Interprocedural Analysis of Asynchronous Programs *

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Abstract

An *asynchronous program* is one that contains procedure calls which are not immediately executed from the callsite, but stored and "dispatched" in a non-deterministic order by an external scheduler at a later point. We formalize the problem of interprocedural dataflow analysis for asynchronous programs as AIFDS problems, a generalization of the IFDS problems for interprocedural dataflow analysis. We give an algorithm for computing the precise meet-over-valid-paths solution for any AIFDS instance, as well as a *demand-driven* algorithm for solving the corresponding demand AIFDS instances. Our algorithm can be easily implemented on top of any existing interprocedural dataflow analysis framework. We have implemented the algorithm on top of BLAST, thereby obtaining the first safety verification tool for unbounded asynchronous programs. Though the problem of solving AIFDS instances is EXPSPACE-hard, we find that in practice our technique can efficiently analyze programs by exploiting standard optimizations of interprocedural dataflow analyses.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Languages, Verification, Reliability.
Keywords: asynchronous (event-driven) programming, dataflow analysis.

1. Introduction

Asynchronous programming is a popular and efficient programming idiom for managing concurrent interactions with the environment. In addition to the usual, or *synchronous*, function calls where the caller waits at the callsite until the callee returns, asynchronous programs have *asynchronous procedure calls* which, instead of being executed from the callsite, are stored in a task queue for later execution. An application-level *dispatcher* chooses a call from the task queue, executes it to completion (which might lead to further additions to the task queue), and repeats on the remaining pending calls.

Asynchronous calls permit the interleaving of several logical units of work, and can be used to hide the latency of I/O-intensive tasks by deferring their execution to a point where the system is not otherwise busy. They form the basis of event-driven programming, where the asynchronous calls correspond to callbacks that may be triggered in response to external events. Further, if mechanisms to ensure atomicity, either by using synchronization [24] or by using transactions [15, 29], are used to ensure asynchronous calls are executed atomically, then the scheduler can be multi-threaded, running different asynchronous calls concurrently on different threads or processors [32]. There have been a variety of recent proposals for adding asynchronous calls to existing languages via libraries, such as LIBASYNC [20], LIBEVENT [21], and LIBEEL [6, 5]. These libraries have been used to build efficient and robust systems software such as network routers [19] and web servers [25]. Further, several recent languages such as NESC [12], a language for networked embedded systems, and MACE [23], a language to build distributed systems, provide explicit support for asynchronous calls.

The flexibility and efficiency of asynchronous programs comes at a price. The loose coupling between asynchronously executed methods makes the control and data dependencies in the program difficult to follow, making it harder to write correct programs. As asynchronous programs are typically written to provide a reliable, high-performance infrastructure, there is a critical need for techniques to analyze such programs to find bugs early or to discover opportunities for optimization.

For programs that exclusively use synchronous function calls, *interprocedural dataflow analysis* [31, 28] provides a general framework for program analysis. In the setting of [28], interprocedural dataflow problem is formulated as a context-free reachability problem on the program graph, *i.e.*, a reachability problem where the admissible paths in the graph form a context free language of nested calls and returns. Unfortunately, this approach does not immediately generalize to asynchronous programs, for example, by treating asynchronous calls as synchronous. In fact, such an analysis yields unsound results, because the facts that hold at the point where the asynchronous call is made may no longer hold at the point where the stored call is finally dispatched. Though the values passed as parameters in the asynchronous call remain unaltered till the dispatch, the operations executed between the asynchronous call and its dispatch may completely alter the values of the global variables. Further, the pairing of asynchronous calls and their actual dispatches makes the language of valid program executions a non-context free language, and a simple reduction to context free reachability seems unlikely.

This paper formalizes the problem of dataflow analysis for asynchronous programs as *Asynchronous Interprocedural Finite Distributive Subset* (AIFDS) problems, a generalization of the IFDS problems of Reps, Horwitz and Sagiv [28] to programs that additionally contain asynchronous procedure calls. The key challenge in devising algorithms to solve AIFDS problems precisely, that is, to compute the *meet over all valid paths* (MVP) solutions for such problems, lies in finding a way to handle the unbounded set

* This research was sponsored in part by the research grants NSF-CCF-0427202, NSF-CNS-0541606, and NSF-CCF-0546170.

POPL'07 January 17–19, 2007, Nice, France.

of pending asynchronous calls, in addition to the unbounded call stack. We surmount this challenge through three observations.

1. **Reduction** We can *reduce* an AIFDS instance into a standard, synchronous dataflow analysis problem where the set of dataflow facts is the product of the original set with a set of *counters* which track, for each of finitely many *kinds* of pending calls, the exact *number* of instances of the call that are pending. Though the reduced instance has the same solution as the AIFDS instance, we cannot use standard dataflow analyses to compute the solution as the lattice of dataflow facts is now unbounded: the counters can grow unboundedly to track the number of pending asynchronous calls.
2. **Approximation** Given any fixed parameter $k \in \mathbb{N}$, we can compute approximations of the meet-over-valid path solutions in the following way. We compute an *under-approximation* of the infinite reduced instance using a counter that counts up to k, dropping any asynchronous call if there are already k pending instances for that call. We call this problem the k-reduced IFDS problem. We compute an *over-approximation* of the infinite reduced instance using a counter that counts up to k, and bumps up to infinity as soon as the value exceeds k. This has the effect of tracking up to k pending calls precisely, and then supposing that an unbounded number of calls are pending if an additional asynchronous call is performed. We call this problem the k^∞-reduced IFDS problem. For each k, both the over- and the under-approximations are instances of standard interprocedural dataflow analysis as the abstraction of the counters makes the set of dataflow facts finite. Thus, we can compute over- and under-approximations of the precise solution of the AIFDS instance by running standard interprocedural dataflow analysis algorithms [28].
3. **Convergence** In a crucial step, we prove that for each AIFDS instance, there always exists a k for which the solutions of the over-approximate IFDS instance and the under-approximate IFDS instance coincide, thereby yielding the precise solution for the AIFDS instance. Thus, our simple algorithm for computing the meet over valid paths solutions for AIFDS instances is to run an off-the-shelf interprocedural analysis on the k and k^∞-reduced IFDS instances for increasingly larger values of k, until the two solutions converge upon the precise AIFDS solution.

The proof of the third observation, and therefore, that our algorithm is complete, proceeds in two steps. First, we demonstrate the existence of a *finite* representation of the *backward* or *inverse* MVP solution of the infinite reduced instance. To do so, we design a *backward* version of the algorithm of Reps, Horwitz and Sagiv [28] and prove that it terminates with the finite upward-closed backwards solution by using properties of well quasi-orderings [1, 10]. Second, we prove that if the backward solution is the upward closure of some finite set, then there exists a k at which the solutions of the finite k- and k^∞-reduced IFDS instances converge. Though the correctness proof uses some technical machinery, its details are entirely hidden from an implementer, who need only know how to instantiate a standard interprocedural dataflow analysis framework.

We have implemented this algorithm on top of the BLAST interprocedural reachability analysis which is a lazy version of the summary-based interprocedural reachability analysis of [28]. The result is an automatic safety verifier for recursive programs with unboundedly many asynchronous procedure calls. Our reduction technique enables the reuse of optimizations that we have previously found critical for software verification such as on-the-fly exploration, localized refinement [18], and parsimonious abstraction [17]. While we cannot hope for an algorithm that works efficiently for all asynchronous programs (the AIFDS problem is EXPSPACE-hard, in contrast to IFDS which is polynomial time), our initial experiments suggest that in practice the forward reachable state space and the k required for convergence is usually small, making the algorithm practical. In preliminary experiments, we have used our implementation to verify and find bugs in an open source load balancer (`plb`) and a network testing tool (`netchat`). We checked for null pointer errors, buffer overruns, as well as application-specific protocol state properties. In each case, our implementation ran in less than a minute, and converged to a solution with $k = 1$.

```
global request_list *r;

main(){
  ...//setup request list r
  async reqs();
  ...//dispatch loop
}

reqs(){
  if(r==NULL){
    async reqs();
    return;
  }
  rc = malloc(…);
  if (rc == NULL){
    return ABORT_OUT_OF_MEM;
  }
  async client(rc,r->id);
  r = r->next;
  reqs();
}

client(client_t *c, int id){
  ...//setup
  c->id = id;
  ...//continue processing
  return;
}
```

Figure 1. An Example `Plb`

Related Work. Recently, the reachability (and hence, dataflow analysis) problem for asynchronous programs was shown decidable [30], using an algorithm that we believe will be difficult to implement and harder to scale to real systems. First, the algorithm works backwards, thereby missing the opportunities available for optimization by restricting the analysis to the (typically sparse) reachable states that we have found critical for software verification [18]. Second, one crucial step in their proof replaces a recursive synchronous function with an equivalent automaton constructed using Parikh's lemma [26]. Thus, their analysis cannot be performed in an on-the-fly manner: the language-theoretic automaton construction must be performed on the entire exploded graph which can be exponentially large in software verification. Finally, instead of multiset rewriting systems and Parikh's lemma, our proof of completeness relies on counter programs and a version of context free reachability on well quasi-ordered state spaces [10].

Counters [22] have been used to model check concurrent C [16] and Java programs, via a reduction to Petri Nets [7]. However, those algorithms were not interprocedural and did not deal with recursion. Our proof technique of providing a forward abstraction-based algorithm whose correctness is established using a backward algorithm was used in [16] and formalized for a general class of infinite state systems in [13].

Notice that in contrast to the decidability of AIFDS, the dataflow analysis problem for two threads each with recursive synchronous function calls is undecidable [27]. This rules out similar algorithmic techniques to be applied to obtain exact solutions for multithreaded programs, or models in which threads and events are both present.

2. Problem

Figure 1 shows an asynchronous program `Plb` culled from an event-driven load balancer. Execution begins in the procedure `main` which makes an asynchronous call to a procedure (omitted for brevity) that adds requests to the global request list `r`, and makes another asynchronous call to a procedure `reqs` that processes the request list (highlighed by a filled box). The `reqs` procedure checks if `r` is empty, and if so, reschedules itself by asynchronously calling itself. If instead, the list is not empty, it allocates memory for the first request on the list, makes an asynchronous call to `client` which handles the request, and then (synchronously) calls itself (highlighted by the unfilled box) after moving `r` to the rest of the list. The procedure `client` handles individual requests. It takes as input the formal `c` which is a pointer to a `client_t` structure. In the second line of `client` the pointer `c` is dereferenced, and so it is critical that when `client` begins executing, `c` is not null. This is ensured by the check performed in `reqs` before making the asynchronous call to `client`. However, we cannot deduce this by treating asynchronous calls as synchronous calls (and using a standard interprocedural dataflow analysis) as that would additionally conclude the unsound deduction that `r` is also not null when `client` is called.

We shall now formalize the *asynchronous interprocedural finite dataflow analysis (AIFDS) framework*, a generalization of the IFDS framework of [28], solutions of which will enable us to soundly deduce that when `client` begins executing, `c` is non-null, but that `r` may be null.

2.1 Asynchronous Programs

In the AIFDS framework, programs are represented using a generalization of control flow graphs, that include special edges corresponding to asynchronous function calls.

Let P be a finite set of procedure names. An *Asynchronous Control Flow Graph* (ACFG) G_p for a procedure $p \in P$ is a pair (V_p, E_p) where V_p is the set of *control nodes* of the procedure p, including a unique *start node* v_p^s and a unique *exit node* v_p^e, and E_p is a set of directed *intraprocedural edges* between the control nodes V_p, corresponding to one of the following:

- *an operation edge* corresponding to a basic block of assignments or an assume predicate derived from a branch condition,
- *a synchronous call edge* to a procedure $q \in P$, or
- *an asynchronous call edge* to a procedure $q \in P$.

For each directed call edge, synchronous or asynchronous, from v to v' we call the source node v the *call-site* node, and the target node v' the *return-site* node.

EXAMPLE 1: Figure 2 shows the ACFG for the procedures `main`, `reqs` and `client` of the program `Plb`. For each procedure, the start node (resp. exit node) is denoted with a short incoming edge (resp. double circle). The labels on the intraprocedural edges are either operations corresponding to assumes (in box parentheses), and assignments, or asynchronous call edges, shown in filled boxes, *e.g.*, the edge at v_1, or synchronous call edges, shown in unfilled boxes, such as the recursive call edge at node v_9, for which the call-site and return-site are respectively v_9 and v_{10}. □

A *Program* G^* comprises a set of ACFGs G_p for each procedure in $p \in P$. The control locations of G^* are V^*, the union of the control locations of the individual procedures. The edges of G^* are E^*, the union of the (intraprocedural) edges of the individual procedures together with a special set E' of *interprocedural edges* defined as follows. Let *Calls* be the set of (intraprocedural) synchronous call edges in G^*. For each synchronous call edge from call-site v to procedure q returning to return-site v' in *Calls* we have:

- An interprocedural *call-to-start* edge from the call-site v to the start node v_q^s of q, and,
- An interprocedural *exit-to-return* edge from the exit node v_q^e of q to the return-site v'.

As in [28], the call edges (or call-to-return-site edges) allow us to model local variables and parameter passing in our framework.

In Figure 2, the dotted edges correspond to interprocedural edges. The edge from call-site v_9 to the start node v_4 of `reqs` is a call-to-start edge, and the edge from the exit node v_{10} to the return-site v_{10} is an exit-to-return edge.

An *Asynchronous Program* is a program G^* that contains a special *dispatch* procedure `main` (with ACFG G_{main}), which is not called by any other procedure, and that has, for every other procedure, a *self-loop synchronous call edge* from its exit node v_{main}^e to itself. The exit node v_{main}^e is called the *dispatch node*, the self-loop synchronous call edges of the dispatch node are called *dispatch call edges*, the call-to-start edges from the dispatch node are called *dispatch call-to-start edges*, and the exit-to-return edges to the dispatch node are called *dispatch exit-to-return edges*.

Thus, an Asynchronous Program is a classical supergraph of [28] together with special asynchronous call edges, and a special dispatch procedure that has synchronous call edges for each procedure, which are used to model asynchronous dispatch.

EXAMPLE 2: The ACFG for `main` shown in Figure 2 is the dispatch procedure for `Plb`. The exit node v_3, shaded in blue, is the dispatch node with dispatch edges to `reqs` and `client`. The interprocedural edge from v_3 to v_{12} is a dispatch call-to-start edge to `client` and the edge from v_{15} to v_3 is a dispatch exit-to-return edge. □

2.2 Asynchronous Program Paths

Executions of an asynchronous program correspond to paths in the ACFGs. However, not all paths correspond to valid executions. In addition to the standard requirement of interprocedural validity, namely that synchronous calls and returns match up, we require that a dispatch can take place only if there is a pending asynchronous call to the corresponding procedure.

Paths. A *path* of length n from node v to v' is a sequence of edges $\pi = (e_1, \ldots, e_n)$ where v is the source of e_1, v' is the target of e_n, and for each $0 \leq k \leq n-1$, the target of e_k is the source of e_{k+1}. We write $\pi(k)$ to refer to the kth edge of the path π.

Interprocedural Valid Paths. Suppose that each call edge in *Calls* is given a unique index i. For each call edge $i \in$ *Calls* suppose that the call-to-start edge is labeled by the symbol $(_i$ and the exit-to-return edge is labeled by the symbol $)_i$. We say that a path π from v to v' is an *interprocedural valid path* if the sequence of labels on the edges along the path is a string accepted by the following Dyck language, generated by the non-terminal D:

$$\begin{array}{lcll} \mathsf{M} & \rightarrow & \epsilon \mid \mathsf{M}\ (_i\ \mathsf{M}\ )_i & \text{for each } i \in \mathit{Calls} \\ \mathsf{D} & \rightarrow & \mathsf{M} \mid \mathsf{D}\ (_i\ \mathsf{M} & \text{for each } i \in \mathit{Calls} \end{array}$$

We use $IVP(v, v')$ to denote the set of all interprocedural valid paths from v to v'.

Intuitively, M corresponds to the language of perfectly balanced parentheses, which forces the path to match the return edges to the corresponding synchronous call sites, and D allows for some procedures to "remain on the call stack."

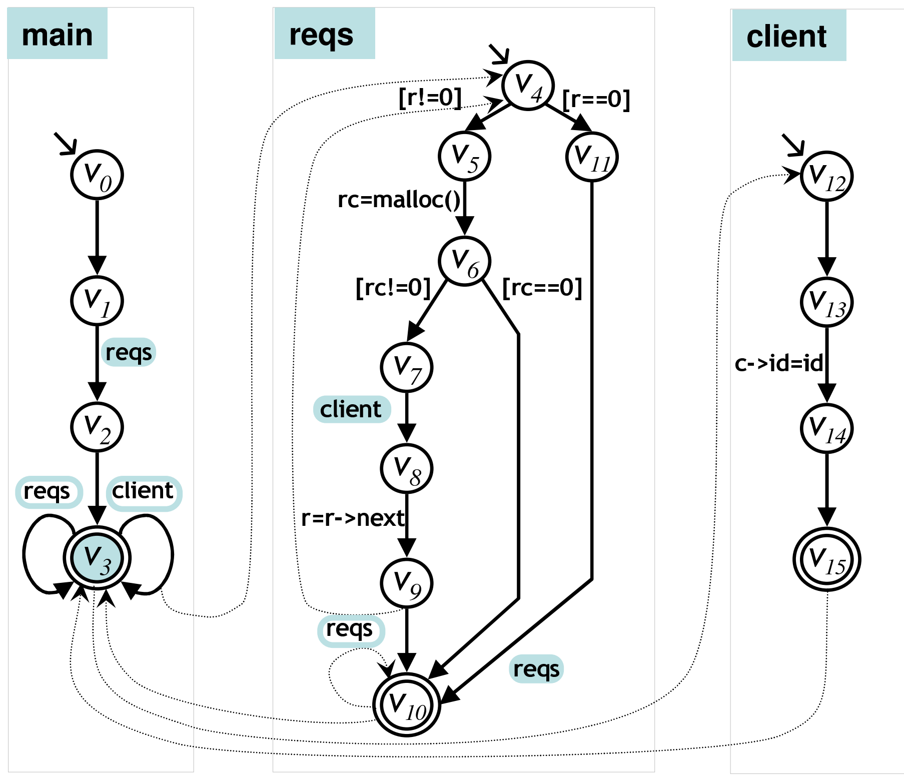

Figure 2. ACFGs for `Plb`

Unlike in synchronous programs, not all Dyck paths correspond to potential executions of the Asynchronous Program, as we have not accounted for asynchronous procedure calls. For example, the path along the edges between nodes v_0,v_1,v_2,v_3,v_{12} of the ACFGs of Figure 2 is a valid interprocedural path, but does not correspond to a valid asynchronous execution as there is no pending asynchronous call to `client` at the dispatch node v_3. To restrict analyses to valid asynchronous executions, we use *schedules* to map dispatch call-to-start edges on paths to matching prior asynchronous call edges.

Schedules. Let π be a path of length n. We say $\sigma : \mathbb{N} \to \mathbb{N}$ is a *schedule* for π iff σ is one-to-one, and for each $0 \leq k \leq n$, if $\pi(k)$ is a *dispatch call-to-start edge* to procedure p, then:

- $0 \leq \sigma(k) < k$, and,
- the edge $\pi(\sigma(k))$ is an asynchronous call to procedure p.

Intuitively, the existence of a schedule implies that at each synchronous "dispatch" of procedure p at step k, there is a pending asynchronous call to p made in the past, namely the one on the $\sigma(k)$-th edge of the path. The one-to-one property of σ ensures that the asynchronous call is dispatched only once. There are no asynchronous executions corresponding to interprocedural paths that have no schedules.

EXAMPLE 3: Figure 3 shows a path of `Plb`, abbreviated to show only the asynchronous call edges and synchronous call-to-start edges. Ignore the boxes with the numbers on the left and the right for the moment. For the prefix comprising all but the last edge, there are two schedules indicated by the arrows on the left and right of the path. Both schedules map the dispatch call-to-start edge 2 to the asynchronous call at edge 1. The left (right) schedule maps the dispatch call-to-start edges $8, 9$ to the asynchronous calls at $5, 3$ respectively ($3, 5$ respectively). If we include the last edge, there is no schedule as there are three dispatch call-to-start edges to `client` but only two asynchronous calls, and so, by the pigeonhole principle there is no one-to-one map. □

2.3 Asynchronous IFDS

An instance of a dataflow analysis problem for asynchronous programs can be specified by fixing a particular asynchronous program, a finite set of *dataflow facts*, and for each edge of the program, a distributive *transfer function* that given the set of facts that hold at the source of the edge, returns the set of facts that hold at the target.

AIFDS Instance. An instance A of an *asynchronous interprocedural finite distributive subset problem* or (AIFDS problem), is a tuple $A = (G^*, D_g, D_l, M, \sqcap)$, where:

1. G^* is an asynchronous program (V^*, E^*),
2. D_g, D_l are finite sets, respectively called global and local dataflow facts – we write D for the product $D_g \times D_l$ which we called the dataflow facts,
3. $M : E^* \to 2^D \to 2^D$ maps each edge of G^* to a distributive dataflow transfer function,
4. $\sqcap$ is the meet operator, which is either set union or intersection.

Unlike the classical formulation for synchronous programs (*e.g.* [28]), the asynchronous setting requires each dataflow fact to be explicitly split into a *global* and a *local* component. This is because at the point where the asynchronous call is made, we wish to capture, in addition to which call was made, the initial input dataflow fact resulting from the passing of parameters to the called procedure. We cannot use a single global set of facts to represent the input configuration, as operations that get executed between the asynchronous

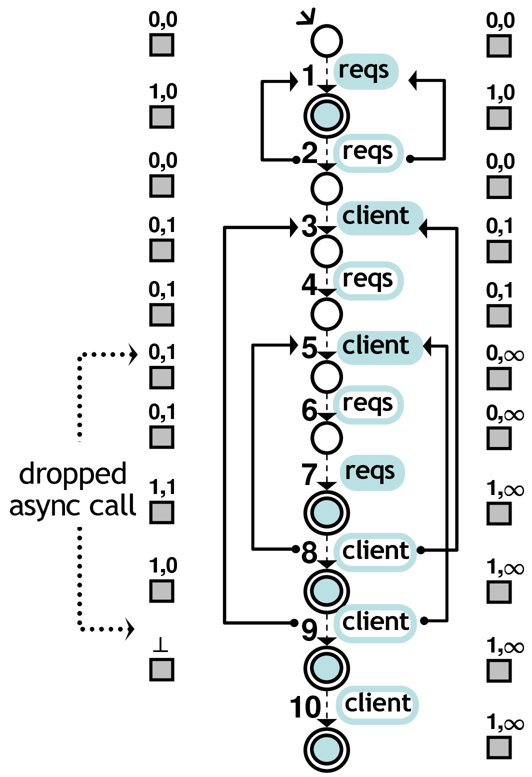

Figure 3. Path showing a sequence of asynchronous posts (in shaded boxes) and synchronous calls (in unfilled boxes). Two different schedules are shown using the arrows from dispatch call-to-start edges to asynchronous call points.

call and the actual dispatch may change the global fact, but not the local fact.

For example, in `Plb` (Figure 1), at the point where the asynchronous call to `client` is made, the global pointer `r` is not null, but this fact no longer holds when `client` begins executing after a subsequent dispatch. However, the local pointer `c` passed via a parameter cannot be changed by intermediate operations, and thus, is still not null when `client` begins executing after a subsequent dispatch.

Thus, our dataflow facts are pairs of global facts D_g and local facts D_l. By separating out global and local facts, when dispatching a pending asynchronous call, we can use the "current" global fact together with the local fact from the asynchronous call to which the schedule maps the dispatch.

EXAMPLE 4: The following is an example of an AIFDS instance. G^* is the asynchronous program of Figure 2, D_g is the set $\{r, \bar{r}\}$ that respectively represent that the global pointer `r` is definitely not null and `r` may be null, and D_l is the set $\{rc, \overline{rc}, c, \bar{c}\}$ that respectively represent that the local pointer `rc` is definitely not null, `rc` may be null, `c` is definitely not null and `c` may be null. We omit the standard transfer functions for these facts for brevity. Thus, the pair $(\bar{r}, c)$ is the dataflow fact representing program states where `r` may be null, but `c` is definitely not null. □

Path Functions. Let $A = (G^*, D_g, D_l, M, \sqcap)$ be an AIFDS instance. Given an interprocedural valid path π, we define a *path relation* $PR(A)(\pi) \subseteq D \times D$ that relates dataflow facts that hold *before* the path to those that hold *after* the operations along the path are executed. Formally, given an interprocedural valid path $\pi = (e_1, \ldots, e_n)$ from v to v' we say that $(d, d') \in PR(A)(\pi)$ if there exists a schedule σ for π and a sequence of data flow facts $d_0, \ldots, d_n$ such that, $d = d_0$, $d' = d_n$ and, for all $1 \leq k \leq n$:

- if e_k is an asynchronous call edge, then $d_k = d_{k-1}$,
- if e_k is a dispatch call-to-start edge, then $d_k = (d_g, d_l)$ where $d_{k-1} = (d_g, \cdot)$ and $(\cdot, d_l) \in M(e_{\sigma(k)})(d_{\sigma(k)})$
- otherwise $d_k \in M(e_k)(d_{k-1})$.

We define the *distributive closure* of a function f as the function: $\lambda S. \cup_{x \in S} f(x)$. The path function is the distributive closure of:

$$PF(A)(\pi) = \lambda d.\{d' \mid (d, d') \in PR(A)(\pi)\}$$

As a path may have multiple schedules, the path relation is defined as the union of the path relation for each possible schedule, which, in turn is defined by appropriately composing the transfer functions for the edges along the path as follows. We directly compose the transfer functions for the edges that are neither asynchronous calls nor dispatch call-to-start edges. We defer applying the transfer function for asynchronous call edges until the matching dispatch call-to-start edge is reached. For each call-to-start edge, we use the given schedule to find the matching asynchronous call edge. The global dataflow fact after the dispatch is the global fact just before the dispatch. The local fact after the dispatch, is obtained by applying the transfer function for the matching asynchronous call edge to the dataflow fact just before the matching asynchronous call was made.

EXAMPLE 5: Figure 4 shows a path of the program `Plb`, together with the dataflow facts obtained by applying the path function on the prefix of the path upto each node. At the start of the first call to `reqs`, the global `r` and the local `rc` may both be null. After the first check, at v_5, we know that `r` is definitely not null, hence the global fact is r. Similarly after the `malloc` and the subsequent check, the local fact at v_7 is rc, *i.e.*, `rc` is not null. After the subsequent assignment to `r`, it may again become null, hence the global fact is $\bar{r}$. Note that at v_7 where the asynchronous call to `client` is made, r holds, but not at v_3 just before the dispatch call to `client`. There is a single schedule for this path, that maps the dispatch edge from v_3 to v_{12} to the asynchronous call edge from v_7 to v_8. Thus, the global fact at v_{12} is the same as at the previous dispatch location, namely $\bar{r}$, that `r` may be null. The local fact at v_{12} is obtained by applying the transfer function of the matching asynchronous call to the dataflow fact (r, rc) that held at the matching asynchronous call site at v_7. As the call passes the local `rc` as the formal `c`, the local fact is c, *i.e.*, `c` is not null. □

AIFDS Solutions. Let $A = (G, D_g, D_l, M, \sqcap)$ be an AIFDS instance. The *meet over all valid paths* (MVP) solution to A is a map $MVP(A) : V^* \to 2^D$, defined as:

$$MVP(A)(v) = \sqcap_{\pi \in IVP(v^s_{main}, v)} PF(A)(\pi)(\top)$$

Thus, given an AIFDS instance A, the problem is to find an algorithm to compute the MVP solution for A.

If a path has no schedule, then its path relation is empty, and so its path function maps all facts to $\perp$. Thus, the MVP solution only takes into account paths that correspond to a valid asynchronous executions.

3. Algorithm

There are two problems that any precise interprocedural analysis for asynchronous programs must solve. First, it must keep track of the unbounded set of pending asynchronous calls in order to only consider valid asynchronous program executions. Second, it must find a way to determine the local dataflow facts corresponding to the input parameters, that hold after a dispatch call-to-start edge.

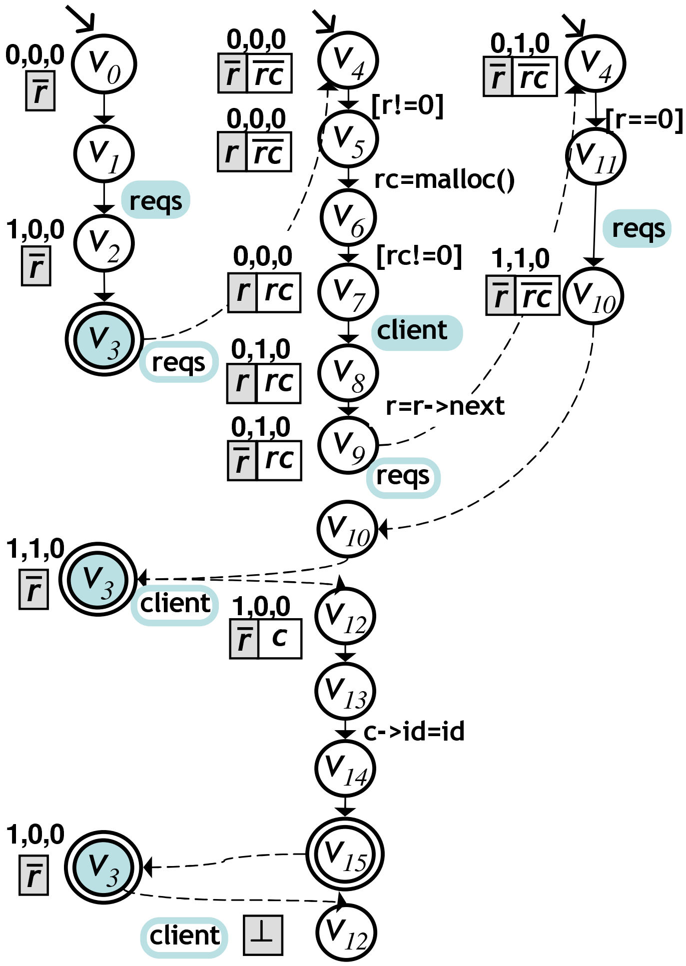

Figure 4. A path of the program `P1b`. The rectangles denote the dataflow facts obtained by applying the path function on the prefix of the paths upto each node. The shaded grey box is the global fact, and the unshaded box the local fact at each point. To reduce clutter, we show the facts at nodes where they differ from the facts at the predecessor.

This is challenging because these local facts are the result of applying the transfer function to the dataflow facts that held at the point when the matching asynchronous call was made, which may be unboundedly far back during the execution.

Our approach to solving both these problems is to reduce an AIFDS instance into a standard IFDS instance by encoding the pending asynchronous calls inside the set of dataflow facts, by taking the product with a new set of facts that *count how many* asynchronous calls to a particular function, with a given input dataflow fact are pending. However, as the pending set is unbounded, this new set of facts is infinite, and so we cannot directly solve the instance. Instead, we *abstractly* count the number of facts, thus yielding a finite instance, and then use the standard IFDS algorithm to obtain a sequence of computable under- and over-approximations of the exact AIFDS solution, which we prove, is guaranteed to converge to the exact solution. We first recall the standard (synchronous) Interprocedural Dataflow Analysis framework and then describe our algorithm.

Solving Synchronous IFDS Instances. A Synchronous Dataflow Analysis problem instance (IFDS [28]) is a tuple $I = (G^*, D, \{\top\}, M, \sqcap)$ that is a special case of an AIFDS instance, where:

1. the program G^* has no asynchronous call edges,
2. there is a single global set of dataflow facts D.

For any valid interprocedural path from v to v' all schedules are trivial as there no dispatch call edges. The MVP solution for an IFDS instance I can be computed by using the algorithm of [28] that we shall refer to as RHS.

THEOREM 1. **[Algorithm RHS [28]]** *For every IFDS instance* $I = (G^*, D, \{\top\}, M, \sqcap)$*, we have* $\mathsf{RHS}(I) = MVP(I)$.

Counters. A *counter* C is a contiguous subset of $\mathbb{N} \cup \{\infty\}$. We assume that $\infty \in C$ whenever the counter C is an infinite subset. For a counter C, and a natural number $n \in \mathbb{N}$, $\max_C(n)$ is n if $n \in C$ and $\max C$ otherwise, and $\min_C(n)$ is n if $n \in C$ and $\min C$ otherwise. For a map f, we write $f[s \mapsto v]$ for the new map:

$$\lambda x.\ \text{if } x = s \text{ then } v \text{ else } f(x)$$

A *counter map* f is a map from some set S to a counter C. For any $s \in S$, we write $f +_C s$ for the counter map:

$$f[s \mapsto \max_C(f(s) + 1)]$$

and we write $f -_C s$ for the map:

$$f[s \mapsto \min_C(f(s) - 1)]$$

Note that both $f +_C s$ and $f -_C s$ are maps from S to C. Intuitively, we think of $f +_C s$ (resp. $f -_C s$) as "adding" (resp. "removing") an s to (resp. from) f. We define the counter C_∞ as the set $\mathbb{N} \cup \{\infty\}$, and for any $k \geq 0$, the counter C_k as $\{0, \ldots, k\}$, and the counter C_k^∞ as $\{0, \ldots, k, \infty\}$. We write c_0 for the counter map $\lambda s.0$. A C_∞ counter map tracks the exact number of s in f. A C_k counter map tracks the exact number of s in f upto a maximum value of k, at which point it "ignores" subsequent additions. A C_k^∞ counter map tracks the exact number of s in f upto a maximum of k after which a subsequent increment results in the map getting updated to ∞, which remains, regardless of the number of subsequent removals.

3.1 Algorithm ADFA

We now present our Algorithm ADFA for computing the MVP solution of AIFDS instances. The key step of the algorithm is the use of counter maps to encode the set of pending asynchronous calls inside the dataflow facts, and thereby converting an AIFDS instance into an IFDS instance.

Given an AIFDS instance $A = (G^*, D_g, D_l, M, \sqcap)$, and a counter C we define the C-reduced IFDS instance as the tuple $(G_C^*, D_C, \{\top\}, M_C, \sqcap_C)$ where:

- G_C^* is obtained by replacing each asynchronous call edge in G^* with a fresh trivial operation edge between the same source and target node,
- D_C is the set $(D_g \times D_l) \times (P \times D_l \to C)$. The elements of the set are pairs (d, c) where d is a dataflow fact in $D_g \times D_l$ and c is a counter map that tracks, for each pair of asynchronous call and input dataflow fact, the *number of such calls that are pending*.
- M_C is defined on the new dataflow facts and edges as follows.
  - if e is an asynchronous call edge to p in G^* then $M_C(e)(d, c) = \{(d, c +_C (p, d_l')) \mid (\cdot, d_l') \in M(e)(d)\}$
  - if e is a dispatch call to start edge to p in G^* then $M_C(e)(d, c) = \{((d_g, d_l'), c -_C (p, d_l')) \mid c(p, d_l') > 0,\ \ d = (d_g, \cdot)\}$

- otherwise $M_C(e)(d, c) = \{(d', c) \mid d' \in (M(e)(d)\}$.

- $\sqcap_C$ is the union (resp. intersection) operation if $\sqcap$ is the union (resp. intersection) operation.

Intuitively, the reduced transfer function for an asynchronous call "adds" the pair of the called procedure and the initial local dataflow fact to the counter map. For a dispatch call-to-start edge to procedure p, the transfer function returns the set of tuples of the current global dataflow fact together with those local facts d_l for which the counter map of (p, d_l) is positive, together with the countermaps where the pairs (p, d_l) have been removed. If for all pairs $(p, \cdot)$ the counter map value is zero, then the transfer function returns the empty set, *i.e.* $\perp$.

EXAMPLE 6: Figure 4 shows a path of the C_∞-reduced instances of `Plb`. On the left of each (intraprocedural) path, we show the dataflow facts resulting from applying the path function to the prefix of the path upto each corresponding node. The shaded box contains the global dataflow fact, the white box the local fact, and the numbers i, j, k on top represent the counter map values for $(\texttt{reqs}, \top)$, $(\texttt{client}, c)$, and $(\texttt{client}, \bar{c})$ respectively. For all other pairs, the counter map is always zero. Note that the value for $(\texttt{reqs}, \top)$ increases after the asynchronous call at v_1, decreases after the dispatch at v_3 and again increases after the asynchronous call at v_{11}. At the second occurrence of v_3 (the dispatch location), $(\texttt{client}, c)$ is the only pair with `client` as the first parameter, for which the counter map value is positive. Thus, after the dispatch, the dataflow fact is the pair of the global $\bar{r}$ from the dispatch location and the local c from the counter map. □

Our first observation is that the MVP solution of the C_∞-reduced instance is equivalent to the MVP solution of the original AIFDS instance. This is because the C_∞-reduced instance *exactly* encodes the unbounded number of pending asynchronous call and initial local fact pairs within the counter maps of the dataflow facts. Thus, for any interprocedural valid path the (reduced) path function returns the union of the set of dataflow facts resulting from *every* possible schedule.

For two sets $s \subseteq B \times D$ and $s' \subseteq B \times D$, we say that $s \dot{=} s'$ (resp. $s \dot{\subseteq} s'$) if $\{b \mid (b, \cdot) \in s\}$ is equal to (resp. included in) the $\{b' \mid (b', \cdot) \in s'\}$. For two functions $f : A \to 2^{B \times D}$ and $f' : A \to 2^{B \times D'}$, we say $f \dot{=} g$ (resp. $f \dot{\subseteq} g$) if for all x, the set $f(x) \dot{=} f'(x)$ (resp. $f(x) \dot{\subseteq} f'(x)$).

THEOREM 2. **[Counter Reduction]** *For every AIFDS instance A, if I is the C_∞-reduced instance of A, then $MVP(I) \dot{=} MVP(A)$.*

Unfortunately, this reduction does not directly yield an algorithm for solving AIFDS instances, as the C_∞-reduced instance has infinitely many dataflow facts, due to the infinite number of possible counter maps.

Our second observation is that we can generate finite IFDS instances that approximate the C_∞-reduced instance and thus, the original AIFDS instance. In particular, for any k, the C_k-reduced and C_k^∞ instances are, respectively, an *under-approximation* and an *over-approximation* of the C_∞-instance.

In the C_k-reduced IFDS instance, the path function returns $\perp$ for any path along which there are $k + 1$ (or more) successive dispatches to some function starting with some given local fact. This happens as because the number of tracked pending calls never rises above k, after the k successive dispatches, the map value must be zero, thus the $k + 1$-th call yields a $\perp$. Thus, the MVP solution for the C_k-reduced instance is an underapproximation of the exact AIFDS solution that includes exactly those paths along which there are at most k successive dispatches to a particular procedure with a given local fact.

Dually, in the C_k^∞-reduced IFDS instance, once a $k + 1$-th pending call is added for some procedure, the counter map is updated to ∞ (instead of $k+1$). As a result, from this point on, it is *always* possible to dispatch a call to this procedure. Thus, the MVP solution for the C_k^∞-reduced instance is an over-approximation of the exact AIFDS solution that includes all the valid paths of the AIFDS instance, and also extra paths corresponding to those executions where at some point there were more than k pending calls to some procedure.

EXAMPLE 7: Figure 3 illustrates how the C_1-reduced instance and the C_1^∞-reduced instance are respectively under- and over-approximations of the C_∞-reduced IFDS instance of `Plb`. Suppose that D_g and D_l are singleton sets containing $\top$. On the left and right we show the sequence of dataflow facts obtained by applying the path functions for the C_1 and C_1^∞ respectively, on the prefix of the operations upto that point on the path. The numbers i, j above the boxes indicate the counter map value for $(\texttt{reqs}, \top)$ and $(\texttt{client}, \top)$ respectively. As each asynchronous call is made, the counter map for the corresponding call is updated, and for each dispatch call, the value is decremented.

In the C_1-reduced instance (left), the second asynchronous call to `client` is dropped, *i.e.*, the counter is not increased above 1, and thus, the second dispatch to `client` results in $\perp$. Thus, the effect of this path is not included in the (under-approximate) MVP solution for the C_1-reduced instance. In the C_1^∞-reduced instance (right), the second asynchronous call results in the counter for `client` is increased to ∞. Thus, in this instance, the second dispatch to `client` yields a non-$\perp$ dataflow fact. Moreover, *any* subsequent dispatch yields a non-$\perp$ value, all of which get included in the (over-approximate) MVP solution for the IFDS instance. □

THEOREM 3. **[Soundness]** *For every AIFDS instance A, for every $k \geq 0$, if I, I_k, I_k^∞ are respectively the C_∞-reduced, C_k-reduced and C_k^∞-reduced IFDS instances of A, then:*

(a) $MVP(I_k) \dot{\subseteq} MVP(I) \dot{\subseteq} MVP(I_k^\infty)$

(b) $MVP(I_k) \dot{\subseteq} MVP(I_{k+1})$

(c) $MVP(I_{k+1}^\infty) \dot{\subseteq} MVP(I_k^\infty)$

The proof of the soundness Theorem 3, follows by observing that the C_k- (resp. C_k^∞-) instance effectively only considers a subset (resp. superset) of all the valid asynchronous executions, and for each path for which both the AIFDS path function and the reduced instance's path function return a non-$\perp$ value, the value's returned by the two are identical.

As for each k, the counters C_k and C_k^∞ are finite, we can use RHS to compute the MVP solutions for the finite IFDS instances I_k and I_k^∞, thereby computing under- and over- approximations of the MVP solution for the AIFDS instance.

Our algorithm ADFA (shown in Algorithm 1) for computing the MVP solution for an AIFDS instance A is to compute successively more precise under- and over-approximations. An immediate corollary of the soundness theorem is that if we find some k for which the under- and over-approximations coincide, then the approximations are equivalent to the solution for the C_∞-reduced instance, and hence, the exact MVP solution for A. The next theorem states that for every AIFDS instance, there exists a k for which the under- and over-approximations coincide, and therefore, the Algorithm ADFA is guaranteed to terminate.

THEOREM 4. **[Completeness]** *For each AIFDS instance A there exists a k such that, if I_k and I_k^∞ are respectively the C_k- and C_k^∞-reduced IFDS instances of A, then $MVP(I_k) \dot{=} MVP(I_k^\infty)$*

This Theorem follows from the following lemma.

Algorithm 1 Algorithm ADFA

Input: AIFDS instance A
Output: MVP solution for A
$k = 0$
repeat
  $k = k + 1$
  $I_k = C_k$-reduced IFDS instance of A
  $I_k^\infty = C_k^\infty$-reduced IFDS instance of A
until $\mathsf{RHS}(I_k) \doteq \mathsf{RHS}(I_k^\infty)$
return $\mathsf{RHS}(I_k)$

Algorithm 2 Algorithm DemADFA

Input: AIFDS instance A, Error node $v_\mathcal{E}$
Output: SAFE or UNSAFE
$k = 0$
loop
  $k = k + 1$
  $I_k = C_k$-reduced IFDS instance of A
  $I_k^\infty = C_k^\infty$-reduced IFDS instance of A
  if $\mathsf{DemRHS}(I_k)(v_\mathcal{E}) \neq \perp$ **then return** TRUE
  if $\mathsf{DemRHS}(I_k^\infty)(v_\mathcal{E}) = \perp$ **then return** FALSE

LEMMA 1. **[Pointwise Completeness]** *Let $A = (G^*, D_g, D_l, M, \sqcap)$ be an AIFDS instance, and I be the C_∞-reduced IFDS instance of A. For every $d \in D_g \times D_l$ and $v \in V^*$, there exists a $k_{d,v} \in \mathbb{N}$ such that for all $k \geq k_{d,v}$, $\exists c_k$ s.t. $(d, c_k) \in MVP(I_k)(v)$ iff $\exists c$ s.t. $(d, c) \in MVP(I)(v)$ iff $\exists c_k^\infty$ s.t. $(d, c_k^\infty) \in MVP(I_k^\infty)(v)$.*

To prove Theorem 4 we pick any k greater than $\max_{d,v} k_{d,v}$ (this is well defined since D and V^* are finite sets). Thus, the crux of our completeness result is the proof of Lemma 1 which we postpone to Section 5.

THEOREM 5. **[Correctness of ADFA]** *For every AIFDS instance A, Algorithm ADFA returns $MVP(A)$.*

The proof follows from Theorems 1,3,4.

3.2 Demand-driven AIFDS Algorithm

We now present an algorithm for solving a Demand-AIFDS problem. This algorithm works by invoking a standard Demand-IFDS Algorithm on C_k- and C_k^∞- reduced IFDS instances of the AIFDS instance.

Demand-AIFDS Instance. An instance A of a *Demand AIFDS* problem is a pair $(A, v_\mathcal{E})$ where A is an AIFDS instance, and $v_\mathcal{E}$ is a special query node of the supergraph of A. Given a Demand AIFDS instance, the *Demand-AIFDS problem* is to determine whether $MVP(A)(v_\mathcal{E}) \neq \perp$.

Demand-IFDS and DemRHS. We define a Demand-IFDS Instance as an AIFDS instance $(I, v_\mathcal{E})$ where I is an IFDS instance. Let DemRHS be a Demand-IFDS Algorithm such that $\mathsf{DemRHS}(I, v_\mathcal{E})$ returns TRUE iff $MVP(I)(v_\mathcal{E}) \neq \perp$.

To solve a Demand-AIFDS problem, we use C_k- and C_k^∞-reduced under- and over-approximations as before. Only, instead of increasing k until the under- and over-approximations coincide, we increase it until either:

1. in the under-approximation (*i.e.*, the C_k-reduced IFDS instance), the MVP solution is not $\perp$, in which case we can deduce from Theorem 3 that the exact AIFDS solution is also not $\perp$, or dually,
2. in the over-approximation (*i.e.*, the C_k^∞-reduced IFDS instance), the MVP solution is $\perp$, in which case we deduce from Theorem 3 that the exact AIFDS solution is also $\perp$.

The completeness theorem guarantees that this demand-driven algorithm DemADFA (summarized in Figure 2) terminates.

THEOREM 6. **[Correctness of DemADFA]** *For each Demand-AIFDS instance $(A, v_\mathcal{E})$, DemADFA terminates and returns TRUE if $MVP_A(v_\mathcal{E}) \neq \perp$ and FALSE otherwise.*

Though we would have liked polynomial time algorithms for solving AIFDS and Demand-AIFDS problems, the following result (also in [30]), that follows by reduction from reachability of structured counter programs [11], shows that this is impossible.

THEOREM 7. **[EXPSPACE-Hardness]** *The Demand-AIFDS problem is EXPSPACE-hard, even when there are no recursive synchronous calls.*

3.3 Optimizations

We now describe two general optimizations that can be applied to any AIFDS instance that reduce the number of states explored by the analysis.

1. Effective Counting The first optimization is based on two observations. First, the dispatch node is the only node where the counter maps are "read" (have any effect on the transfer function). At other nodes, the counter map is either added to (for some asynchronous calls), or copied over. Thus, rather than exactly propagating the counter maps in the dataflow facts, we need only to *summarize the effect* of a (synchronous) dispatch on the counter map, and use the summaries to update the counter maps after each dispatch call returns to the dispatch location. Second, between the time a dispatch call begins and the time it returns, the counter map values *only increase* due to asynchronous calls that may happen in the course of the dispatch.

Thus, we summarize the effect of a dispatch on the counter map as follows. Suppose that the counter map at a (synchronous) callsite is c. For a call-to-start edge to procedure p, for each entry dataflow fact for p, we *reset* the counter map to c_0 (all zeros) and only compute the dataflow facts reachable from such reset configurations. For each summary edge [28] for p with the target counter map c', we propagate the summary edge at the callsite, by updating the counter map to: $\lambda x.\max_C(c(x) + c'(x))$, where C is the counter being used in the reduced instance. The saving from this optimization is that for each procedure, for each entry dataflow fact, we only compute summaries starting from the single reset counter map c_0, rather than upto $|C|^{|D_l||P|}$ distinct counter maps.

2. Counter Map Covering The second optimization follows from observing that there is a partial order between the counter maps. For two counter maps c, c', we say that $c \leq c'$ if for all s, we have $c(s) \leq c'(s)$. It is easy to check that if $c \leq c'$, then for any instance I, for all paths π, for all dataflow facts $d \in D_g \times D_l$, the $PF(I)(\pi)(d, c) \dot{\subseteq} PF(I)(\pi)(d, c')$. This implies that we only need to maintain maximal elements in this ordering. Thus, the set of facts reachable from c is *covered* by the facts reachable from c', and so in our implementation of RHS, when we find two instances of the dispatch location in the worklist, with facts (d, c) and (d, c') with $c \leq c'$, we drop the former instance from the worklist.

4. Application: Safety Verification

We now describe how the ADFA algorithm can be applied to the task of *safety verification*, *i.e.*, determining whether in a given asynchronous program, some user-specified error location $v_\mathcal{E}$ is reachable.

EXAMPLE 8: Figure 5 shows a typical idiom in asynchronous programs where different clients attempt to write files to a device.

```
global struct device dev;
const int k>0;
```

```
main(){
  dev.owner = 0;
  socket = create_socket();
  async listen(socket);
}
```

```
listen(int socket){
  gid = new_gid();
  if (gid > 0){
    async new_client(gid);
  }
  async listen(socket);
  return;
}
```

```
new_client(int gid){
  if (dev.owner > 0){
    async new_client(gid);
  } else {
     dev.owner = gid;
     fd=file_descript(gid);
     async write(gid,fd);
  }
}
```

```
write(int id, int fd){
  if(id==k&& dev.owner!=k) ERR:
  if (transfer(fd,dev)){
    async write(id,fd);
    return;
  } // else, write complete
  dev.owner = 0;
  fd = file_descriptor(id);
  free_gid(id,fd);
}
```

Figure 5. Example Race

The main function spawns an asynchronous listen procedure that is nondeterministically called every time a new client joins on a socket. The procedure then calls new_client with a unique gid or "group id" [6] which processes the request of the individual clients. A critical mutual exclusion property in such programs is that once a client, represented by its gid, has "acquired" and thus begun writing to the device, no other client should be given access until the first client is finished. To ensure mutual exclusion, many asynchronous programs use state-based mechanisms like that in Race. The device is stamped with an owner field that tracks the last gid that wrote to the device, and a client is granted access if the owner field is 0, indicating there is no current client writing to the device. To verify the mutual exclusion, we encode the property as an assertion by creating a (skolem) constant k that represents some arbitrary client id, and checking the assertion that whenever the device is written to in write, that the id of the writer is k, then the owner of the device is also k. Thus, the program satisfies the mutual exclusion property iff the error location corresponding to the label ERR is not reachable. □

To perform safety verification, we instantiate the general AIFDS framework with dataflow facts and transfer functions derived via predicate abstraction [2, 14]. The result is a Demand AIFDS instance that we solve using the DemADFA algorithm. If the MVP solution for the error node is $\perp$, then we can deduce that the error location is not reachable. If the solution is not $\perp$, then either the error location is reachable, or the set of predicates is too imprecise, and we automatically learn new predicates from the infeasible counterexample whose path function is not $\perp$, using the technique of [17]. We then repeat the verification with the new predicates, until we find an execution that reaches the error location, or the location is proven to be unreachable [4, 3, 18]. We now describe how to generate Demand AIFDS instances for a safety verification problem by describing the corresponding AIFDS instances.

4.1 Predicate Abstraction AIFDS Instances

A Predicate Abstraction AIFDS instance is a tuple $A = (G^*, D_g, D_l, M, \sqcap)$, where:

- G^* is an asynchronous program,
- D_g is a finite set of *global* predicates, *i.e.*, predicates over the global program variables,
- D_l is a finite set of *local* predicates, *i.e.*, predicates over the local program variables,
- $M(e)$ is the defined as the distributive closure of:

  $\lambda(d_g, d_l).\ \{(d'_g, d'_l) | sp(e, d_g \wedge d_l) \wedge d'_g \wedge d'_l$ is satisfiable $\}$

  where $sp(e, \varphi)$ is the *strongest postcondition* [9] of φ w.r.t. the operation e,
- $\sqcap$ is the set union operator.

This is slightly different from the standard formulation of predicate abstraction [14], where the elements of D_g and D_l are all the possible cubes over some set of atomic predicates.

We can generate an AIFDS instance $A_{\texttt{Race}}$ for the safety verification problem for Race as follows. The set of global predicates is $\{ow = 0, ow > 0 \wedge ow = k, ow > 0 \wedge ow \neq k\}$, where ow is an abbreviation for dev.owner, and the set of local predicates is $\{gid > 0, gid = k, gid \neq k, id = k, id \neq k\}$. With these predicates, for example, the transfer function for the edge dev.owner $= 0$ is the distributive closure of $\lambda(d_g, d_l).(ow = 0, d_l)$, *i.e.*, the global predicate becomes $ow = 0$ and the local predicate remains unchanged.

Figure 6 shows the result of the optimized Demand IFDS analysis for the C_1^∞ reduced IFDS instance of $A_{\texttt{Race}}$. The grey box contains the global predicate and the white box the local predicate. The numbers i,j,k,l above the boxes correspond to the counter map values for (listen, $\top$),(new_client, $gid > 0$),(write, $id = k$) and (write, $id \neq k$) respectively.

Execution begins in main, with no pending asynchronous calls, and proceeds to the dispatch location where the global predicate $ow = 0$ holds, and the only pending call is to listen. We analyze listen, the only pending call from the dispatch call site 1, from the counter map mapping all predicates to zeros (p stands for any of the global predicates). The exploded supergraph for listen shows that an execution of listen preserves the global dataflow fact, makes an asynchronous call to listen, and may, if the call to new_gid is successful (*i.e.*, returns a positive gid), make an asynchronous call to new_client with a positive argument. We plug in the summary edges from listen into the dispatch call site 1 – the results are shown with the dotted edges labeled L.

For each generated (*i.e.*, "exploded") instance of the dispatch call site, we compute the results of dispatching each possible pending asynchronous call (together with the input dataflow fact). Thus, at the dispatch call site instance 2, there are pending calls to listen and new_client. Plugging in the summaries for listen, we deduce that the result is either a self loop back to dispatch call site 2, or, if another asynchronous call to new_client is made, then a dotted summary edge to dispatch callsite 3 where there are ∞ calls pending on new_client because the actual value 2 obtained by adding the *effect* 1 to the previous counter map value at the call site gets abstracted to ∞ in the C_1^∞ reduction.

Similarly, we plug in the summaries for new_client and write (shown in the respective boxes), for each of the finitely many dispatch call site instances, resulting in the successors corresponding to dotted edges labeled N and W respectively. The call site instances 3, 5 are *covered* by the instances 6, 7 respectively, and so we do not analyze the effects of dispatches from 3, 5.

Notice that new_client is always called with a positive argument, and that write is only called either when both id and owner are equal to k or when neither is equal to k, and so the mutual exclusion property holds.

4.2 Experiences

We have implemented the DemADFA algorithm along with these optimizations in BLAST[18], obtaining a safety verification tool for recursive programs with asynchronous procedure calls. In our experiments, we checked several safety properties of two event driven programs written using the LIBEEL event library [6]. These programs were ported to LIBEEL from corresponding LIBEVENT pro-

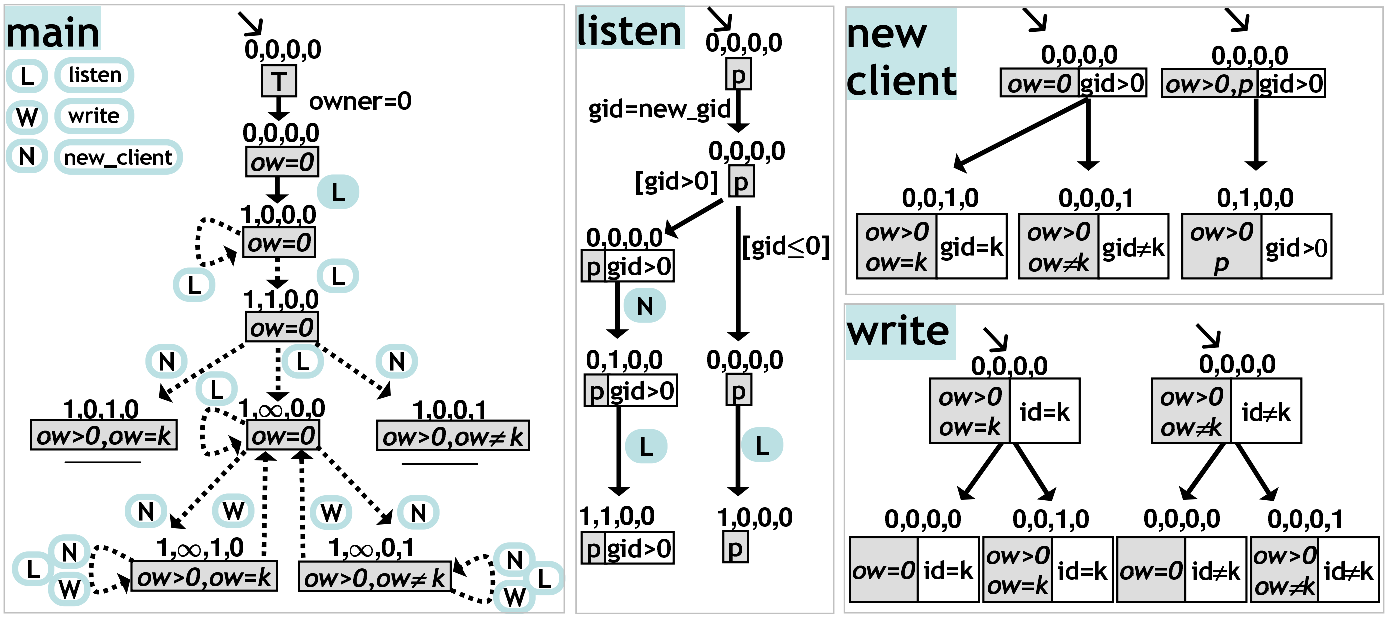

Figure 6. Summaries for `Race`

grams, and are available from the LIBEVENT web page [21]. `plb` is a high performance load balancer (appx 4700 lines), and `nch` is a network testing tool (appx 684 lines). We abstract the event registration interface of LIBEEL in the following way. We assume that external events can occur in any order, and thus, the registered callbacks can be executed in any order. In particular, this means that we abstract out the actual times for callbacks that are fired on some timeout event. With this abstraction, each event registration call in LIBEEL becomes an asynchronous call that posts the callback. While the predicate discovery procedure implemented in BLAST[17] is not guaranteed to find well-scoped predicates in the presence of asynchronous programs, we use it heuristically, and it does produce well-scoped predicates in our examples. We think a predicate discovery algorithm that takes asynchronous calls into account is an interesting open research direction.

Null Pointer. The first property checks correctness of pointer dereferences in the two benchmarks. For each callback, we insert an assertion that states that the argument pointer passed into the callback is non-null. Usually, this is ensured by a check on the argument in a caller up the asynchronous call chain. Hence, the correctness depends on tracking program flows across asynchronous as well as synchronous calls. The results are shown in Table 1. There are 4 instances of these checks for `plb`, namely, `plb-1` through `plb-4` and 2 for `nch`. The instances for `plb` are all safe. There is a bug in one of the checks in `nch` where the programmer forgets to check the result of an allocation. All the runs take a few seconds. In each example, we manually provide the predicates from the assertions to BLAST, but additional predicates are found through counterexample analysis.

Protocol State. `plb` maintains an internal protocol state for each connection. The protocol state for an invalid connection is 0, on connection, the state is 1, and the state moves to 2 and then 3 when certain operations are performed. These operations are dispatched from a generic callback that gets a connection and decides which operation to call based on the state. It is an error to send an invalid connection (whose state is 0) to this dispatcher. We checked the assertion that the dispatcher never receives a connection in an invalid state (file `plb-5`). We found a bug in `plb` that showed this property could be violated. The bug occurs if the client sends a too large request to read, in which case the connection is closed and the state reset to 0. However, the programmer forgot to return at this point in the error path. Instead, control continues and the next callback in the sequence is posted, which calls the dispatcher with an invalid connection.

Buffer Overflow. Each connection in `plb` maintains two integer fields: one tracks the size of a buffer (the number of bytes to write), and the second tracks the number of bytes already written. The second field is incremented on every write operation until the required number of bytes is written. We check that the second field is always less than or equal to the first (file `plb-6`). The complication is that the system `write` operation may not write all the bytes in one go, so the callback reschedules itself if the entire buffer is not written. Hence the correctness of the property depends on data flow through asynchronous calls. BLAST can verify that this property holds for the program. We model the `write` procedure to non-deterministically return a number of bytes between 0 and the number-of-bytes argument.

Our initial experiences highlight two facts. First, even though the algorithm is exponential space in the worst case, in practice, the reachable state space as well as the counter value required for convergence is small (in all experiments $k = 1$ was sufficient). Second, the correctness of these programs depends on complicated dataflow through the asynchronous calls: this is shown by the number of distinct global states reached at the dispatch location.

5. Proof

The foundation on which our technique for solving AIFDS is based is that the C_k- and C_k^∞-reduced under- and over-approximations actually converge to the C_∞-reduced instance, and therefore to the precise MVP solution of the AIFDS instance. The main technical challenge is to prove the completeness Theorem 4, which, as outlined earlier, proceeds from the proof of Lemma 1. We prove

Program	Time	Preds	Total	Dispatch
`plb-1`	3.05	7	2047	30
`plb-2`	4.10	16	1488	18
`plb-3`	7.05	20	1583	20
`plb-4`	5.290	14	1486	25
`nch-1`	1.32	4	521	27
`nch-2(*)`	0.440	0	-	-
`plb-5(*)`	30.20	55	-	-
`plb-6`	22.68	41	1628	22

Table 1. Experimental results. **Time** measures total time in seconds. **Preds** is the total number of atomic predicates used. **Total** state is the total number of reachable "exploded" nodes. **Dispatch** is the number of reachable "exploded" dispatch nodes. `(*)` indicates the analysis found a violation of the specification.

Lemma 1 in two steps. In the first step (Lemma 3), we show that the backward solution of the C_∞-reduced IFDS instance is equivalent to the upward closure of some finite number of facts. In the second step (Lemma 4), we show how, from the finite set of facts, we can find a k such that the C_k^∞-reduced instance coincides with the C_∞-reduced solution.

Upward Closure. For two counter maps c, c' from S to C, we write $c \leq c'$ if for all $s \in S$, we have $c(s) \leq c'(s)$. Let $D_g \times D_l$ be a finite set of dataflow facts. The *upward closure* of a set $B \subseteq (D_g \times D_l) \times (D_l \to C)$ is the set

$$B^{\leq} = \{(d, c') \mid \exists (d, c) \in B \text{ s.t. } c \leq c'\}$$

We say B is *upward closed* if $B = B^{\leq}$.

The ordering $\leq$ on counter maps with a finite domain is a *well quasi-order*, that is, it has the property that any infinite sequence $c_1, c_2, \ldots$ of counter maps must have two positions i and j with $i < j$ such that $c_i \leq c_j$ [8]. We shall use the following fact about well quasi-orderings.

LEMMA 2. *[1] Let f be a function from counter maps to counter maps that is monotonic w.r.t. $\leq$. Let $(f^{-1})^*$ denote the reflexive transitive closure of the inverse of f. For any upward closed set U of counter maps, there is a finite set B of counter maps such that $B^{\leq} = (f^{-1})^*(U)$.*

Backward Solutions. For an IFDS instance $I = (G^*, D, \{\top\}, M, \sqcap)$ (where D may be infinite), for any $v' \in V^*$ and $d \in D$, we define the *backwards meet over valid paths* solution $MVP^{-1}(I, v', d')(v)$ as:

$$\{d \mid \exists \pi \in IVP(v, v') \text{ s.t. } (d, d') \in PR(I)(\pi)\}$$

Intuitively, the backwards or inverse solution for v', d' is the set of facts at v, which get "transferred" along some valid path to the fact d' at v'. If D is finite, we can compute the inverse solution using a backwards version of the RHS algorithm. It turns out, that if D corresponds to the infinite set of facts for a C_∞-reduced instance of an AIFDS, then the infinite inverse solution is equivalent to the upward closure of a finite set. Recall that c_0 is the map $\lambda x.0$.

LEMMA 3. *[30, 10] Let $A = (G^*, D_g, D_l, M, \sqcap)$ be an AIFDS instance, and I be the C_∞-reduced IFDS instance of A. For every $v' \in V^*$ and $d' \in D_g \times D_l$, there exists a finite set $B(v', d', v) \subseteq (D_g \times D_l) \times (P \times D_l \to \mathbb{N})$ such that: $B(v', d', v)^{\leq} = MVP^{-1}(I, v', (d', c_0))(v)$.*

PROOF. *Sketch.* The proof relies on two facts: first, $\leq$ forms a well quasi-order on $(D_g \times D_l) \times (P \times D_l \to \mathbb{N})$, and second, that the dataflow facts and transfer function for A is a monotonic function on this order. Intuitively, the transfer function is not "inhibited" by adding elements to the counter map. With these in mind, and using Lemma 2, we can devise a backwards RHS algorithm whose termination (shown in [10]) is guaranteed by the well quasi-ordering of $(D_g \times D_l) \times (P \times D_l \to \mathbb{N})$ [8, 1] and guarantees the existence of the finite set $B(v', d', v)$. The backward RHS algorithm propagates dataflow facts backward and creates summaries from return points to corresponding call points. ∎

An alternate proof of the above result is obtained following the proof in [30] which reduces the AIFDS instance via Parikh's lemma to a multiset rewriting system and uses well quasi-ordering arguments on multisets to guarantee termination. Parikh's lemma is used to replace the original program that may have recursive calls with an automaton which has the same effect w.r.t. counter maps.

In the the second step, we show that from the set $B(v', d', v)$, we can obtain a k, that suffices to prove the Completeness Lemma 1.

Maxcount. Let $A = (G^*, D_g, D_l, M, \sqcap)$ be an AIFDS instance, and let I be the C_∞-reduced (infinite) IFDS instance of A. We define the *maxcount* of A, as:

$$1 + \max \bigcup_{d', v', v} \{c(s) \mid (d, c) \in B(v', (d', c_0), v), s \in S\}$$

Note that as d', v', v range over finite sets $D_g \times D_l$ and V^* respectively, and from Lemma 3 $B(v', (d', c_0), v)$ is finite, the maxcount of A is also finite.

We observe that if k is the *maxcount* of the AIFDS instance, then if the fact d' is not in the MVP solution for v' in the C_∞-reduced instance then it is not in the solution of the finite C_k^∞-reduced IFDS instance.

LEMMA 4. *Let $A = (G^*, D_g, D_l, M, \sqcap)$ be an AIFDS instance, with maxcount k, and I (resp. I_k^∞) be the C_∞- (resp. C_k^∞-) reduced IFDS instances of A. For every $v' \in V^*$ and $d' \in D_g \times D_l$,*

(a) if $(\top, c_0) \notin MVP^{-1}(I, v', (d', c_0))(v^s_{main})$ then for all $v \in V^*$, $MVP(I_k^\infty)(v) \cap MVP^{-1}(I, v', (d', c_0))(v) = \emptyset$

(b) if $\nexists c'.\ (d', c') \in MVP(I)(v')$ then $\nexists c'.\ (d', c') \in MVP(I_k^\infty)(v')$.

PROOF. First, note that (b) follows from (a) by observing from the definitions of solutions and backwards solutions that there exists a c' such that $(d', c') \in MVP(I)(v')$ iff $(\top, c_0) \in MVP^{-1}(I, v', d')$, then instantiating the universal quantifier in (a) with v', and finally applying the fact that $MVP^{-1}(I, v', (d', c_0))$ is upward closed (from Lemma 3). Next, we prove the following statement which implies (a).

IH $\forall n \in \mathbb{N}, v \in V^*, \forall \pi \in IVP(v^s_{main}, v)$ of length n, if $(d, c) \in PF(I_k^\infty)(\pi)(\top, c_0)$ then $(d, c) \notin MVP^{-1}(I, v', (d', c_0))(v)$.

The proof is by induction on n. The base case follows from the hypothesis that $(\top, c_0)$ is not in the backwards solution for $v', (d', c_0)$.

For the induction step, suppose the **IH** holds upto n. Consider a path $\pi = \pi'', (v'', v)$ of length $n + 1$ where the prefix π'' is of length n. By the definition of the path function, we know there exists a (d'', c'') such that (1) $(d, c) \in M(v'', v)(d'', c'')$, (2) (d'', c'') is in $PF(I_k^\infty)(\pi'')(\top, c_0)$, and (3) therefore, by the IH, that $(d'', c'') \notin MVP^{-1}(I, v', (d', c_0))(v'')$.

We shall prove the induction step by contradiction. Suppose that $(d, c) \in MVP^{-1}(I, v', (d', c_0))(v'')$. By Lemma 3, there is a $(d, c_*) \in B(v', d', v)$, such that $c_* \leq c$. Consider the countermap $\lfloor c \rfloor = \lambda x. \min\ \{c(x), k+1\}$. As c is a C_k^∞ counter, and k is bigger than every element in the range of c_* (it is the maxcount), it follows that $c_* \leq \lfloor c \rfloor$. Thus, as backwards solutions are upward closed (Lemma 3), $(d, \lfloor c \rfloor) \in MVP^{-1}(I, v', (d', c_0))(v)$. By splitting cases on the possible operations on the edge (v'', v), we can show that that there exists a c''_* such that: (i) $(d, \lfloor c \rfloor) \in M(v'', v)(d'', c''_*)$ and (ii) $c''_* \leq c''$. In other words, (d'', c''_*) is in $MVP^{-1}(I, v', (d', c_0))(v'')$. By the upward closure of the backwards solution, $(d'', c'') \in MVP^{-1}(I, v', (d', c_0))(v'')$, thereby contradicting (3) above. Hence, $(d, c) \notin MVP^{-1}(I, v', (d', c_0))(v'')$, completing the proof of **IH** and therefore, the lemma. ∎

We can now prove Completeness Lemma 1.

PROOF. *(of Lemma 1)*. Suppose that there exists a c such that $(d, c) \in MVP(I)(v)$. Then there exists some path $\pi \in IVP(v^s_{main} v,)$ such that $(d, c) \in PF(I)(\pi)(v)$. Picking the length of π as $k_{d,v}$ suffices, as for all k greater than this $k_{d,v}$ we can prove that, $(d, \cdot) \in MVP(I_k)(v)$, and from Theorem 3, $(d, \cdot) \in MVP(I_k^\infty)(v)$.

Suppose that there is no c such that $(d, c) \in MVP(I)(v)$. If we let $k_{d,v}$ be the maxcount of A, then Lemma 4 shows that there is no $(d, \cdot) \in MVP(I_k^\infty)$, and from Theorem 3, there can be no $(d, \cdot) \in MVP(I_k)$. ∎

This concludes the proof of correctness. Notice that while the correctness proof relies on several technical notions, these can all be hidden from an implementer, who only needs to call an interprocedural dataflow analysis algorithm with appropriate lattices.

6. Conclusion

We believe that our AIFDS framework and algorithm provides an easy to implement procedure for performing precise static analysis of asynchronous programs. While theoretically expensive, our initial experiments indicate that the algorithm scales well in practice. Thus, we believe the algorithms presented in this paper open the way for soundly transferring the dataflow analysis based optimization and checking techniques that have been devised for synchronous programs to the domain of asynchronous programs, thereby improving their performance and reliability.

References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuan Tsay. General decidability theorems for infinite-state systems. In *LICS 96*, pages 313–321. IEEE Press, 1996.

[2] T. Agerwala and J. Misra. Assertion graphs for verifying and synthesizing programs. Technical Report 83, University of Texas, Austin, 1978.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

[5] R. Cunningham. eel: Tools for debugging, visualization, and verification of event-driven software, 2005. Master's Thesis, UC Los Angeles.

[6] R. Cunningham and E. Kohler. Making events less slippery with Eel. In *HotOS-X*, 2005.

[7] G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded Java programs. In *TACAS 02*, LNCS 2280, pages 173–187. Springer, 2002.

[8] L.E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with r distinct prime factors. *Amer. Journal Math.*, 35:413–422, 1913.

[9] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] M. Emmi and R. Majumdar. Decision problems for the verification of real-time software. In *HSCC 06*, LNCS 3927, pages 200–211. Springer, 2006.

[11] J. Esparza. Decidability and complexity of Petri net problems –an introduction. In *Lectures on Petri Nets I: Basic Models*, LNCS 1491, pages 374–428. 1998.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003: Programming Languages Design and Implementation*, pages 1–11. ACM, 2003.

[13] G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, enlarge, and check: New algorithms for the coverability problem of WSTS. In *FSTTCS 04*, LNCS 3328, pages 287–298. Springer, 2004.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 03: Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.

[16] T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI 04: Programming Languages Design and Implementation*. ACM, 2004.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[18] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computing Systems*, 18(3):263–297, 2000.

[20] Libasync. http://pdos.csail.mit.edu/6.824-2004/async/.

[21] Libevent. http://www.monkey.org/%7Eprovos/libevent/.

[22] B.D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs i. *Acta Informatica*, 21:125–169, 1984.

[23] The mace project. http://mace.ucsd.edu/.

[24] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL '06: Principles of programming languages*, pages 346–358. ACM, 2006.

[25] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Tech. Conf.*, pages 199–212. Usenix, 1999.

[26] Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[27] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2):416–430, 2000.

[28] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[29] M.F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP 05*, pages 92–104, New York, NY, USA, 2005. ACM.

[30] K. Sen and M. Vishwanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV 06*, LNCS 4314, pages 300–314. Springer, 2006.

[31] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[32] N. Zeldovich, A. Yip, F. Dabek, R.T. Morris, D. Mazières, and M.F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Technical Conference*, pages 239–252, 2003.