2$ is defined to be the iDFG

$$G_1 \mathbin{⩕} G_2 = rc(\langle V^{⩕}, E^{⩕}, \varphi, \varphi', v_0, V^{⩕}_{\mathsf{final}} \rangle)$$

where $V^{⩕} = V_1 \cup V_2$, $E^{⩕} = E_1 \cup E_2$, $V^{⩕}_{\mathsf{final}} = V^1_{\mathsf{final}} \cup V^2_{\mathsf{final}}$, and $\mathsf{act}(v)$ is defined in the obvious way. For simplicity, this definition assumes that the initial vertex of G is also the initial vertex of G' and that otherwise their vertices are disjoint. ⌟

Applying the rc operator in the merge plays an essential role in reducing the size of iDFG proofs. For example, the iDFG proof for the Ticket example from Section 2 would require $O(N!)$ vertices without applying the rc operator (see Example 6.9).

The progress of our algorithm depends on the following lemma concerning the merge operator:

Lemma 6.7. Let G_1 and G_2 be two iDFGs. We have

$$[\![G_1]\!] \cup [\![G_2]\!] \subseteq [\![G_1 \mathbin{⩕} G_2]\!].$$

An example of the merge operator appears in Figure 4: part (c) is the merge of parts (a) and (b). Notice that (c) proves that the trace

`m1=t++.m3=t++.[m3 <= s]`

is infeasible (starting from the precondition `s` $=$ `t` $= 0$), even though this fact is not proved by (a) or (b).

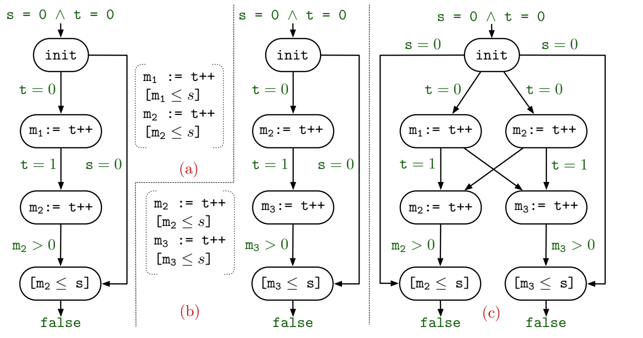


Figure 4. Intermediate iDFGs for the Ticket algorithm

6.3 Putting it all together

We are now ready to present a formal description of our verification algorithm. The last component of this algorithm that has not already been described is the initialization of G. If φ_{pre} implies φ_{post} (or equivalently, the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \varepsilon\ \{\varphi_{\mathsf{post}}\}$ holds), then G is initialized to an iDFG with a single vertex that is both initial and final (and therefore, $[\![G]\!] = \{\varepsilon\}$). Otherwise, G is initialized to an iDFG that has no final vertices (and therefore, $[\![G]\!] = \emptyset$).

Algorithm Verification algorithm

```
Input: Program $\mathcal{P}$, precondition $\varphi_{\mathsf{pre}}$, postcondition $\varphi_{\mathsf{post}}$
Output: Yes if $\mathcal{P}$ is correct w.r.t. the specification; No otherwise.
  if $\varphi_{\mathsf{pre}}$ entails $\varphi_{\mathsf{post}}$ then
    $G := \langle v_0, \emptyset, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, \{v_0\} \rangle$
  else
    $G := \langle v_0, \emptyset, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, \emptyset \rangle$
  end if
  while $\mathcal{L}(\mathcal{P}) \not\subseteq [\![G]\!]$ do
    Let $\tau \in \mathcal{L}(\mathcal{P}) \setminus [\![G]\!]$
    if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ holds then
      $G_\tau := \mathsf{construct\text{-}idfg}(\tau, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$
      $G := G ⩕ G_\tau$
    else
      return No (with counter-example $\tau$)
    end if
  end while
  return Yes
```

As a direct consequence of Lemmas 6.2 and 6.7, we can state the following progress property of the verification algorithm:

Proposition 6.8 (Progress)**.** If G_i and G_{i+1} are the iDFGs constructed by the verification algorithm in (respectively) the round i and round $(i + 1)$, then we have $[\![G_i]\!] \subset [\![G_{i+1}]\!]$ (the inclusion is strict).

We conclude this section with an example run of the verification algorithm on the Ticket algorithm mentioned in Section 2.

Example 6.9 We consider the 3-thread instance of the Ticket mutual exclusion algorithm, which runs three copies of the thread below in parallel (where i is substituted for the thread id).

Thread i

```
$\ell_{i,1}$:  m$_i$ := t++
$\ell_{i,2}$:  [m$_i$ ≤ $s$]
               // critical section
$\ell_{i,3}$:  s := s + 1
```

The first two rounds of the verification algorithm are depicted in Figure 4. Since the property of interest is mutual exclusion, we take the traces of this program to be the ones that end with (at least) two threads inside

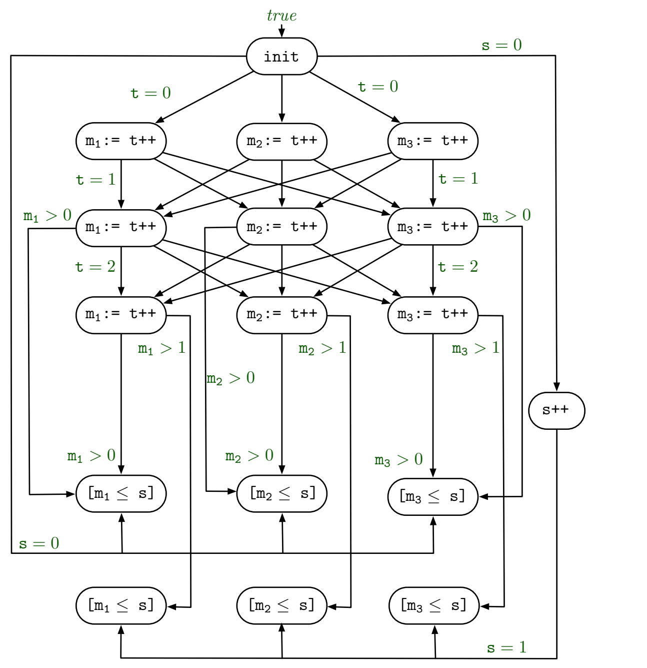


Figure 5. Correctness proof of the Ticket algorithm. Note that the post-conditions (all `false`) have been removed from the figure to keep it clean. In a complete version of this figure all 6 nodes with labels $[\mathtt{m_i} \le \mathtt{s}]$ have a dangling arrow with a label `false`.

their critical section, and prove that the program meets the specification with precondition $\mathtt{s} = 0 \wedge \mathtt{t} = 0$ and postcondition `false` (i.e., every trace violating mutual exclusion is infeasible).

The algorithm begins with the empty iDFG, $G_\emptyset$. The first program trace τ_1 that we select in $\mathcal{L}(\mathcal{P}) \setminus [\![G_\emptyset]\!]$ is depicted in Figure 4(a) along with an iDFG proof for τ_1, as constructed by construct-idfg. We call this iDFG G_1.

On the next iteration of the loop, we select the trace τ_2 depicted in part Figure 4(b), and merge its corresponding iDFG proof (called G_{τ_2}) with G_1 to form G_2 pictured in Figure 4(c). Since $[\![G_2]\!]$ does not cover every program trace (i.e., $\mathcal{L}(\mathcal{P}) \not\subseteq \mathcal{L}(G_2)$), the algorithm continues the loop. After four more iterations, the algorithm will terminate after all traces that violate mutual exclusion have been proved to be infeasible. The iDFG that is constructed by this algorithm is depicted in Figure 5.

The intuition for the iDFG in Figure 5 can be used to construct a proof for the Ticket mutual exclusion algorithm for any number of threads. For any such iDFG proof, the number of vertices is quadratic in the number of threads. ⌟

7. Properties of the Verification Algorithm

We will now investigate the properties of the verification algorithm. In particular, we prove that the algorithm is sound, that it is complete (under the assumption that the auxiliary procedure Interpolate returns the right assertions), and that its time and space complexity is polynomial in the data complexity of the input program and specification (again, under the assumption about Interpolate). We will also introduce some technical machinery required to prove these results. We begin with soundness (if the algorithm returns a result, then the result is correct), which is a direct consequence of Theorem 3.6.

Theorem 7.1 (Soundness)**.** If the verification algorithm returns Safe, $\mathcal{P}$ satisfies the given specification. If it returns Unsafe, the program does not satisfy the given specification.

We now move on to our completeness result. No algorithm exists that is complete in the strong sense (due to the halting problem). Instead, we show that the verification algorithm is complete in the sense that, if a there exists a safety proof for a program, then the algorithm will find one *under the assumption that Interpolate produces the right assertions*. We formalize this by assuming that Interpolate is a nondeterministic procedure that may return any valid interpolant, and showing that the following holds:

Theorem 7.2 (Completeness)**.** If a program has a Hoare safety proof, then there exists a run of the verification algorithm that terminates with a Safe result.

In fact, we can strengthen this completeness theorem and give bounds on the time and space required by the verification algorithm:

Theorem 7.3 (Complexity)**.** If a program $\mathcal{P}$ has a Hoare proof that it satisfies a specification $\varphi_{\mathsf{pre}}/\varphi_{\mathsf{post}}$, then there exists a run of the verification algorithm that terminates with an iDFG of size polynomial in the data complexity of $\mathcal{P}$ (w.r.t. $\varphi_{\mathsf{pre}}/\varphi_{\mathsf{post}}$). Moreover, the number of iterations of the main loop of the algorithm required to produce this proof is polynomial in the data complexity.

The proof of this theorem requires some technical machinery, which will be presented in the following. However, we present some early intuition on the proof for readers that wish to skip the technical details in the remainder of this section. We assume that we are given a proof presented as an iDFG G_0 (the size of which we may assume is polynomial in the data complexity, by Theorem 5.4). Given any trace τ, there is a substructure of G_0 that corresponds to τ (roughly speaking, this structure corresponds to the accepting run of τ through the AFA corresponding to G_0). If Interpolate returns the assertions corresponding to this structure, then construct-idfg$(\tau, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ produces this exact structure. This is a way of intuitively reasoning about why the appropriate assertions exist for Interpolate to produce. Since G_0 is finite, we can show that it is "covered" by finitely many such substructures, so that the algorithm will terminate in finite time with a proof that is "essentially the same" as G_0.

Proofs

We now present the technical machinery required to prove Theorem 7.3. The main concept we introduce here is *iDFG embeddings*, which is a structural relationship between iDFGs. We say that an iDFG G embeds into G' if G can be mapped onto a subgraph of G' in a way that is, in some sense, "tight". Formally, we define an iDFG embedding as follows:

Definition 7.4 (Embedding) Given iDFGs

$$G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}} \rangle$$

and

$$G' = \langle V', E', \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0', V_{\mathsf{final}}' \rangle$$

sharing the same precondition and postcondition, we say that a map $h : V \to V'$ is an *embedding* if the following hold:

- $\forall v \in V$, $\mathsf{act}(v) = \mathsf{act}(h(v))$ and
$$\{\varphi \mid \exists u.u \xrightarrow{\varphi} v \in E\} = \{\varphi \mid \exists u.u \xrightarrow{\varphi} h(v) \in E'\}$$
- $h(v_0) = v_0'$
- $\forall v \in V_{\mathsf{final}}, h(v) \in V_{\mathsf{final}}'$
- $\forall u \xrightarrow{\varphi} v \in E$, $h(u) \xrightarrow{\varphi} h(v) \in E'$

If such an embedding exists, we say that G *embeds* into G'. ⌟

The main property of interest concerning embeddings is the following lemma:

Lemma 7.5. If G and G' are iDFGs such that G embeds into G', then $[\![G]\!] \subseteq [\![G']\!]$.

The key idea of our completeness theorem is that we can use a given iDFG proof G_0 as an oracle to guide the interpolation procedure, so that $\mathsf{construct\text{-}idfg}(\tau, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ will yield an iDFG representing some substructure of the target proof G_0 (i.e., an iDFG that embeds into G_0). Formally,

Lemma 7.6. For any iDFG $G = \langle V, E, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}, v_0, V_{\mathsf{final}}\rangle$ and any trace $\tau \in [\![G]\!]$, there is a run of $\mathsf{construct\text{-}idfg}(\tau, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ that produces an iDFG G_τ such that G_τ embeds into G.

Our proof of Theorem 7.2 is based on being able to maintain a loop invariant that G embeds into some target proof G_0. In order to prove this invariant, we must show that if G and G' are embed into some target proof G_0, then $G \mathbin{/\!\!\!\wedge\!\!\!\backslash} G'$ also embeds into G_0. Formally, we have the following:

Lemma 7.7. For any iDFGs G and G' sharing the same precondition and postcondition, both G and G' embed into $G \mathbin{/\!\!\!\wedge\!\!\!\backslash} G'$. For any complete iDFG G_0 such that G and G' both embed into G_0, $G \mathbin{/\!\!\!\wedge\!\!\!\backslash} G'$ embeds into G_0[1].

We are now ready to provide a sketch of the proof of Theorem 7.3. In fact, we will prove a stronger (by Theorem 5.4) result: for any reduced, complete iDFG G_0 such that $\mathcal{L}(\mathcal{P}) \subseteq [\![G_0]\!]$, there exists a run of the verification algorithm that terminates with an iDFG G such that $\mathcal{L}(\mathcal{P}) \subseteq [\![G]\!]$ and such that G embeds into G_0 (since G and G_0 are reduced and complete, the fact that G embeds into G_0 implies that G is no larger than G_0). Moreover, regardless of how traces are chosen from $\mathcal{L}(\mathcal{P})\setminus[\![G]\!]$, the algorithm will terminate in at most $|V_0|$ iterations, where $|V_0|$ is the number of vertices in G_0.

The proof is by induction: we assume that at the start of the iteration of the loop, G embeds into G_0, and prove that there is an execution of the loop body such that the number of vertices in G increases and such that G still embeds into G_0 at the end of the loop.

Let $\tau \in \mathcal{L}(\mathcal{P}) \setminus [\![G]\!]$. Since G_0 is a safety proof, we must have $\tau \in \mathcal{L}(G_0)$. By Lemma 7.6, there exists a run of $\mathsf{construct\text{-}idfg}(\tau, \varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ that produces an iDFG G_τ that embeds into G_0. We let $G' = G \mathbin{/\!\!\!\wedge\!\!\!\backslash} G_\tau$ be the iDFG at the end of the loop. The invariant that G' embeds into G_0 is ensured by Lemma 7.7. The condition that G' has more vertices than G is ensured by the fact that G_τ embeds into G', but not G (since that would contradict $\tau \notin [\![G]\!]$).

8. Related Work

Concurrent Program Verification. As mentioned above, existing approaches to the algorithmic verification of concurrent programs, e.g. [10, 11, 16, 19], provide a different angle of attack at the same fundamental issue: the exponential space complexity (exponential in the number of threads). None of these approaches shifts the burden of the exponential growth of space towards a combinatorial problem (over finite graphs). The practical potential of these approaches is demonstrated on a selection of practical examples. None of the approaches investigates the question whether there are assumptions under which the space complexity is polynomial. The exponential cost of space is not an issue that occurs just in theory. We observe the exponential curve, e.g., when we run THREADER [16] and SLAB [11] on the ticket algorithm and the program Increment (see Section 2). These are two concurrent programs (parametrized in the number n of threads) where our approach comes with a formal guarantee for polynomial space consumption.

All the approaches mentioned above are based on abstraction. The construction of an iDFG from a trace can be viewed as an abstraction of the trace. It is interesting to compare the two concepts of abstraction. In the setting of the above-mentioned approaches, a *too coarse* abstraction introduces spurious errors. In our setting, it is desirable to abstract a trace aggressively (in order to obtain a succinct iDFG that denotes a large set of traces). This is because iDFG's are used to represent *correct* behaviours rather than program behaviours. More over-approximation leads to a larger set of correct traces.

Compositional Proof Rules. Our notion of inductive data flow graph bears similarities with Owicki-Gries style proof rules and other compositional proof rules for concurrent programs (e.g., the use of local assertions, of conjunction, of stability) [2]. Compositionality is often thought of as the only way to go for polynomial-sized proofs (in the number of threads). Our approach to algorithmic verification can be viewed as the automation of a *non-compositional* proof rule (the proof rule is obtained by reformulating the characterization of program correctness in Theorem 3.6). However, one can view sets of traces as *semantics-based* modules (as opposed to modules based on the syntactic construction of programs). These modules capture the intuitive notion of scenarios. The composition of modules is set union.

Thread-Modular Verification. The thread-modular approach to verification [14] achieves modularity at the expense of giving up completeness. In fact, the approach is complete exactly for the class of programs whose correctness can be proven in Owicki-Gries style proofs without auxiliary variables [9, 24]. Our approach, in contrast, is motivated by combining the goal of full (relative) completeness with space efficiency.

Data Flow Graphs. Variations of data flow graphs have a long history within the compilers community, both as a means for exposing parallelism in sequential code for parallelizing compilers [13, 22], and as a data structure for use in sparse dataflow analysis [20, 29]. Our use of DFGs is closer to this first line of work: $\mathsf{construct\text{-}idfg}$ can be seen as a procedure that exposes the parallelism in a single example trace. For a parallelizing compiler to be correct, parallelization must preserve the behaviour of the sequential code; the correctness of $\mathsf{construct\text{-}idfg}$ depends on the much weaker condition that a proof argument is preserved.

More recently, DFGs have been used for invariant generation in both concurrent [12] and sequential [27] settings. The work of [12] is particularly relevant: it uses data flow graphs to generate numerical invariants for parameterized concurrent programs. These invariants can be used to prove safety properties, but if the invariants are too weak to prove a property of interest, a false alarm is reported. Our verification algorithm produces no false alarms, and can provide a counter-example for properties that fail. Moreover, the inductive DFGs presented in this paper are capable of expressing proofs that cannot be represented using the (variable-labeled, rather than assertion-labeled) DFGs used in [12] (i.e., every DFG proof in [12] corresponds to an iDFG, but some iDFG's do not correspond to DFG proofs).

Trace Abstraction. The work in [17, 18] presents an automata-theoretic approach to the analysis and verification of sequential and recursive programs. The present paper continues this line of work and extends it to concurrent programs.

[1] This proposition establishes that $G \mathbin{/\!\!\!\wedge\!\!\!\backslash} G'$ is a coproduct in the category of reduced complete iDFGs, where the morphisms are embeddings.

9. Conclusion and Future Work

In this paper, we have introduced a new approach for the verification of concurrent programs. The approach succeeds in putting the well-established static analysis techniques for sequential programs to work, namely by assembling the output of the static analysis applied to interleaved executions in a space-efficient way. We formalize under what assumptions the space efficiency can be guaranteed. For fundamental reasons, we cannot avoid the exponential explosion in the number of threads but we can shift its burden to a combinatorial problem over finite graphs ("to a place where it hurts less").

The approach has an interesting practical potential. The focus in this paper was to introduce the approach and to investigate its formal foundation. There are several directions in which one can explore the practical potential of the approach. This should be the focus of future work.

The most critical operation is perhaps the construction of an iDFG from a given trace τ of the program (construct-idfg). In Section 6.1, we already hinted at several directions for practical optimizations. We need to develop and evaluate these optimizations for practical examples.

The operation construct-idfg depends on the static analysis that is applied to the trace (as a special case of a sequential program). The abstract values generated by the static analysis are used to extract single conjuncts for the labeling of an iDFG with inductive assertions. This raises an interesting topic for research on abstract domains for abstract interpretation. We are used to working with Moore families where abstract values are closed under 'conjunction' but not necessarily under 'disjunction'. As a consequence, the notion of 'conjunctive completion' has not yet been explored to the same degree as the notion of 'disjunctive completion'.

A related question is whether we can enhance interpolant generation in order to produce the 'right' conjuncts for the labeling of an iDFG with inductive assertions. A possible direction to explore here are *tree interpolants* [25] which were originally intended to generate summaries for recursive programs and which generate interpolants for graph-like structures.

References

[1] G. R. Andrews. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991. ISBN 978-0-8053-0086-4.

[2] K. R. Apt, F. S. de Boer, and E. R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 2009. ISBN 978-1-84882-744-8.

[3] 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. Springer Berlin / Heidelberg, 2008.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[5] J. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10 (1):19 – 35, 1980.

[6] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, Jan. 1981.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[9] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles*. les-Nancy, 1985.

[10] 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.

[11] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. Slab: a certifying model checker for infinite-state concurrent systems. In *TACAS*, pages 271–274, 2010.

[12] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, pages 297–308, 2012.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

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

[15] A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.

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

[17] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS*, pages 69–85, 2009.

[18] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482, 2010.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, pages 262–274, 2003.

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

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

[22] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, pages 207–218, 1981.

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

[24] A. Malkis. *Cartesian abstraction and verification of multithreaded programs*. PhD thesis, University of Freiburg, 2010.

[25] K. McMillan. Personal communication, 2012.

[26] A. Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP*, pages 398–418, 2011.

[27] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229–238, 2012.

[28] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19:279–285, May 1976.

[29] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *POPL*, pages 297–310, 1994.