Local Reasoning for Global Graph Properties

SIDDHARTH KRISHNA, New York University, USA
ALEXANDER J. SUMMERS, ETH Zurich, Switzerland
THOMAS WIES, New York University, USA

Separation logics are widely used for verifying programs that manipulate complex heap-based data structures. These logics build on so-called separation algebras, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. While heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character, such as reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such global graph properties remains notoriously difficult, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.

In this paper, we address the question: What separation algebra can be used to avoid proof arguments reverting back to tedious global reasoning in such cases? To this end, we consider a general class of global graph properties expressed as fixpoints of algebraic equations over graphs. We present mathematical foundations for reasoning about this class of properties, imposing minimal requirements on the underlying theory that allow us to define a suitable separation algebra. Building on this theory we develop a general proof technique for modular reasoning about global graph properties over program heaps, in a way which can be integrated with existing separation logics. We further devise a strategy for automating this technique using SMT-based verification tools. We have implemented this strategy on top of the verification tool Viper and applied it successfully to a variety of challenging benchmarks including 1) algorithms involving general graphs such as Dijkstra’s algorithm and a priority inheritance protocol, 2) inductive data structures such as linked lists and B trees, 3) overlaid data structures such as the Harris list and threaded trees, and 4) OO design patterns such as Composite and Subject/Observer. We are not aware of any single other approach that can handle these examples with the same degree of simplicity or automation.

1 INTRODUCTION

Separation logic (SL) [O’Hearn et al. 2001; Reynolds 2002] provides the basis of many successful verification tools that can verify programs manipulating complex data structures [Appel 2012; Calcagno et al. 2015; Jacobs et al. 2011; Müller et al. 2016]. This success is due to the logic’s support for reasoning modularly about modifications to heap-based data. For simple inductive data structures such as lists and trees, much of this reasoning can be automated [Berdine et al. 2004; Enea et al. 2017; Katelaan et al. 2019; Piskac et al. 2013]. However, these techniques often fail when data structures are less regular (e.g. multiple overlaid data structures) or provide multiple traversal patterns (e.g. threaded trees). Such idioms are prevalent in real-world implementations such as the fine-grained concurrent data structures found in operating systems and databases. Solutions to these problems have been proposed [Hobor and Villard 2013] but remain difficult to automate. For proofs of general graph algorithms, the situation is even more dire. Despite substantial improvements in the verification methodology for such algorithms [Raad et al. 2016; Sergey et al. 2015], significant parts of the proof argument still typically need to be carried out using non-modular reasoning. This paper presents a general technique for automated modular reasoning about graph properties that...
Fig. 1. Pseudocode of the PIP and a state of the protocol data structure. Round nodes represent processes and rectangular nodes resources. Nodes are marked with their current priorities curr_prio as well as the aggregate priority multiset prios. A node’s default priority is underlined and marked in bold blue.

Applies to a broad class of data structures and graph algorithms. In fact, for many of the examples that we consider in this work, no fully-modular proof had existed before.

As a motivating example, we consider an idealized priority inheritance protocol (PIP), which is a technique used in process scheduling [Sha et al. 1990]. The purpose of the protocol is to avoid (unbounded) priority inversion, i.e., a situation where a process is blocked from making progress by a lower priority process. The protocol maintains a bipartite graph with nodes representing processes and resources. An example graph is shown in Fig. 1. An edge from a process $p$ to a resource $r$ indicates that $p$ is waiting for $r$ to become available whereas an edge in the other direction means that $r$ is currently held by $p$. Every node has an associated default priority as well as a current priority, both of which are strictly positive integers. The current priority affects scheduling decisions. When a process attempts to acquire a resource currently held by another process, the graph is updated to avoid priority inversion. For example, when process $p_1$ with current priority 3 attempts to acquire the resource $r_1$ that is held by process $p_2$ of priority 2, then $p_1$’s higher priority is propagated to $p_2$ and, transitively, to any other process that $p_2$ is waiting for ($p_3$ in this case). As a result, all nodes on the created cycle will be updated to current priority 3. The protocol thus maintains the following invariant: the current priority of each node is the maximum of its default priority and the current priorities of all its predecessors. Priority propagation is implemented by the method update shown in Fig 1. The implementation represents graph edges by next pointers and handles both kinds of modifications to the graph: adding an edge (acquire) and removing an edge (release - code omitted). To recalculate the current priority of a node (line 10), each node maintains a multiset prios which contains the priorities of all its immediate predecessors as well as its own default priority.

Verifying that the PIP maintains its invariant using established separation logic (SL) techniques is challenging. In general, SL assertions describe resources and express the fact that the program has permission to access and manipulate these resources. We stick to the standard model of SL where resources are memory regions represented as partial heaps. Assertions describing larger regions are built from smaller ones using separating conjunction, $\phi_1 \uplus \phi_2$. Semantically, the $\uplus$ operator is tied to a notion of resource composition defined by an underlying separation algebra [Calcagno et al. 1]

1The algorithm can then detect the cycle to prevent a deadlock, but we ignore this here to keep the presentation simple.
We revisit the core algebra behind flows reasoning, and derive a different algebraic foundation. Work on unrestricted graphs are generally not supported by the original framework. Moreover, several significant improvements over that of Krishna et al. [2018] and eliminates its most stark limitations. First, we present a simplified and generalized meta theory of flows that makes the foundational flow framework suitable for a variety of challenging benchmarks (§6). These include 1) algorithms involving trees, and 4) OO design patterns such as Composite and Subject/Observer. We are not aware of any other approach that can handle these examples with the same degree of simplicity or automation.

Contributions. In this paper we answer the question: What separation algebra allows us to reason modularly about the effects of local changes on global properties of graphs? We consider a general class of global graph properties that can be expressed in terms of flows – functions from nodes of the graph to values. For the PIP, the flow maps each node to the multiset of its incoming priorities. For example, consider the PIP scenario depicted in Fig. 1. If \( \phi_1 \) describes the subgraph containing only node \( p_1 \), \( \phi_2 \) the remainder of the graph, and \( \phi'_1 \) the graph obtained from \( \phi_1 \) by adding the edge from \( p_1 \) to \( r_1 \), then the PIP invariant will no longer hold for the new composed graph described by \( \phi'_1 \ast \phi_2 \). On the other hand, if \( \phi_1 \) captures \( p_1 \) and the nodes reachable from \( r_1 \) (i.e., the set of nodes modified by \text{update}), \( \phi_2 \) the remainder of the graph, and we reestablish the PIP invariant locally in \( \phi_1 \) obtaining \( \phi'_1 \) (i.e., run \text{update} to completion), then \( \phi'_1 \ast \phi_2 \) will also globally satisfy the PIP invariant. The separating conjunction \( \ast \) is not sufficient to differentiate these two cases; both describe valid partitions of a possible program heap. As a consequence, prior techniques have to revert back to non-modular reasoning to prove that the invariant is maintained.

Flows Redesigned. Our work is inspired by the recent flow framework due to Krishna et al. [2018]. We revisit the core algebra behind flows reasoning, and derive a different algebraic foundation by analysing the minimal requirements for general local reasoning; we call our newly-designed reasoning framework the foundational flow framework. Our new mathematical foundation makes several significant improvements over that of Krishna et al. [2018] and eliminates its most stark limitations. First, we present a simplified and generalized meta theory of flows that makes the approach much more broadly applicable. The original framework cannot reason about programs that make non-trivial changes to cycles (such as the PIP implementation). Thus, algorithms that work on unrestricted graphs are generally not supported by the original framework. Moreover,
the original definition of a flow is too restrictive to express many invariants of interest (including the PIP graph invariant). Next, the proofs of programs shown in [Krishna et al. 2018] are only on paper and depend on a bespoke program logic. This logic requires new reasoning primitives that are not supported by the logics implemented in existing SL-based verification tools. Our general proof technique eliminates the need for a dedicated program logic and can be implemented on top of standard separation logics and existing SL-based tools. Finally, the underlying separation algebra of the original framework makes it hard to use equational reasoning, which is a critical prerequisite for enabling proof automation. We demonstrate that the simplified separation algebra paves the way for automated reasoning about flows using SMT solvers. We provide a more detailed technical comparison to [Krishna et al. 2018] and other related work in §7.

2 FLOWS FROM FIRST PRINCIPLES

In this section, we explain the core mathematical notions behind our foundational flow framework, and their motivation with respect to local reasoning principles. We aim for a general technique for modularly proving the preservation of recursively-defined invariants over partial graphs, with well-defined decomposition and composition operations. Partiality is essential for modular reasoning; when applying our reasoning technique to programs, method calls need not be specified with knowledge of the global graph, and in a concurrent setting, multiple threads can simultaneously operate on disjoint subgraphs.

Flow Domain. Our foundational flow framework is parametric with an underlying flow domain which is a four-tuple \((M, 0, +, E)\) whose components are elaborated and motivated next. As we will show, different instantiations of these parameters can capture a flexible variety of graph properties which can be tracked and reasoned about compatibly with separation-logic-style local reasoning.

Flow Values and Flows. General recursive properties of graphs naturally depend on non-local information; for example, we cannot express that a graph is a DAG directly as a conjunction of independent invariants per node in the graph. To make expressing such properties possible locally, we require a means of summarising this external information, embodied by flow values in our technique; the set of flow values \(M\) is the first parameter of our flow domains. A flow value is assigned (under constraints explained below) to each node in a graph, capturing sufficient information about the graph to express and reason about non-local properties of interest. Our technique enforces minimal restrictions on the choice of \(M\), which gives it its generality; we consider three examples for the rest of this section:

**PIP Domain** For reasoning about the PIP example (cf. Figure 1) flow values capture multisets of integers, representing the priorities of the current node and all those directly referencing it in the PIP data structure.

**Path Counting** Flow values capture the number of paths from a distinguished root node \(r\); one can then express that a graph is a tree rooted at \(r\) with the local condition that the flow at each node is 1.

**Inverse Reachability** Flow values capture multisets of sets of nodes; each set represents the nodes along a simple path (one with no cycles) leading to the current node in the graph.

For a graph \(G\) over a set of nodes \(N\) we express properties of \(G\) in terms of node-local conditions that may depend on the nodes’ flow. A flow is a function flow \(\colon N \rightarrow M\) that assigns every node a flow value and must be some fixpoint of the following flow equation:

\[
\forall n \in N. \text{flow}(n) = \text{in}(n) + \sum_{n' \in N} \text{flow}(n') \triangleright e(n', n)
\]
Intuitively, one can think of the flow as being obtained by a fold computation over the graph\(^2\): the
inflow in: \( N \rightarrow M \) defines an initial flow at each node. This initial flow is then updated recursively
as follows. For every node \( n \), the current flow value at its predecessor nodes \( n' \) is transferred to \( n \)
via edge functions \( e(n', n) : M \rightarrow M \) (we use \( \ast \) to denote function application where the function
is on the right). The transferred flow values are aggregated using the summation operation \( + \) provided
by the flow domain to obtain the updated flow of \( n \). We next motivate the individual components
of the flow equation and discuss the constraints imposed on them by local reasoning principles.

Edge Functions. In any partial graph, the flow value assigned to a node by a flow is propagated to
its neighbours (and transitively) according to a labelling of pairs of nodes \((n, n')\) with edge functions
\( e : M \rightarrow M \), mapping the flow value at the source node \( n \) to one propagated on this edge to the target
node \( n' \). We require such a labelling for all pairs of source node \( n \) inside the graph, and target node
\( n' \) (possibly outside the graph), but provide a convenient default case. We require a distinguished
0 flow value to represent no flow (the second element of our flow domains); the corresponding
(constant) zero function \( \lambda_0 = (\lambda m. \ 0) \) then conceptually represents the absence of an edge in the
graph\(^3\). We write \( e(n, n') \) for the edge function labelling the pair \((n, n')\). A set of edge functions \( E \)
from which this labelling is chosen makes up the fourth element of our flow domains; as we will
see in §3.3, restrictions to certain sets \( E \) can be exploited to strengthen our overall technique.

For our PIP Domain example, the set of edge functions would be zero functions (where no edge
exists in the PIP structure), and otherwise functions which return a singleton multiset storing the
maximum value in the input multiset. In the path-counting example, the edge functions would be
identity functions (edge present) and zero functions (edge absent). For inverse reachability, the
(non-zero) edge functions add the source node to each set in the input multiset, except those which
already contained the node, which are dropped from the multiset (such cases pertain to cycles in
the graph, and this particular domain tracks only simple paths).

Flow Aggregation and Inflows. The flow value at a node is defined by those propagated to it
via edge functions, along with an additional inflow value, explained here. Since multiple edges
can reach a single node, we need to model the aggregation of these values, for which a binary
+ operator on flow values must be defined; the third element of our flow domains. To make this
aggregation of values order-independent, we require + to be commutative and associative. The 0
flow value must act as a unit with respect to +. E.g., in our multiset-based flow domains we let + be
multiset union whereas in the path-counting flow domain + means addition on natural numbers.

Every node has an inflow, modelling contributions to its flow value which do not come from
nodes inside the graph. This inflow term plays two important roles: firstly, since our graphs are
partial, the inflow models the contributions from nodes outside of the graph in question. Secondly,
inflow can be artificially added as a means of specialising the computation of flow values. For
example, in our PIP domain, the inflow of each node will be the singleton multiset containing the
node’s default priority. In our path counting domain, we can select the distinguished root node \( n \)
by giving it an inflow of 1; we could also do this for multiple nodes, to count paths from each. In
the inverse reachability domain, we can employ multisets containing a single empty set, forcing
paths from these nodes to be tracked by the flow computation.

The aggregation of flow values from both edge functions \( e \) inside a graph with nodes \( N \) and inflow
in then gives rise to the flow equation (1); preserving solutions to this equation is a fundamental
goal of our technique. A graph (including its edge functions and flow value assignment) is called a

\(^2\)We note that flows are not generally defined in this manner as we consider any fixpoint of the flow equation to be a flow.
Though, the analogy helps to build the right intuition.

\(^3\)We will sometimes informally refer to paths in a graph as meaning sequences of nodes for which no edge function labelling
a consecutive pair in the sequence is the zero function \( \lambda_0 \).
flow graph if there is some choice of inflow for its nodes satisfying the flow equation. We now turn to how this property can be preserved under changes to the graph in order to aid the verification of flow-dependent invariants.

**Graph Updates and Cancellativity.** Consider that we take a flow graph along with a correct inflow, and obtain a modified graph different only in that a single pair of nodes \((n_1, n_2)\) has a different edge function. We are concerned with the question of whether and how we can change the flow values in the new graph (keeping the inflow unchanged) to satisfy the flow equation.

Consider first the simplified case that the target \(n_2\) of the modified edge propagates no flow via edge functions \((e(n_2, n') = \lambda_0\) for all \(n')\); it may however receive additional flow from edge functions \(e(n'', n_2)\) coming from nodes \(n''\) other than \(n_1\). For example, in the PIP graph shown in Fig.1, removing the edge from \(p_5\) to \(r_4\) (i.e. setting it to the zero function \(\lambda_0\)) does not affect the current priority of \(r_4\) whereas if \(p_7\) had current priority 1 instead of 2, then the current priority of \(r_4\) would have to decrease. For this reason, our PIP Domain aggregates multisets of incoming flow values, rather than having + simply collapse these to their maximum. The multisets contain enough information to locally adjust the flow value when an edge is removed from the graph, whereas if we knew only the maximum and removed an edge \((p, r)\) which provided exactly this value, we could not decide whether or not to decrease the flow value of \(r\) without some knowledge of all of \(r\)’s incoming edges. In this example, recomputing the flow value for \(r_4\) is simply a matter of subtraction (removing \(\{2\}\) from the multiset at \(r_4\)); this exploits the property that this flow domain is cancellative with respect to +, giving us a unique solution. Note that without this property, the recomputation of a flow value for the target node \(n_2\) of the removed edge consistent with the rest of the graph would in general depend on the values of flow\((n'')\)\(\times e(n'', n_2)\) for all nodes \(n''\) (such as \(p_7\) in our example), causing the recomputation to concern unboundedly-many nodes which were not involved in the change to the edge.

Mathematically, cancellativity is the key property which allows removal of edges to be handled without this non-local dependency; for this reason, we make cancellativity of + a requirement on our flow domains\(^4\).

**Flow Footprints and Interfaces.** The cancellativity of + alone is not sufficient to reason locally about general flow graph modifications; it avoids dependency on arbitrary incoming edges, but once we lift the above assumption that the target node \(n_2\) had no non-zero outgoing edges, recomputing a single flow value is not sufficient; we need to account for the propagation of this change transitivity throughout the graph. For example, if we add the edge \((p_1, r_1)\) in Fig. 1 and hence, 3 to the flow of \(r_1\), we also add 3 to the flow of all other nodes reachable from \(r_1\). On the other hand, adding an edge from \(r_4\) to \(p_3\) only affects these two nodes. To capture the relative locality of the side-effects of such updates, we introduce the notion of flow footprint of a modification to a flow graph. A modification’s flow footprint is the smallest subset of the graph nodes containing those nodes which are sources of modified edges, plus all those whose flow values need to be changed in order to obtain a new flow graph (with an unchanged inflow). For example, the flow footprint for the addition of the edge \((p_1, r_1)\) in Fig. 1 is \(p_1\) and all nodes reachable from \(r_1\) (including \(r_1\) itself).

Flow footprints can be used to localise the effect of a graph modification; from the perspective of the graph outside of the flow footprint, nothing observable in the flow domain has changed. We use this observation to extract an abstraction of flow graphs which we call flow interfaces; essentially capturing the pointwise inflow and outflow of the graph (being the flow contributions its nodes make to all nodes outside of the graph), but abstracting over the nodes and flow values inside. Our

---

\(^4\)As we will show in §3.2, an analogous problem for composition of flow graphs is also directly solved by this choice to force aggregation to be cancellative.

abstraction thus identifies flow graphs which “look the same” to the external graph; the same values are propagated inwards and outwards. This idea, while simple, turns out to be powerful enough to build a separation algebra over our reasoning technique, allowing graphs to be decomposed, locally modified and recomposed in ways yielding all the local reasoning benefits of separation logics. In particular, for graph operations within a subgraph with a certain interface, we need to prove: (a) that the modified subgraph is still a flow graph (by checking that the flow equation still has a solution locally in the subgraph) and (b) that it satisfies the same interface (in other words, the flow footprint of the modification is within the subgraph), and the meta-level results for our technique justify that we can recompose the modified subgraph with any graph that the original could be composed with. These steps form the core of our reasoning technique, and are defined formally in §3.

Local Reasoning Challenges. Our main technique, elaborated in the following section, employs flow interfaces to abstract over subgraphs in which localised graph modifications can be shown to be opaque to the remaining graph. Despite the power of this mechanism, when applying it in practice with a particular flow domain and graphs, the following key questions arise:

1. When does the flow equation have a fixpoint solution? How can this be checked, in particular, when is a newly-modified subgraph a flow graph?
2. When is a solution to the flow equation guaranteed to be unique? If desired, how can this property be enforced and preserved?
3. Which graph modifications have which flow footprints? In particular, how localised can their effects on flow values be?

The first two questions are particularly pertinent when we use the flow values of each node in a graph to express properties of interest. For example, consider the path-counting flow domain. If the graph contains a cycle that is reachable from the dedicated root nodes (i.e., those with non-zero inflow), then no flow exists. On the other hand, if the graph contains cycles but no cycle is reachable from the root nodes, then many flows exist; we can assign arbitrary flow values to the cycles as long as the nodes on each cycle are assigned the same value. The property that all nodes have a flow value of 1 expresses that the graph is a tree only if we can restrict the fixpoint solution to be one that assigns 0 flow to unreachable cycles.

In the next section, we formalise our base flow framework and core reasoning techniques, and present a variety of techniques for addressing these key questions.

3 THE FOUNDATIONAL FLOW FRAMEWORK

We now present the formal development of our flow framework: a general technique for preserving global properties of graphs using modular reasoning. We begin with some basic definitions and notations that we use in the rest of this paper.

3.1 Preliminaries

Definition 3.1. A partial monoid is a set \( M \), along with a partial binary operation \( + : M \times M \rightarrow M \), and a special zero element \( 0 \in M \), such that (1) \( + \) is associative, i.e., \((m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)\); and (2) \( 0 \) is an identity element, i.e., \( m + 0 = 0 + m = m \). Here, equality means that either both sides are defined and equal, or both sides are undefined.

We identify a partial monoid with its support set \( M \). If \( + \) is a total function, then we call \( M \) a monoid. Let \( m_1, m_2, m_3 \in M \) be arbitrary elements of the (partial) monoid in the following. We call a (partial) monoid \( M \) commutative if \( + \) is commutative, i.e., \( m_1 + m_2 = m_2 + m_1 \). Similarly, a
commutative monoid \( M \) is called \( \textit{cancellative} \) if + is cancellative, i.e., \( m_1 + m_2 = m_1 + m_3 \) implies \( m_2 = m_3 \).

We say \( M \) is \( \textit{positive} \) if \( m_1 + m_2 = 0 \) implies that \( m_1 = m_2 = 0 \).

\textbf{Definition 3.2 (Separation Algebra [Calcagno et al. 2007]).} A \textit{separation algebra} is a cancellative, partial commutative monoid.

We write \( \delta_{n=n'} : M \rightarrow M \) for the function defined by \( \delta_{n=n'}(m) \) := \( m \) if \( n = n' \) else \( 0 \). We also write \( \lambda_0 := (\lambda m. 0) \) for the identically zero function, \( \lambda_{id} := (\lambda m. m) \) for the identity function, and use \( e \equiv e' \) to denote function equality. For \( e : M \rightarrow M \) and \( m \in M \) we write \( m \circ e \) to denote the function application \( e(m) \).

For \( e, e' : M \rightarrow M \), we write \( e + e' \) for the function that maps \( m \in M \) to \( e(m) + e'(m) \).

We lift this construction to a set of functions \( E \) and write it as \( \sum_{e \in E} e \).

We write \( e \circ e' \) to denote function composition, i.e., \( (e \circ e')(m) = e'(e(m)) \) for \( m \in M \), and use superscript notation \( e^p \) to denote the function composition of \( e \) with itself \( p \) times.

\textbf{Definition 3.3.} A function \( e : M \rightarrow M \) is called an \textit{endomorphism} on \( M \) if for every \( m_1, m_2 \in M \), \( e(m_1 + m_2) = e(m_1) + e(m_2) \). We denote the set of all endomorphisms on \( M \) as \( \text{End}(M) \).

Note that for every endomorphism \( e \in \text{End}(M) \), \( e(0) = 0 \) by cancellativity. It is easy to see that \( e + e' \in \text{End}(M) \) for any \( e, e' \in \text{End}(M) \). Similarly, for \( E \subseteq \text{End}(M) \), \( \sum_{e \in E} e \in \text{End}(M) \). We say that a set of endomorphisms \( E \subseteq \text{End}(M) \) is \textbf{closed} if for every \( e, e' \in E \), \( e \circ e' \in E \) and \( e + e' \in E \).

\subsection{3.2 Flows}

A flow is a recursively-constrained quantity expressed over a labelled graph. The graph labels and flow values are determined by a flow domain; our entire theory is parametric on this domain.

\textbf{Definition 3.4 (Flow Domain).} A \textit{flow domain} \((M, 0, +, E)\) consists of a commutative cancellative (total) monoid \((M, 0, +)\) and a set of functions \( F \subseteq M \rightarrow M \).

\textbf{Example 3.5.} The flow domain used for the path-counting flow is \((\mathbb{N}, +, 0, \{\lambda_{id}, \lambda_0\})\), consisting of the monoid on natural numbers under addition and the set of edge functions containing only the identity function and the zero function.

\textbf{Example 3.6.} For the PIP, we define the flow domain \((\mathbb{N}^\mathbb{P}, \cup, 0, \{\lambda S. \{\max(S)\}, \lambda_0\})\), consisting of the monoid of multisets of natural numbers under multiset union and two edge functions: \( \lambda_0 \) and the function mapping a multiset \( S \) to the singleton multiset containing the maximum value in \( S \).

\textbf{Example 3.7.} Given two flow domains \((M_1, +_1, 0_1, E_1)\) and \((M_2, +_2, 0_2, E_2)\), the \textit{product} domain \((M_1 \times M_2, +, (0_1, 0_2), E_1 \times E_2)\) where \((m_1, m_2) + (m'_1, m'_2) := (m_1 + m'_1, m_2 + m'_2)\) is a flow domain.

In the rest of this section we fix a flow domain \((M, E)\) and a (potentially infinite) set of nodes \( \mathcal{R} \). We abstract heaps using directed partial graphs; integration of our graph reasoning with direct proofs over program heaps is handled in §4. The graphs are partial because they describe abstractions of heaplets rather than the whole heap.

\textbf{Definition 3.8 (Graph).} A \textit{(partial) graph} \( G = (N, e) \) consists of a finite set of nodes \( N \subseteq \mathcal{R} \) and a mapping from pairs of nodes to edge functions \( e : N \times \mathcal{R} \rightarrow E \).

A flow of graph \( G = (N, e) \) under \textit{inflow} : \( N \rightarrow M \) is a solution of the following fixpoint equation (the same as (1), repeated for clarity) over \( G \), denoted \( \text{FlowEqn}(\text{in}, e, \text{flow}) \):

\[ \forall n \in \text{dom}(	ext{in}). \text{flow}(n) = \text{in}(n) + \sum_{n' \in \text{dom}(\text{in})} \text{flow}(n') \circ e(n', n) \]  

\textbf{(FlowEqn)}

\textbf{Example 3.9.} Consider the graph in Figure 1; if the flow domain is as in Example 3.6, \( \text{in} \) assigns to every node \( n \) the multiset containing \( n \)'s default priority and we let \( \text{flow}(n) \) be the multiset labelling every node in the figure, then \( \text{FlowEqn}(\text{in}, e, \text{flow}) \) holds.
An important fact about flows is that any flow of a graph over a product of two flow domains is the product of the flows on each flow domain component; this fact greatly simplifies reasoning about overlaid graph structures. Note that a flow of a graph may (in general) not exist, or may not be unique, depending on the possible solutions to the flow equation \((\text{FlowEqn})\) above.

**Definition 3.10 (Flow Graph).** A flow graph \(H = (N, e, \text{flow})\) consists of a graph \((N, e)\) and a function flow: \(N \rightarrow M\) such that there exists an inflow \(\text{in}\): \(N \rightarrow M\) satisfying \(\text{FlowEqn}(\text{in}, e, \text{flow})\). We let \(\text{dom}(H) = N\), and sometimes identify \(H\) and \(\text{dom}(H)\) to ease notational burden. For \(n \in H\) we write \(H_n\) for the singleton flow subgraph of \(H\) induced by \(n\).

Two flow graphs with disjoint domains always compose to a graph, but this will only be a flow graph if their flows are chosen consistently to admit a solution to the resulting flow equation.

**Definition 3.11 (Flow Graph Algebra).** The flow graph algebra \((\text{FG}, \bullet, H_0)\) for flow domain \((M, E)\) is defined by

\[
\text{FG} := H \in \{(N, e, \text{flow}) \mid (N, e, \text{flow}) \text{ is a flow graph}\} \mid H_i
\]

\[
(N_1, e_1, \text{flow}_1) \bullet (N_2, e_2, \text{flow}_2) := \begin{cases} 
H \ & H = (N_1 \cup N_2, e_1 \cup e_2, \text{flow}_1 \cup \text{flow}_2) \in \text{FG} \\
H_i \ & \text{otherwise}
\end{cases}
\]

\[
\_ \cdot H_i := H_i =: H_i \cdot 
\]

\[
H_0 := (e_0, \text{flow}_0)
\]

where \(e_0\) and \(\text{flow}_0\) are the edge functions and flow on the empty set of nodes \(N = \emptyset\).

As discussed in §2, cancellativity of the flow domain operator \(+\) is key to defining an abstraction of flow graphs that permits local reasoning. The following lemma follows from the fact that \(+\) is cancellative.

**Lemma 3.12.** Given a flow graph \((N, e, \text{flow}) \in \text{FG}, there exists a unique inflow \(\text{in}: N \rightarrow M\) such that \(\text{FlowEqn}(\text{in}, e, \text{flow})\).**

Our abstraction of flow graphs consists of two complementary notions. Lemma 3.12 implies that any flow graph has a unique inflow. Thus we can define an inflow function that maps each flow graph \(H = (N, e, \text{flow}) \neq H_i\) to the unique inflow \(\text{in}(H): H \rightarrow M\) such that \(\text{FlowEqn}(\text{in}(H), e, \text{flow})\). We can also define the **outflow** of \(H\) as the function \(\text{out}(H): \emptyset \setminus N \rightarrow M\) defined by

\[
\text{out}(H)(n) := \sum_{n' \in N} \text{flow}(n') \ast e(n', n).
\]

**Definition 3.13 (Flow Interface).** Given a flow graph \(H \in \text{FG}, its flow interface \(\text{int}(H)\) is the tuple \((\text{in}(H), \text{out}(H))\) consisting of its inflow and its outflow. The set of all flow interfaces is \(\text{FI} := \{\text{int}(H) \mid H \in \text{FG}\}\).**

We use \(I\) to range over interfaces, and write \(I^\text{in}, I^\text{out}\) for the two components of the interface \(I = (\text{in}, \text{out})\). The interfaces of a singleton flow graph containing \(n\) capture the flow and the outflow values propagated by \(n\’s\) edges:

**Lemma 3.14.** For any flow graph \(H = (N, e, \text{flow})\) and \(n, n' \in N\), \(\text{int}(H_n)^\text{in}(n) = \text{flow}(n)\) and \(\text{int}(H_n)^\text{out}(n') = \text{flow}(n) \ast e(n, n')\).**

We next show the key result for this abstraction: the ability for two flow graphs to compose depends only on their interfaces.

**Lemma 3.15.** \(\text{int}(H_1) = \text{int}(H'_1) = I_1 \land \text{int}(H_2) = \text{int}(H'_2) = I_2 \Rightarrow \text{int}(H_1 \bullet H_2) = \text{int}(H'_1 \bullet H'_2)\).
We can now define the flow interface algebra as follows:

\(\text{FIA} := I \in \{\text{int}(H) \mid H \in \text{FG}\} \mid I_1\)  
\(I_0 := \text{int}(H_0)\)

\(I_1 \oplus I_2 := \text{int}(H_1 \bullet H_2)\) for any \(H_1, H_2\) s.t. \(\text{int}(H_1) = I_1 \land \text{int}(H_2) = I_2\).

In all other cases, we define \(I_1 \oplus I_2 = I_1\). We call any interface \(I \neq I_1\) a valid interface, denoted \(\forall V(I)\).

The following result shows that we can use flow interfaces as an abstraction compatible with separation-logic-style framing.

**Theorem 3.16.** The flow interface algebra \((\text{FIA}, \oplus, I_0)\) is a separation algebra.

We next make the notion of flow footprint that we introduced in §2 formally precise.

**Definition 3.17 (Flow Footprint).** Let \(H\) and \(H'\) be flow graphs such that \(\text{int}(H) = \text{int}(H')\), then the flow footprint of \(H\) and \(H'\), denoted \(\text{ffp}(H, H')\), is the smallest flow graph \(H'_1\) such that there exists \(H_1, H_2\) with \(H = H_1 \bullet H_2, H' = H_1' \bullet H_2\) and \(\text{int}(H_1) = \text{int}(H'_1)\).

The following lemma states that the flow footprint captures exactly those nodes in the graph that are affected by a modification (i.e. either their flow or their outgoing edges change).

**Lemma 3.18.** Let \(H\) and \(H'\) be flow graphs such that \(\text{int}(H) = \text{int}(H')\), then for all \(n \in H, n \in \text{ffp}(H, H')\) iff \(H_n \neq H'_n\).

In general, when modifying a flow graph \(H\) to another flow graph \(H'\), requiring that \(H'\) satisfies the same interface \(\text{int}(H)\) can be too strong a condition. In particular, it does not permit allocating new nodes in the modified region. Instead, we want to allow \(\text{int}(H')\) to differ from \(\text{int}(H)\) in that the new interface could have larger domain, as long as the new nodes are fresh and edges from the new nodes do not change the outflow of the modified region.

Formally, we say an interface \(I = (\text{in}, \text{out})\) is contextually extended by \(I' = (\text{in}', \text{out}')\), written \(I \preceq I'\), if and only if (1) \(\text{dom}(\text{in}) \subseteq \text{dom}(\text{in}')\), (2) \(\forall n \in \text{dom}(\text{in})\). \(\text{in}(n) = \text{in}'(n)\), and (3) \(\forall n' \notin \text{dom}(\text{in})\). \(\text{out}(n') = \text{out}'(n')\). The following theorem states that contextual extension preserves composability and is itself preserved under interface composition.

**Theorem 3.19 (Replacement Theorem).** If \(I = I_1 \oplus I_2\), and \(I_1 \preceq I'_1\) are all valid interfaces such that \(I'_1 \cap I_2 = \emptyset\), and \(\forall n \in I'_1 \setminus I_1\). \(I_2\text{out}(n) = 0\), then there exists a valid \(I' = I'_1 \oplus I_2\) such that \(I \preceq I'\).

### 3.3 Existence and Uniqueness of Flows

We typically express global properties of a graph \(G = (N, e)\) by fixing a global inflow \(\text{in} : N \rightarrow M\) and then constraining the flow of each node in \(N\) using node-local conditions. However, as we discussed at the end of §2, there is no general guarantee that a flow exists or is unique for a given \(\text{in}\) and \(G\). The remainder of this section presents three complementary conditions under which we can prove that our flow fixpoint equation always has a unique solution. To this end, we say that a flow domain \((M, 0, +, E)\) has unique flows if for every graph \((N, e)\) over this flow domain and inflow \(\text{in} : N \rightarrow M\), there exists a unique flow that satisfies the flow equation \(\text{FlowEqn}(\text{in}, e, \text{flow})\).

**3.3.1 Edge-local Flows.** A simple but useful case is when all edge functions \(e \in E\) ignore their input; i.e. are constant functions. We call such a flow domain edge-local. In this case, the flow of every node can be computed as a direct aggregation (according to the flow domain operator \(+\)) of the (constant) values its neighbours edge functions propagate; the flow equation is no-longer recursive and always has a unique solution.

**Example 3.20.** The flow of a PIP graph can be encoded using an edge-local flow domain. This is because the PIP implementation tracks the flow explicitly as part of the state of each object (the multisets stored in the \texttt{prios} field). We explain this in more depth in §4.3.
Lemma 3.21. If \((M, 0, +, E)\) is a flow domain such that for every \(e \in E\) there exists \(a \in M\) such that \(e \equiv (\lambda m. a)\), then this flow domain has unique flows.

3.3.2 Nilpotent Cycles. Let \((M, 0, +, E)\) be a flow domain where every edge function \(e \in E\) is an endomorphism on \(M\). In this case, we can show that the flow of a node \(n\) is the sum of the flow as computed along each path in the graph that ends at \(n\). Suppose we additionally know that the edge functions are defined such that their composition along any cycle in the graph eventually becomes the identically zero function. In this case, we need only consider finitely many paths to compute the flow of a node, which means the flow equation has a unique solution.

Formally, such edge functions are called nilpotent endomorphisms:

Definition 3.22. A closed set of endomorphisms \(E \subseteq \text{End}(M)\) is called nilpotent if there exists \(p > 1\) such that \(e^p \equiv 0\) for every \(e \in E\).

Example 3.23. The edge functions of the inverse reachability domain of §2 are nilpotent endomorphisms (taking \(p = 2\)).

If all edges of a flow graph are labelled with edges from a nilpotent set of endomorphisms, then the flow equation has a unique solution:

Lemma 3.24. If \((M, 0, +, E)\) is a flow domain such that \(M\) is a positive monoid and \(E\) is a nilpotent set of endomorphisms, then this flow domain has unique flows.

3.3.3 Effectively Acyclic Flow Graphs. There are some flow domains that compute flows useful in practice, but which do not guarantee either existence or uniqueness of fixpoints a priori for all graphs. For example, the path-counting flow from Example 3.5 is one where for certain graphs, there exist no solutions to the flow equation, and for others, there can exist more than one. In such cases, we explore how to restrict the class of graphs we use in our flow-based proofs such that each graph has a unique fixpoint; the difficulty is that this restriction must be respected for composition of our graphs. Here, we study the class of flow domains \((M, 0, +, E)\) such that \(M\) is a positive monoid and \(E\) is a set of reduced endomorphisms (defined below); in such domains we can decompose the flow computations into the various paths in the graph, and achieve unique fixpoints by restricting the kinds of cycles graphs can have.

Definition 3.25. A flow graph \(H = (N, e, \text{flow})\) is effectively acyclic (EA) if for every \(1 \leq k\) and \(n_1, \ldots, n_k \in N\),

\[
\text{flow}(n_1) \triangleright e(n_1, n_2) \triangleright \cdots \triangleright e(n_k, n_1) = 0.
\]

The simplest example of an effectively acyclic graph is one with a zero edge function on at least one edge of every cycle. However, our semantic condition is weaker: for example, when reasoning about two overlaid acyclic lists whose union happens to form a cycle, a product of two path-counting domains will naturally satisfy effective acyclicity.

Lemma 3.26. Let \((M, 0, +, E)\) be a flow domain such that \(M\) is a positive monoid and \(E\) is a closed set of endomorphisms. Given a graph \((N, e)\) over this flow domain and inflow in: \(N \rightarrow M\), there exists a unique flow that satisfies the flow equation FlowEqn\((in, e, \text{flow})\) and such that \((N, e, \text{flow})\) is effectively acyclic.

While the restriction to effectively acyclic flow graphs guarantees us that the flow is the unique fixpoint of the flow equation, it is not easy to show that modifications to the graph preserve EA while reasoning locally. For instance, it is quite easy to modify a subgraph to another with the same flow interface, which we know guarantees that it will compose with any context, but inadvertently create a cycle in the larger composite graph. However, for a special class of endomorphisms, we
show that a local property of the modified subgraph can be checked, which implies that the modified composite graph continues to be EA.

**Definition 3.27.** A closed set of endomorphisms \( E \subseteq \text{End}(M) \) is called reduced if \( e \circ e \equiv \lambda_0 \) implies \( e \equiv \lambda_0 \) for every \( e \in E \).

Note that if \( E \) is reduced, then no \( e \in E \) can be nilpotent. In that sense, this class of instantiations is complementary to those in \( \S 3.3.2 \).

**Example 3.28.** Examples of flow domains that fall into this class include positive semirings of reduced rings (with the additive monoid of the semiring being the aggregation monoid of the flow domain and \( E \) being any set of functions that multiply their argument with a constant flow value). Note that any direct product of integral rings is a reduced ring. Hence, products of the path counting flow domain are a special case.

For reduced endomorphisms, it is sufficient to check that a modification preserves the flow routed between every pair of source and sink node. This pairwise check ensures that we do not create any new cycles in any larger graph.

**Definition 3.29.** The capacity of a flow graph \( G = (N, e) \) is \( \text{cap}(G) \colon N \times \mathbb{R} \to (M \to M) \) defined inductively as \( \text{cap}(G) := \text{cap}^G(G) \), where

\[
\text{cap}^0(G)(n, n') := \delta_{n=n'} \quad \text{cap}^{i+1}(G)(n, n') := \delta_{n=n'} + \sum_{n'' \in G} \text{cap}^i(G)(n, n'') \circ e(n'', n').
\]

Intuitively, \( \text{cap}(G)(n, n') \) is the function that summarizes how flow is routed from any source node \( n \) in \( G \) to any other node \( n' \), including those outside of \( G \). We define a relation analogous to contextual extension, that constrains us to modifications that preserve EA while allowing us to allocate new nodes.

**Definition 3.30.** A flow graph \( H' \) is a subflow-preserving extension of \( H \), written \( H \preceq_s H' \), if

\[
\forall n \in H, n' \notin H', m. \ m \leq \inf(H)(n) \Rightarrow m \triangleright \text{cap}(H)(n, n') = m \triangleright \text{cap}(H')(n, n'), \quad \text{and}
\]

\[
\forall n \in H' \setminus H, n' \notin H', m. \ m \leq \inf(H')(n) \Rightarrow m \triangleright \text{cap}(H')(n, n') = 0.
\]

We now show that it is sufficient to check our local condition on a modified subgraph to guarantee composition back to an effectively-acyclic composite graph:

**Theorem 3.31.** Let \((M, 0, +, E)\) be a flow domain such that \( M \) is a positive monoid and \( E \) is a reduced set of endomorphisms. If \( H = H_1 \bullet H_2, H'_1, H'_2 \in \mathcal{FG} \) are all effectively acyclic flow graphs, and \( H_1 \preceq_s H'_1 \), then there exists an effectively acyclic flow graph \( H' = H'_1 \bullet H_2 \) such that \( H \preceq_s H' \).

4 PROOF TECHNIQUE

We next show how to integrate flow interface reasoning into a standard separation logic. We present a proof technique that is widely applicable, and illustrate its usage on a simple example program; the same core proof technique can be applied to all examples discussed in the paper.

4.1 Separation Logic

Since flow graphs and flow interfaces form separation algebras, it is possible to define a separation logic (SL) using these notions as its semantic model (indeed, this is the proof approach taken by Krishna et al. [2018]). By contrast, we encode flow interfaces within a standard separation logic without modifying its semantics. This has the important technical advantage that our proof
We now describe our proof technique, using the running example of the insertion procedure on a singly-linked list. While this is a simple example which can be handled by a wide variety of other techniques, we use it to illustrate the key points of our technique as it minimizes any example-specific complexity. As we show subsequently, by layering on more flow domains the same proof technique can be naturally integrated with existing separation logics and verification tools supporting SL-style reasoning. In §5 we demonstrate this concretely for the Viper verifiers, but our technique is also easy to extend to logics such as Iris which support (ghost) resources ranging over user-defined separation algebras [Jung et al. 2017].

Abstracting the Heap using Flow Interfaces. The key idea behind encoding flow interfaces in the simple SL presented above is to use a special ghost field \texttt{intf} to store an interface for every node. The interface stored in this field at address \(x\) is the singleton interface of a flow graph containing only node \(x\). While the foundational flow framework gives us the power to use interfaces of any size in our proofs to reason about concrete states, this choice is in some sense canonical. If we tie the singleton interfaces to abstract the singleton heap regions, then we can express the interface of

\[
\text{Heap} := \{h \mid h : \text{Addr} \rightarrow (\text{Field} \rightarrow \text{Val})\}
\]

It is easy to see that, under the disjoint union operator \(\uplus\), Heap forms a separation algebra.

Let \(\text{Var}\) be an infinite set of variables (we omit sorts and type-checking from the presentation, for simplicity). We assume the syntax of assertions \(\phi\) includes the following, where \(P\) stands for pure first-order assertions, and \(T\) stands for terms:

\[
\phi := P \mid \text{true} \mid x \mapsto \{f_1 : T_1, \ldots \} \mid \phi \ast \phi \mid \bigstar_{x \in X} \phi \mid \ldots
\]

The \(\bigstar_{x \in X} \phi\) syntax represents iterated separating conjunction (the bound variable \(x\) ranges over a set \(X\)). The semantics of the separation logic assertions are standard, and are defined with respect to an interpretation of (logical and program) variables \(i : \text{Var} \rightarrow \text{Val}\). We write \([T]_i\) for the denotation of term \(T\) under interpretation \(i\). In particular, we have:

\[
\begin{align*}
  h, i & |\ x \mapsto \{f_1 : T_1, \ldots, f_k : T_k\} \iff h([x]_i) = \{f_1 \mapsto T_1, \ldots, f_k \mapsto T_k\} \\
  h, i & |\ \phi_1 \ast \phi_2 \iff \exists h_1, h_2. (h = h_1 \uplus h_2) \land (h_1, i \models \phi_1) \land (h_2, i \models \phi_2)
\end{align*}
\]

Note that the logic presented here is garbage-collected [Cao et al. 2017] (also known as intuitionistic). Thus, the semantics of the points-to assertion \(x \mapsto \{f_1 : T_1, \ldots, f_k : T_k\}\) does not restrict the heap \(h\) to only contain the address \(x\), it only requires \(x\) to be included in its domain. This restriction is not essential but simplifies the presentation in the following. We focus here on a simple separation logic without support for e.g. fine-grained concurrency, but our compatibility with standard SL semantics makes integration with more sophisticated logics straightforward.

4.2 Encoding Flow-based Proofs in SL

We now describe our proof technique, using the running example of the insertion procedure on a singly-linked list. While this is a simple example which can be handled by a wide variety of other techniques, we use it to illustrate the key points of our technique as it minimizes any example-specific complexity. As we show subsequently, by layering on more flow domains the same proof sketch extends to insertion into doubly-linked lists or the Harris list, illustrating the power of flow-based proofs to scale to algorithms with arbitrary traversals and complex overlaid structures.

Local Reasoning for Global Graph Properties 1:13
Fig. 2. A proof sketch using our flow-based proof technique for the insert procedure of a singly-linked list.

any set of nodes as the composition of the respective singleton interfaces. On the other hand, if we only tied the interface of a larger region to the heap, we would lose the ability to precisely reason about modifications to single nodes when needed (see Lemma 3.14).

**Encoding.** The basic building block of flow-based specifications is a node predicate \( N(x, \vec{t}_x, J, J') \) that abstracts a node \( x \) to the corresponding singleton flow graph:

\[
N(x, \vec{t}_x, J, J') := x \mapsto \{ \text{intf}: J_x, \vec{t}_x \} \cap \text{dom}(J_x) = \{ x \} \land \forall y. J_x \not\in \text{out}(y) = J'_x(x) \not\in \text{edges}(\vec{t}_x, x, y)
\]

Here, \( J \) and \( J' \) are logical variables of type \( \mathcal{Rl} \to \mathcal{Fl} \), and we use the notation \( f_x \) for the interface that \( J \) maps node \( x \) to. This predicate expresses the fact that we have a heap cell at address \( x \), whose \text{intf} field stores the singleton interface \( f_x \) and whose other fields are captured by the parameter \( \vec{t}_x \) (a list of field-name/value mappings). \( N \) is parametric in a user-defined abstraction function edges that abstracts the field values (as defined by \( \vec{t}_x \)) of \( x \) to the flow graph edge function for the pair of nodes \( (x, y) \); we constrain the outflow of \( x \)'s singleton interface \( f'_x \) in terms of this edge.

The purpose of using two interface maps \( J_x \) and \( J'_x \) is to allow the fields of \( x \) to be abstracted by an interface that may temporarily differ from the one stored in intf. We call such nodes **dirty nodes**. This flexibility will be used in our proofs to model intermediate states between modifying the heap and updating the flow graph abstraction.

---

\[\text{We assume here, for simplicity of presentation, that each address on the heap corresponds to a graph node, i.e.,} \mathcal{Rl} = \text{Addr}. \text{This approach can easily be extended to the case where a graph node represents more than one heap address by replacing the} \ x \mapsto \{ \text{intf}: J_x, \ldots \} \text{predicate with a user-defined predicate spatialRep that can be instantiated to specify all the addresses abstracted by graph node} \ x.\]
Specifications. Specifications in our technique, for instance method pre and postconditions, are naturally expressed using the node predicate $N$ within an iterated separating conjunction over a set of nodes $X$, to express a region of the heap abstracted by a flow interface:

$$
Gr(X, J, y, \varphi) := \bigotimes_{x \in X} (N(x, \bar{t}_x, J) \land y(x, \bar{t}_x, J, J_x^{in}, \ldots)) \land \mathcal{V}(J_x) \land \varphi(J_x)
$$

$Gr(X, J, y, \varphi)$ describes a set of nodes $X$, each of which satisfies $N$ and none of whom are dirty (as we use $N(x, \bar{t}_x, J, J')$ with $J = J'$). $J_x$ is syntactic sugar for the (iterated) interface composition $\bigoplus_{x \in X} J_x$; the assertion $\mathcal{V}(J_x)$ represents that the singleton interfaces stored in the interface fields of nodes in $X$ compose to a valid interface. The $Gr$ predicate is parameterised by two user-specified predicates (chosen per specification construct) $y$ and $\varphi$, used to encode the invariants of the data structure in question. Node-local properties, including constraints on the flow values of nodes, can be written in the predicate $y$. The predicate $\varphi$ can instead be used to constrain the composed interface $J_x$, for instance expressing that it is a closed region with no outgoing edges.

Consider the list insertion example shown in Figure 2; for this proof, we use the path-counting flow domain from Example 3.5. The abstraction function $\text{edges}$ conditions on the next field of a source node and defines the corresponding graph edge to be the identity function $\lambda_{id}$ if the field is non-null and the zero function $\lambda_0$ otherwise (we write $(P ? E_1 : E_2)$ for C-style conditional expressions). The precondition of traverse on line 7 uses $Gr$ with parameters $y_b$s expressing that each node has a flow (i.e. path count) of 1, and $\varphi$ restricting the composite interface $J_X$ to have inflow 1 at the head of the list $h$, inflow 0 everywhere else, and 0 outflow to all nodes. Thus, the precondition $Gr(X, J, y, \varphi)$ implies that $X$ contains a closed list rooted at $h$.

Method Calls and Frame Reasoning. The list insertion procedure in Figure 2 is structured into two methods: the $\text{traverse}$ method iterates through the list and stops at a nondeterministic point $l$ (this is for simplicity; one could select a position based on value criteria, etc.). It then calls the $\text{insert}$ method, which creates and inserts a new node $n$ after $l$.

$\text{traverse}$ sets a variable $l$ to be the head and reads its next field into $r$ before hitting a loop. The loop’s invariant has the same spatial component as the precondition, but additionally says that $l$ is in $X$, $r$ is $l$’s next field, and if $r \neq \text{null}$ then it is also in $X$ (line 13). This is true initially, and since the definition of edges implies that $l$ has an outflow of 1 to $r$, and $\varphi$ restricting the composite interface $J_X$ is always 0, we can conclude that walking next fields must stay within $X$.

After the loop, the code calls the $\text{insert}$ method. The precondition of $\text{insert}$ (line 22) states that it operates on the singleton set of nodes $X_1 = \{l\}$; it is easy to show that the loop invariant implies this precondition. However, after $\text{insert}$ returns, not only has $l$’s interface potentially changed to a new interface $J'_X$, but the set of nodes has potentially increased to $X'_1$. To establish the postcondition of $\text{traverse}$, we need to show that the interface of $X'_1$ composes with the interface of the frame $X_2 := X \setminus X_1$ resulting in a valid interface satisfying $\varphi$.

To perform this frame reasoning, we use the Replacement Theorem (Theorem 3.19). We know $J_X = J_X^i \oplus J_X^c$ and, from the postcondition of $\text{insert}$, that $J_X^i \preceq J_X^i'$. By standard separation logic framing, we know that any new nodes in $X^c_1$ cannot overlap with any of the nodes in $X_2$. The final condition to prove is that there is no outflow from $X_2$ to any newly allocated node in $X^c_1$. In a general setting, we can discharge this condition using additional ghost state; unfortunately, space does not permit exploring this in the present paper. The condition is in any case guaranteed true in a garbage collected environment, so long as we additionally restrict the abstraction function edges to only propagate flow along an edge $(n, n')$ if $n$ has a (non-ghost) field with a reference to $n'$. This is the condition we use in our Viper implementation in §5.

---

\(^6\)In specifications, we implicitly existentially quantify at the top level over free variables such as $\bar{t}_X$, etc.
Our Replacement Theorem allows us to deduce that $J_{X'} = J_{X'}^i \oplus J_{X'}$ is a valid interface and $J_X \preceq J_{X'}$. By setting $J' = J$ for all nodes in $X_2$, and checking that $\preceq$ preserves the conditions in $\phi$, we obtain the postcondition of traverse in line 8.

A similar argument is used in the proof of insert. After reading $l$’s next field into $r$, the code allocates a new node at address $n$ using the inbuilt heap allocation procedure alloc. It then calls a helper method initNode that initializes $n$’s intf field and has the following specification (its proof is omitted here, but can also be discharged with our technique):

$$\{ n \mapsto \_ \} \text{initNode}(n, m) \{ \langle N(n, \vec{f}_n, J, J) \ast I_0 \preceq J_n \ast J_n^\text{in} = m \rangle \}
$$

The postcondition gives us a graph node $N(n, \vec{f}_n, J, J)$ whose singleton interface $J_n$ is a contextual extension of the empty interface $I_0$. Contextual extension forces $n$ to have no outflow, but since $n$ is a new node we determine $n$’s inflow by the method argument $m$. In the case of insert, since we expect to satisfy the flow invariant $\gamma_l$ by linking $n$ into the list, we do not need any external inflow to $n$, so we pass 0. Again, following our proof technique, we use the Replacement Theorem to lift the contextual extension guaranteed by alloc to the current footprint, $J_{X_1} \preceq J_{X_1}'$, obtaining the intermediate assertion shown on line 29.

### Reasoning about Modifications

When the program we are reasoning about makes changes to the heap, the modified nodes become dirty: for some nodes we hold $N(x, \vec{f}_x, J, J')$ with $J_x \neq J'_x$. Eventually, we need to update the intf fields of all such dirty nodes, to once more obtain a state in which the heap is in sync with the flow abstraction. We add a special procedure sync to update the flow abstraction of a given dirty region $D$:

$$\left\{ \prod_{x \in D} N(x, \vec{f}_x, J, J') \ast J_D = J_D' \right\} \text{sync}(D) \left\{ \prod_{x \in D} N(x, \vec{f}_x, J', J') \right\}
$$

The precondition of sync checks that the interface of $D$ is preserved, using the fact that the intf fields of these nodes store the previous interfaces which composed to form an interface for $D$. By Lemma 3.18, we know that for this condition to be satisfied, the dirty region that one syncs must be a superset of the flow footprint of the modification.

To see syncing in action, consider the proof of insert. After allocating a new node $n$, the code manipulates the pointers to link $n$ into the list right after $l$. We now have a state in which both $l$ and $n$ are dirty: due to the heap modification, their fields abstract to different interfaces from those still stored in their intf fields. The dirty region before and after modification is shown in Figure 3. We must now show that there exists a flow in the dirty region $(l, n)$, by exhibiting flow values for these nodes that satisfy the flow equation for the new edge functions in this region. In this case, we can see that there exist flow values $\{ l \mapsto 1, n \mapsto 1 \}$ that satisfy the flow equation in the dirty region. Moreover, we can show that the flow invariant $\gamma_l$ is true given these flow values.
values, and we can calculate the outflow of this region and show that it matches that of the desired interface. We thus obtain the intermediate state on line 31, where the new interfaces $J'_e$ have their outflow components computed from the heap by edges, and inflow components equal to the flow values that we computed (recall that inflows for singleton interfaces are equal to the nodes’ flow values). We can now call \textit{sync}, which checks that the dirty region’s interface is indeed preserved and writes the new interfaces into the \texttt{inf} fields of all nodes in the dirty region, resulting in the state described on line 33. Following our proof technique, we follow this procedure call with an application of the Replacement Theorem, which in this case is a vacuous call since the frame is empty. We thus obtain the postcondition, completing our proof.

\textbf{Effective Acyclicity.} The proof above used the path-counting flow domain, but did not restrict itself to effectively acyclic (EA) flow graphs. Thus, the precondition in line 22 does not imply that $X$ contains only a list rooted at $h$; the flow equation in general has multiple fixpoints, and in particular allows $X$ to contain cycles of nodes with flow 1 unreachable from $h$. To rule out such undesirable cases, we can (as discussed in §3.3.3) use the EA restriction of the path-counting domain.

In this case, one needs to perform two extra checks when syncing the dirty region. First, to prove that a flow exists for the modified dirty region, we check that the dirty region does not contain any cycles (simple, in this case). Second, we must check that the change to the dirty region is a subflow-preserving extension (c.f. Definition 3.30) to rule out the possibility of creating cycles in the larger graph. For space reasons, we omit the full details here, though §5.1.1 shows how we automate such checks for our case studies.

\textbf{Extending To The Harris List.} The power of flow-based reasoning is exhibited when we try to extend this proof to one that inserts a node into an overlaid data structure such as the Harris’ list (a structure in which two linked lists, a \textit{main} list and a \textit{free} list, may overlap with one another) [Harris 2001]. To do so, we can use the product of two path-counting flow domains: one to track the path count from the head of the main list, and one from the head of the free list. The definitions of edges, $\gamma$, and $\phi$ require analogous straightforward changes; the proof outline remains essentially identical. All the complexity of reasoning about the various cases of possible overlap are reduced to our standard requirement of showing that the flow interface of the modified region is preserved. As we show in §5 and in our evaluation (§6), the proof of this overlaid structure can also be automated, putting no additional burden on the end-user.

4.3 The Edge-local Flow Transformation

In the example proof above, it was easy to show that the modification that linked the new node $n$ into the list preserved the interface of some region $X'_i$ because the flow footprint of the modification was small (in this case, equal to the set of modified nodes \{l, n\}). However, in general, a particular flow domain and heap modification may yield a larger (and potentially unbounded) flow footprint. Proving that an unbounded dirty region has a solution to the flow equation, and preserves a prior interface becomes challenging, and generally requires precise information about the graph structure, along with ad-hoc inductive reasoning. This section presents a technique to \textit{transform} a given flow domain into one in which the flow footprint has a well-defined structure and moreover always admits a solution to the flow equation. Our technique breaks down the task of reasoning about an unbounded flow footprint into a series of reasoning steps about smaller, more structured regions, facilitating easier and more natural proofs of programs such as the PIP example.

To motivate the problem, consider indeed the flow domain for the PIP introduced in §2; we employ multisets of (strictly positive) integers to capture the priorities of each node and its predecessors. Recall the situation in which a process $p$ attempts to acquire a resource $r$, adding an edge $(p, r)$ to
the PIP graph. In general such a modification has an unbounded flow footprint, in the worst case comprising all nodes reachable from r (e.g., consider p1 and r1 in Fig. 1).

Constructing proofs in the presence of such unbounded flow footprints (dirty regions, in the proofs) makes it difficult to show, after a modification, that the flow footprint can be abstracted to a valid flow graph matching its prior interface: the key requirement of our local reasoning technique. Demonstrating that the flow equation has a solution in an unbounded dirty region in general requires intimate knowledge of the graph structure and field values within this region. Since the entire region must be synced at once, this knowledge must typically cross at least one specification boundary (e.g. a loop head), adding complexity and length to user specifications and impeding proof automation. Most seriously, for proving this fact, one would have to revert to complex inductive reasoning about fixpoints over general graphs – the very problem that flow-based reasoning is intended to avoid.

Our solution to this problem is to transform the flow computation into one that can be computed simply and incrementally. We do this by transforming the flow domain to one which is edge-local; and modifying any given flow graph by replacing its edge functions with corresponding constant functions providing the same flow values. More precisely, for any flow domain \((M, 0, +, E)\), we define \(E_M := \{(\lambda . m) \mid m \in M\}\) to be the set of constant edge functions. The flow domain \((M, 0, +, E_M)\) is an edge-local flow domain (see §3.3.1), which means any graph over this domain always has a unique flow. The following lemma shows that any flow graph \(H\) over the original flow domain can be transformed into a flow graph \(H'\) over the corresponding edge local domain and vice versa.

**Lemma 4.1.** Let \((M, 0, +, E)\) be a flow domain, \(G = (N, e)\) be a graph over this domain, and \(G' = (N, e')\) a graph over the corresponding edge-local flow domain \((M, 0, +, E_M)\). Then given in, flow : \(N \rightarrow M\), the following statements are equivalent

(i) \(H = (N, e, \text{flow})\) is a flow graph with inflow in
(ii) \(H' = (N, e', \text{flow})\) is a flow graph with inflow in and \(\forall n, n'. e'(n, n') := (\lambda . \text{flow}(n) \triangleright e(n, n'))\)

We can lift this edge-local transformation to our proof technique as follows. We define a new field \(\ell w\), and redefine the original edge functions to propagate (from each source node) the value in its \(\ell w\) instead its the actual flow value, using the following abstraction function:

\[
\text{edges}'(\{\ell w: f, \ell x\} , x, y) := (\lambda . \text{edges}(x, \ell x)(y)(f))
\]

where edges is the original abstraction function. Now, if all nodes in a flow graph satisfy an additional invariant that their flow values match their \(\ell w\) fields, then by Lemma 4.1 the flow values in this transformed program are also a solution to the original fixpoint equation. Effectively, we decouple the transitive propagation of flow values, via the additional field \(\ell w\).

The key advantage of this transformation is that the only node whose flow value can change when an edge \((n, n')\) is modified is the single target node \(n'\). Any outgoing edge from \(n'\) propagates the same value that it used to, because the \(\ell w\) field of \(n'\) has not changed; in other words, the flow footprint for any single edge modification is at most the two nodes that the edge connects. Of course, the invariant that the \(\ell w\) of every node matches its flow may be broken by such an update, but only (immediately) for node \(n'\). Crucially, this discrepancy does not prevent us from calling sync in our proofs; the node whose \(\ell w\) value is out of sync with its actual flow values can now be tracked in pure specifications.

This edge-localisation technique lends itself to natural proofs of programs whose algorithms reflect this break-then-notify style. For instance, returning to the PIP; here is what the key proof
definitions look like after our edge-local transformation:

$$\tilde{f} \equiv \{ \text{flw: S, curr_prio: q, next: y, ...} \}$$

$$\text{edges}(\tilde{f}, x, z) \equiv (z = y \neq \text{null} \ (\lambda_\_ \ \{q\}) : \lambda_0)$$

$$\gamma_{\text{pip}}(x, \tilde{f}, m) \equiv S = m \land q = \text{max}(S)$$

If a state of the PIP satisfies the assertion $\text{Gr}(X, J, \gamma_{\text{pip}}, \text{true})$ then $\gamma_{\text{pip}}$ enforces that the value stored in the flw is equal to the flow, and edges specifies that the outflow to the next node is the maximum of the priorities given by the flow (this is also stored in the curr_prio field).

What is interesting is that the code in Figure 1 already stores the flow in the prios field, and the priority going from one node to the next is this locally cached flow value, meaning that our edge-local flow mirrors the algorithm. Moreover, after modifying an edge, the algorithm calls update to correct the target node, which only updates the prios field of that one node. There is a recursive call to update to fix any downstream nodes whose flow may have changed, but the state before this recursive call can be described more naturally using the edge-local flow as one where every node except $n.\text{next}$ satisfies $\gamma_{\text{pip}}$, and for $n.\text{next}$ the required update is to remove the priority $\text{old_prio}$ and add the priority $n.\text{curr_prio}$ from its prios field (which is also the flow given to $n$).

5 PROOF AUTOMATION

In this section, we develop upon our proof strategy to enable automated checking of flow-based proofs. Our checking simulates a translation from source-level programs annotated with flow specifications, into Viper [Müller et al. 2016] programs whose successful verification implies the existence of a source-level proof. Our goal is for the only requirements on the user to be the definition of the flow domain instantiation (with a chosen labelling function lifting heaps to graphs), and the classical specifications for deductive reasoning (pre/post conditions and loop invariants); essentially only the blue annotations in Figure 2. We show how to automate the two main classes of flow domain introduced in §3: edge-local flow domains, and effectively-acyclic (EA) graphs.

The flow-based proof technique presented in §4 affords a great deal of flexibility in the construction of correctness proofs, which was key to its general applicability. Unfortunately, this flexibility requires creativity on the part of the proof author, and presents a number of challenges for achieving the high degree of automation that we aim for:

(C1) When do we call sync on modified regions? Our proof technique allows us to sync the flow interfaces with the heap at any point and any number of times. To have an automated proof technique, we need to find a strategy that works for a large class of examples.

(C2) Which dirty regions do we sync? As we saw in §4, we must sync a superset of the flow footprint of any modification. As the flow footprint is defined in terms of the flow of the larger graph, it is not always possible to compute the flow footprint in a local and automatable manner. We need a heuristic for choosing a suitable dirty region for a given call to sync.

(C3) When we call sync on a given region, how do we prove that it has a flow and that its interface is preserved? We have seen that certain classes of flow domains (§3.3) guarantee the existence of the flow. However, to express the flow equation (FlowEqn) in the dirty region and to derive its old and new flow interfaces requires sum comprehensions. Furthermore, for the effectively acyclic class, one additionally requires a computation over all paths within the region to establish subflow preservation (Definition 3.30).

(C4) Finally, how do we automatically infer relations between the interfaces of different sizes? In the example proof in §4.2, one had to call the Replacement Theorem after the method call to insert in order to relate the interface $J'_{X_1}$ of insert’s footprint to the interface $J'_{X_1}$ of the caller. One also needs to relate singleton interfaces to the interfaces of larger, potentially unbounded, regions containing them. This reasoning depends on applying the definition of...
interface composition to an unbounded region, something that is hard to do automatically. We need a technique to apply such proof steps to the relevant interfaces at appropriate points. We present solutions to all of these challenges in the remainder of this section, under certain restrictions which we explain next.

Restrictions. Given that automated reasoning about inductive properties such as reachability over unbounded graphs, that can be encoded as flows, are known to be undecidable [Immerman et al. 2004], we must sacrifice completeness for a sound automated proof technique. Our automation story (unlike the by-hand proof technique of the previous section) thus works under some additional restrictions: (1) we require that the proof uses either an edge-local flow domain or the EA restriction; (2) we require that pre/post-condition specifications are each written in terms of node-local conditions $\gamma$ on each node’s concrete fields and flow, and a constraint $\varphi$ on the global interface of the subgraph on which a method operates (i.e., we don’t automate support for other combinations of interfaces in specifications); and (3) our calculation of dirty regions is heuristic, and relies on one of our techniques being employed to localise the flow footprint of each modification to a finitely-bounded (not necessarily statically-known) sub-region of the graph.

As we saw in §4.3, any flow domain can be transformed into an edge-local one that has the same set of solutions to the flow equation. Thus any proof can be done with edge-local flows at the cost of more reasoning steps or ghost code; in many cases (e.g. the PIP, Dijkstra, and Composite examples) however, these steps match those of the code itself. While in theory one could write specifications about multiple potentially-overlapping interfaces of the data structure, we have not yet found an algorithm where this is necessary. Finally, our dirty region calculation heuristic does rule out certain modifications that have an unbounded flow footprint. However, our edge-local transformation can again help tame the flow footprint and bring it within reach of our heuristic.

Note that for fine-grained concurrent algorithms, our requirement to keep the flow footprints local corresponds to locality of atomic updates; any node whose flow invariant is broken for longer would have to be locked to prevent other threads observing inconsistent states. As we demonstrate in our evaluation (§6), we are able to capture a wide variety of data structures, flow domains and properties within these restrictions.

When to Sync. Our aim of requiring minimal and simple annotations from the user guides our approach to Challenge (C1). To avoid requiring user-specifications about dirty regions (which, when later synced, will require fine-grained knowledge of the graph structure and flow values), we perform a sync before each verification boundary, meaning before a loop or method call, and at the end of a loop or method body. This design naturally organises the verification of a method into phases; we begin with a consistent graph and interfaces (and an empty dirty region), make modifications which add some nodes to the dirty region, and then sync these before the next verification boundary back to a consistent interface.

Which Region to Sync. Since we only sync at verification boundaries, we only need to build up a suitable dirty region between each such boundary. Our heuristic for maintaining this dirty region is as follows: every time the field of an object is modified in the program code, we add that object to the dirty region. As discussed above, this might not work for some input programs and flow domains, but constructions such as the edge-local transformation of §4.3 can be employed to overcome this limitation. Combined with our restrictions, this gives us a way to solve Challenge (C2) for a wide variety of challenging examples. We describe how we deal with Challenges (C3) and (C4) in the relevant parts of the description of our automation procedure in the next subsection.
5.1 Automatic Generation of a Flow-based Proof

We now describe our automation technique as a syntactic translation from minimally-annotated programs to Viper files that perform a proof in our proof technique from §4.

Viper’s Logic. The core logic employed by the Viper verification infrastructure is based on the implicit dynamic frames logic [Smans et al. 2009], a close relative of separation logic; the embedding of core separation logic is known [Parkinson and Summers 2012]. Viper natively supports expressing and automatically reasoning about iterated separating conjunctions [Müller et al. 2016]. This is why we chose Viper for demonstrating our automation techniques, but any tool with similarly automated support for iterated separating conjunctions and custom mathematical types/functions could in principle be used in its place. Our encodings could also be adapted to less automated settings; for a more manual proof, our heuristically-chosen dirty regions could be overridden by manual specification; our Viper encoding also supports user-selected additions to this dirty region.

User Input. Our automation relies on the following ingredients as inputs from the user:

- A definition of the flow domain in question, along with the choice of whether this employs edge-local flows, or effectively-acyclic graphs.
- A labelling function edges(x, y), defining, for each pair of references x and y, the edge function between their corresponding graph nodes, in terms of the concrete field values of x
- Pre/post-conditions and loop invariants in terms of both node-local conditions (γ from §4) and constraints on the interface of the composite graph in question (φ from §4).

Generation Procedure. Figure 4 shows the structure of the annotated output Viper program generated by our technique. The input to this generation is a verification phase, i.e. a piece of straight-line code C not containing any loops or method calls, and its pre/post-condition specifications (shown in lines 1 and 20 respectively). Note that as allocation is done by a method call to alloc, C cannot modify the footprint, so the same set of nodes X is used in both specifications (handling method calls is described in a subsequent paragraph). We now show how we instrument C to compute the dirty region as per our heuristic, compute the new flow graph for this region, and check that it preserves its interface so that we can sync the region and prove the postcondition of this phase. We achieve this using the following series of auxiliary steps:

(1) The first and last components (lines 2 and 19) are to assume certain flow interface lemmas that relate interfaces of different regions; these will be described in detail in §5.1.2.
(2) We take a snapshot of the graph edges (given by the edges function) before and after the instrumented code (line 5) and store it in variables \(e\) and \(e’\) respectively.

(3) We instrument the code \(C\) to compute the set of dirty nodes \(D\) prescribed by our heuristic. We do this by following every field write command with a command that adds the source node to the set \(D\), the resulting code is denoted \(\text{instrumentedCode}(C, D)\).

(4) We know from the precondition that all the singleton interfaces \(J_x\) for \(x \in D\) composed to a valid interface. Thus, we can compute the old interface of \(D, J_D\), in terms of the singleton interfaces \(J_x\), denoted \(\text{computeInterface}(D, J, e)\) (described in §5.1.1). Moreover, if we are in the EA case, we can also assume that the old flow graph was EA (line 10), which additionally constrains the singleton interfaces \(J_x\) and edges \(e\).

(5) Before we can compute the new interface of \(D\), we must first check that \(D\) has a flow. In the edge-local case, this is always true, but in the EA case we additionally check that the new edges \(e’\) have not introduced any cycles inside \(D\) (\(\text{assertNoCycles}(J_D, e’))\). We can now use \(\text{computeInterface}(D, J’, e’)\) to compute the new interface \(J’_D\) of \(D\) and relate it to the new singleton interfaces \(J’_x\) for \(x \in D\). Additionally, to ensure in the EA case that we do not introduce new cycles in the larger region \(X\), we also check that \(D\)’s interface is subflow preserving (Definition 3.30).

(6) Finally, we try to sync \(D\). This step checks that the old interface of \(D, J_D = J’_D\), the new interface of \(D\).

This generation relies on several macros (\(\text{computeInterface}, \text{assertNoCycles}\), etc.) that are not easy to encode as Viper assertions because they contain arbitrary sums or reasoning about all paths through a region. We next describe how we encode such computations in an automatable manner.

5.1.1 Computing Interfaces and Flows. Solving challenge (C3), computing the flow and flow interface of a region, boils down to solving two problems: automating the sum of a quantity over a statically-unknown set of nodes \(D\), and automating sums over all paths through a region \(D\).

Given a region \(D\), where each \(x \in D\) has interface \(J_x\), and the interface of \(D\) is \(J_D := \bigoplus_{x \in D} J_x\), we know that the outflow of \(D\) is given by \(I_D^\text{out}(y) = \sum_{x \in D} J_x^\text{out}(x, y)\). Computing this sum automatically is hard because in many programs, the size and contents of \(D\) are statically unknown (certain nodes could only be modified in some branches of the code, may or may not alias one another, etc.). Directly instantiating the flow equation (FlowEqn) is also hard for the same reason. We solve this issue by amending \(\text{instrumentedCode}(C, D)\) to also track the set of nodes \(D\) as a mathematical list \(D_L\) containing each node in \(D\) exactly once (the list’s contents may depend on branch conditions, aliasing, etc., in a way known only to the prover). We then axiomatize a function to simulate a functional program to compute the sum over these terms, using E-matching [Detlefs et al. 2005] in place of pattern-matching, to automate this computation via quantifier instantiation within the SMT solver used by Viper to verify the instrumented program.

In the EA case, checking effective acyclicity (Definition 3.25) of the modified graph and showing that the modified interface is subflow-preserving (Definition 3.30) requires computing the capacity of the (old and new) dirty region, which is a sum over all paths through \(D\). To enumerate these paths, we define the following helper function that approximates the capacity of the dirty region, which is a sum over all paths through a region. We next describe how we encode such computations in an automatable manner.

We use the mathematical list \(D_L\) inspired by the Floyd-Warshall algorithm, and contains terms for all simple paths from \(m\) to \(n\) using nodes in \(D_L\) (and some other cyclic paths, but those will be zero by effective acyclicity). When a flow graph \(H = (N, e, \text{flow})\) is EA (for instance, we know this in the pre-state),
we can assume that for every $n \in N$, $\text{flow}(n) \Rightarrow \text{capAux}_{\text{n}, n, D_L \setminus \{n\}}$. And when we check that the new dirty region is effectively acyclic, since we do not yet know the new flow values, we use the following lemma:

**Lemma 5.1.** Let $(M, E)$ be a flow domain such that $M$ is a positive monoid and $E$ is a reduced set of endomorphisms, $G = (N, e)$ be a graph over this domain, and $\text{in} : N \rightarrow M$ be an inflow for this graph. If for every $n, n' \in N$, $\text{in}(n) \Rightarrow \text{capAux}_{\text{n}, n', D_L \setminus \{n, n'\}} \Rightarrow \text{capAux}_{\text{n}', n', D_L \setminus \{n'\}} = 0$, then there exists $\text{flow} : N \rightarrow M$ such that $(N, e, \text{flow}) \in \text{FG}$ is a flow graph and is effectively acyclic.

### 5.1.2 Flow Interface Lemmas.

The problem to solve in Challenge (C4) is how to axiomatize properties of flow interfaces and their compositions over unbounded regions, and how and when to apply them. It is again not possible to have a complete algorithm to decide properties of flow interfaces, so our approach is to instead select a number of core lemmas about flow interfaces that are typically requires, and axiomatise them in Viper. The lemmas concern the interfaces stored in the $\text{intf}$ fields of nodes; some rely on these corresponding to the current flow values of the nodes. We apply these lemmas only in consistent states, (indicated by $\text{assumeLemmas()}$ in Figure 4) at the beginning and end of each verification phase so that they apply both to already-established and newly-established interfaces arising in our proofs. These lemmas include the Replacement Theorem, which is used to reason about interface composition after a method call (see §4.2), which we apply at the beginning of the subsequent verification phase.

### 5.1.3 Soundness.

The automation technique described in this section is sound, but not complete, due to the heuristic choices made (particularly for selecting our dirty regions). As we show in the next section, it nonetheless works extremely well in practice (and accurately identifies bugs in faulty algorithms). Since our technique is simply a syntactic transformation from a restricted class of input programs to annotated Viper programs with an analogous underlying semantic model, soundness of the separation-logic-level reasoning is established by Viper. The properties not checked by our encoded Viper problems are the meta-theoretical lemmas of the foundational flow framework described and proved in this paper.

### 6 EVALUATION

We evaluate the foundational flow framework and our accompanying automation techniques for its proofs on a collection of challenging data-structure examples. These are hand-encoded into Viper, but our encodings simulate the systematic behaviour of a potential future front-end tool for this reasoning, requiring as input only the flow domain instantiation and specified code as described in §5. The corresponding Viper files, and a version of the PIP showing end-user annotations are provided in the anonymous supplementary materials.

Our collection of examples is presented in Figure 5. Some examples (such as the PIP and Composite) concern edge-local flow domains, while most employ effectively-acyclic graphs (and the corresponding additional requirements on interface composition) to structure the reasoning and automation. We employ a variety of flow domains to succinctly capture the graph properties in question, and show the preservation of the key invariants of each data structure.

We chose our benchmarks based on whether at least one existing SL-based reasoning technique would struggle to cope with them (with the exception of the singly and doubly linked-list examples, which can be handled by most techniques). The first non-trivial group of examples (Harris list, threaded tree) involves operations on overlaid data structures (traversal, insertion, deletion, etc.). The Harris list is a singly-linked list that is arbitrarily overlaid with a free list consisting of marked nodes (inspired by the drain technique of manual memory management). These examples are generally difficult to handle with existing techniques because such data structures allow multiple
traversal strategies (following different sets of pointers in the heap) and the invariants of the overlays can be intertwined (e.g., every node is contained in both structures). Compared to the singly-linked list benchmarks, the specification effort is of similar complexity and scales-up in size roughly linearly with the number of overlays. In particular, the proof of a traversal of the Harris list looks almost identical to that of a simple singly-linked list. The next group of benchmarks (B-tree, hash table) is inspired by the technique for verifying functional correctness of search structures presented in [Krishna et al. 2018, §7]. The specifications of these examples are challenging because they involve non-trivial combinations of data and structural invariants. Next, we consider OOP design patterns (Composite, Subject/Observer). These involve multi-object invariants which are known to be challenging to handle at a per-node level of granularity in separation logics. In a sense, using flows we can mimic existing proof strategies for such examples based on object invariants [Summers and Drossopoulou 2010] within separation logic, obtaining the additional benefits of modular framing. The last two examples deal with recursive properties of general graphs. The PIP example is a complete version of the code shown in Fig. 1. The Dijkstra shortest path algorithm is a sequential variant of the algorithm considered by Raad et al. [2016]. We are not aware of any existing SL-based technique that can automatically verify these kinds of proofs. Moreover, we are not aware of any existing local proofs of the properties we considered for these benchmarks.\footnote{We note that Raad et al. [2016]; Sergey et al. [2015] verify concurrent versions of such algorithms. These works focus on modular reasoning techniques for the concurrency aspects (which we ignore here) while the reasoning about the considered graph properties requires some non-local reasoning steps.}

![Table](image-url)

Fig. 5. The results of our evaluation. Here, “EL” means the flow domain is edge-local, while “EA” means the example employs effectively-acyclic graphs. The “buggy” variants are adapted from the correct code by intentionally seeding mistakes. All examples behave as expected (correct versions verify, buggy versions identify the correct errors). All timings were gathered on an Intel Core i7-7700K 4.20Ghz machine running (64 bit) Windows 10; timings were taken 7 times, the lowest and highest times discarded, and the remaining 5 averaged and reported to two decimal places (we observed no significant variations in timings).
We include both correct and buggy variants of the code, in order to demonstrate that we can effectively identify bugs in faulty proofs, as well as to show that our automation is not significantly slower in searching for a wrong proof. As can be seen, the time taken to check each proof is reasonable despite the absence of manual intervention, and a variety of examples which go beyond the state-of-the-art for any proof technique known to enable a similar degree of automation.

7 RELATED WORK

An abundance of SL variants provide complementary mechanisms for modular reasoning about programs (e.g. [Jung et al. 2017; Raad et al. 2015; Sergey et al. 2015]). Most have in common that they are parameterized by the underlying separation algebra; our flow-based reasoning technique can be easily integrated into these existing logics.

The most common approach to reason about irregular graph structures in SL is to use iterated separating conjunction [Müller et al. 2016; Yang 2001] and describe the graph as a set of nodes each of which satisfies some local invariant. This approach has the advantage of being able to naturally describe general graphs. However, it is hard to express non-local properties that involve some form of fixpoint computation over the graph structure. One approach is to abstract the program state as a mathematical graph using iterated separating conjunction and then express non-local invariants in terms of the abstract graph rather than the underlying program state [Hobor and Villard 2013; Raad et al. 2016; Sergey et al. 2015]. However, a proof that a modification to the state maintains a global invariant of the abstract graph must then often revert back to non-local and manual reasoning, involving complex inductive arguments about paths, transitive closure, and so on. Our technique and Viper encoding also exploit iterated separating conjunction for the underlying heap ownership, with the key benefit that flow interfaces exactly capture the necessary conditions on a modified subgraph in order to compose with any context and preserve desired non-local invariants.

The most closely related work is [Krishna et al. 2018], for which we already provided a high-level comparison in §1. In addition to the technical innovations made here (general proof technique that integrates with existing SLs and proof automation), the most striking difference is in the underlying meta theory. The prior flow framework required flow domains to form a semiring; the analogue of edge functions are restricted to multiplication with a constant, which must come from the same flow value set. Our foundational flow framework decouples the algebraic structure defining how flow is aggregated from the algebraic structure of the edge functions. As a consequence, we obtain a more general framework that applies to many more examples, and with simpler flow domains. Strictly speaking, the prior and our framework are incomparable as the prior did not require that flow aggregation is cancellative. As we argue in §2, cancellativity is a natural requirement for local reasoning, and is critical for ensuring that the inflow of a composed graph is uniquely determined. Instead of demanding cancellativity, Krishna et al. [2018] require proofs to reason about flow interface equivalence classes. This complicates proofs (and their automation, introducing quantifier alternations), and entails strong restrictions on modifications of cyclic structures.

An alternative approach to using SL-style reasoning is to commit to global reasoning but remain within decidable logics to enable automation [Itzhaky et al. 2013; Klarlund and Schwartzbach 1993; Lahiri and Qadeer 2008; Madhusudan et al. 2012; Wies et al. 2011]. However, such logics are restricted to certain classes of graphs and certain types of properties. For instance, reasoning about reachability in unbounded graphs with two successors per node is undecidable [Immerman et al. 2004]. Recent work by Ter-Gabrielyan et al. [2020] shows how to deal with modular framing of pairwise reachability specifications in an imperative setting. Their framing notion has parallels to our notion of interface composition, but allows subgraphs to change the paths visible to their context. The work is specific to a reachability relation, and cannot express the rich variety of custom graph properties available by instantiating flow domains in our technique.
8 CONCLUSIONS AND FUTURE WORK

We have presented the foundational flow framework, enabling local modular reasoning about recursively-defined properties over general graphs. The core reasoning technique has been designed to make minimal mathematical requirements, providing great flexibility in terms of potential instantiations and applications. We identified key classes of these instantiations for which we can provide existence and uniqueness guarantees for the fixpoint properties our technique addresses, and showed how two of these (edge-local flow domains and effectively-acyclic graphs) can be built upon to provide automated, simple proof checking for a wide variety of challenging examples.

As future work, we plan to investigate the potential for an extended meta-theory supporting syncing of partial dirty regions, as an alternative proof technique to our edge-local transformation. We aim to connect our techniques to other verification tools, and to mechanise the theoretical foundations. Finally, we plan to develop a lightweight front-end tool, facilitating the application of our novel reasoning to further examples.

ACKNOWLEDGMENTS

This work is in part supported by the National Science Foundation under grants CCF-1815633, CCF-1618059, and CNS-151442.
REFERENCES


A  AN EXAMPLE PROOF IN OUR FRONTEND

To illustrate the benefits of our proof technique and automation, we list here the code and annotations needed to verify the PIP example. We only show the annotations that need be provided by an end-user, all the remaining parts of the proof are determined by our proof technique and translation to Viper.

Since our target verifier is Viper, which supports heap-dependent functions, we directly write field expressions to refer to field values of nodes $x$ to which we have permission (denoted $\text{acc}(x)$). We use the $\text{Gr}$ predicate from §4.2 to express our specifications: for instance, $\text{Gr}(X, \text{nodeInv}(\_, X), \text{true})$ denotes the flow graph of a set of nodes $X$, where each node $x$ satisfies the node-local predicate $\text{nodeInv}(x, X)$, and the flow interface of $X$ satisfies the trivial condition $\text{true}$. The $\text{nodeInvIntermediate}$ is used to denote states where the priority of a node has not yet been updated to match its flow. We note that the annotations required from the end-user are minimal, and consist of flow domain definitions, local invariants on nodes’ fields and flows, and annotations $\_\text{make\_dirty}$ to manually add neighbours to the dirty region in certain cases.

```plaintext

1 // ---- Flow Domain definitions and PIP invariant:
2
3 type FlowDom = Multiset[Int]
4 // Contains:
5 // fd : Multiset[Int] → FlowDom
6 // fdZero : () → FlowDom
7
8 // The fields of a node:
9 field parent: Ref
10 field current_prio: Int
11 field default_prio: Int
12 field priorities: Multiset[Int]
13
14 // Definition of a node and good condition:
15 function edges(x: Ref, y: Ref) : Map[FlowDom, FlowDom]
16 requires acc(x)
17 {
18   (x != y ∧ x.parent != null ∧ x.parent == y) ?
19     {_, ↦ fd(Multiset(x.current_prio))} : {_, ↦ fdZero()}
20 }
21
22 function correct_priorities(x: Ref, from: Int, to: Int): Multiset[Int]
23 requires acc(x)
24 ensures result == (to > 0 ? x.priorities ∪ Multiset(to) : x.priorities) \ Multiset(from)
25
26 define nodeInvIntermediate(x, y, from, to, X)
27   // priorities are strictly positive
28   x.default_prio > 0 ∧ (∀ i: Int :: i ∈ x.priorities) > 0 ⇒ i > 0
29   ∧ fd(y == x ? correct_priorities(x, from, to) : x.priorities) == x.intf(x)
30   ∧ x.current_prio == max(ms_max(x.priorities), x.default_prio)
31   // No self loops
32   ∧ x.parent != x
33   // Data structure is closed
```

\[ x.\text{parent} \neq \text{null} \implies x.\text{parent} \in X \]

define nodeInv(x, X) nodeInvIntermediate(x, null, 0, 0, X)

// ---- PIP algorithm:

method updatePriorities(this: Ref, from: Int, to: Int, X: Set[Ref])
  requires Gr(X, nodeInvIntermediate(_, this, from, to, X), true)
  requires this \in X
  ensures Gr(X, nodeInv(_, X), true)
{
  if (from != to) {
    var old_prio: Int := this.current_prio
    if (to > 0) {
      this.priorities := this.priorities union Multiset(to)
    }
    this.priorities := this.priorities setminus Multiset(from)
    this.current_prio := max(ms_max(this.priorities), this.default_prio)
    _make_dirty(this.parent)
  }
  if (this.current_prio != old_prio && this.parent != null) {
    updatePriorities(this.parent, old_prio, this.current_prio, X)
  }
}

method acquire(this: Ref, r: Ref, X: Set[Ref])
  requires Gr(X, nodeInv(_, X), true)
  requires this \in X \implies r \in X \implies this \neq r \implies this.parent == null
  ensures Gr(X, nodeInv(_, X), true)
{
  if (r.parent == null) {
    r.parent := this
    _make_dirty(r.parent)
    updatePriorities(this, 0, r.current_prio, X)
  } else {
    this.parent := r
    _make_dirty(this.parent)
    updatePriorities(r, 0, this.current_prio, X)
  }
}

method release(this: Ref, r: Ref, X: Set[Ref])
  requires Gr(X, nodeInv(_, X), true)
  requires this \in X \implies r \in X
  ensures Gr(X, nodeInv(_, X), true)
{
  if (r.parent == this) {
    r.parent := null
    _make_dirty(this)
    updatePriorities(this, r.current_prio, 0, X)
  }