TUHH
Technische Universität Hamburg-Harburg

Bachelor Thesis

Boolean Analysis for Path-sensitive Interprocedural Analyses of Asynchronous Programs

Jonathan Hüser

September 24, 2012

Advisor:
Volker Menrad

Supervisor:
Prof. Dr. Sibylle Schupp

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstraße 95
21073 Hamburg

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 24. September 2012

Jonathan Hüser

Contents

1 Introduction

Asynchronous programs contain procedure calls that are not executed from the call site, but posted as *tasks* in a queue for later (asynchronous) execution. Tasks are always run to completion after which the next task is "dispatched" from the queue in a non-deterministic order, creating a *schedule* of executed tasks [4].

With this execution style, tasks in asynchronous programs are similar to atomic operations in concurrent programs. Writing correct concurrent programs is challenging, and the same can be said about asynchronous programs.

Global variables may be altered by operations between the call and the dispatch of a task and it is non-deterministic which operations appear between these points. To gain control, it is important to consider how to enforce a certain execution order (scheduling). A possible option is to utilize boolean flags as guards as seen in the following asynchronous program:

```
task main:              task prepare:          task send:
  pending := false          pending := true      if pending then
  post prepare                                     R: ...
  post send
```

The program starts at the entry of the `main` task by assigning the global boolean variable `pending` to `false`. Then the tasks `prepare` and `send` are posted, either of which could be executed next because of the non-determinism of the dispatcher.

Considering all possible execution traces, we come to the conclusion that the code at label `R` can only ever be reached if the `prepare` task was dispatched *before* `send`, due to the global flag `pending`.

The non-deterministic aspect in the execution of asynchronous programs generates a critical need for better *precision* in static analysis techniques (as asynchronous programs are typically written to provide high-performance infrastructure) [4]. The method of eliminating unwanted schedules from the list of possible execution orders with boolean guards as seen in the program above, is not yet considered by the approach to *data flow* analysis of Jhala and Majumdar [4], despite the fact that it provides potential to improve them.

The goal of this thesis is to improve an existing interprocedural analysis for asynchronous programs by considering global boolean flag values to make it path-sensitive and eliminate invalid data flow. We present an extended approach to the interprocedural analysis of asynchronous programs by taking the following steps:

- We present a unified approach to *path-sensitive* interprocedural analysis of asynchronous programs combining the previous works of Ball and Rajamani with that of Jhala and Majumdar (Section 4). Path-sensitive data flow analyses provide the general means to rule out infeasible paths by tracking correlations between data flow facts, which is exactly what we need to deal with boolean guards.

- A boolean value analysis can be used to correlate flag values to program paths, making it possible to ignore paths from the analysis, that are eliminated by the use of guards. In Section 4 we present a straight-forward boolean value analysis.

- Using the boolean analysis for path-sensitivity in interprocedural analyses adds *precision* to the *meet-over-all-valid-paths* solution of such problems. We show this with the example of a classical *reaching definition* analysis running on our path-sensitive dataflow engine by evaluating it for some example programs and determining the accuracy of the results (Section 5).

- In Section 5 we also show that in some cases our boolean analysis causes the combined analysis to converge to a fixpoint faster, making the underlying algorithm that is used to compute the solution to be more *efficient*. This is an interesting result, because generally there is a trade-off between precision and efficiency in dataflow analyses.

2 Background: Data Flow Analysis

In this section we will introduce all the relevant ideas about data flow analysis that are necessary to understand both the problems that occur when attempting to analyze asynchronous programs, and the solutions to those problems that were developed in the context of the thesis work.

2.1 Bitvector Analysis

We will start by introducing intraprocedural data flow analysis by taking the example of a *reaching definition* analysis. The reaching definition analysis belongs to a group of analyses called *bitvector* analyses, that work well as a basis for the techniques developed throughout the rest of the thesis.

The general goal of data flow analyses is to figure out how data flows through a programs control structures [5]. This can be done by looking at how statements and expressions manipulate and transport the values of variables. There are very different data flow analyses and each of them has the goal of answering a different question. In order to understand how data flow analyses actually work, we first need to understand what it is they analyze and how data flow information is represented.

Flow-sensitive analyses, the kind of data flow analyses we are interested in, work with a *control flow graph* representation of a program.

Control Flow Graph. A procedure is represented by a directed graph $G_p = (V_p, E_p)$ called a *control flow graph* (CFG). The CFG has a unique `Start` node s_p and a unique `End` node e_p. The other nodes represent the statements and conditions (basic blocks). The edges describe the flow of the represented program.

The *reaching definition* analysis is a data flow analysis that determines for all program points, which definitions (assignments) of a variable can *possibly* reach that point without being overwritten [5]. The emphasis is on "possibly", because it is not always possible to get an accurate result with data flow analyses in general. The result of the reaching definition analysis approximates by providing more definitions than are actually valid at a program point, rather then leaving one out. A reaching definition analysis can be useful for finding relations between definitions of variables and expressions in which they are used (*definition-use* pairs) to build data flow dependency models while debugging and for optimization purposes by eliminating definitions that are never used [5].

Consider Figure 2.1, in which you can see the *control flow graph* of a simple procedure with only one variable `a` and the result of a reaching definition analysis annotated on the edges of the graph.

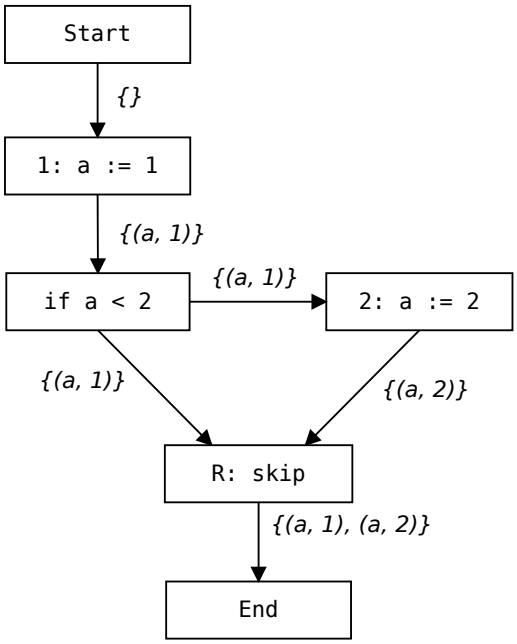

Figure 2.1: Reaching definition analysis for a single variable

There are two assignments in Figure 2.1 that involve variable `a` labeled 1 and 2. The reaching definition analysis starts at the `Start` *block* of the CFG. Because `a` is undefined at that point, the set of reaching definitions is empty, which is annotated as $\{\}$ on the edge from the `Start` block to the first assignment. We use tuples of variable name and assignment label, to refer to a certain definition of a variable. After the first assignment, the set of reaching definitions is $\{(a, 1)\}$, meaning the definition in the first assignment reaches the following program points until there is another assignment. At that point, the control flow splits into two branches, one of which would never be executed in an actual run of the program, because the condition `a < 10` always holds. A simple *flow-sensitive* reaching definition analysis still considers both paths because it does not keep track of the values of variables. Just before the `End` block, the set of reaching definitions is $\{(a, 1), (a, 2)\}$ because in one branch another assignment is made and after the branches are joined either of the two assignments could be the reaching definition.

Given that all nodes in the CFG can be referred to by their unique labels n and $pred(n)$ is defined as the set of labels of all predecessors of node n, the relation between the sets of possibly reaching definitions between different points in the program can generally be described by the following data flow equations [5]:

$$
\begin{aligned}
IN_n &= \begin{cases} \emptyset & n \text{ is a } \texttt{Start} \text{ block} \\ \bigcup_{p \in pred(n)} OUT_p & \text{otherwise} \end{cases} \\
OUT_n &= (IN_n \setminus KILL_n) \cup GEN_n
\end{aligned}
$$

The IN_n set is responsible for combining the previously valid data flow facts and is defined as the union of all OUT_p sets of the nodes predecessors. This combination of data flow fact is also called *meet*. OUT_n is the actual *data flow transfer function* or just data flow function. It is formulated using the three sets IN_n, $KILL_n$ and GEN_n, where

- IN_n is the parameter of the data flow function, the set of data flow facts that are valid *before* block n,
- $KILL_n$ is the set of data flow facts that are *killed* by block n, meaning they are not valid after that block, and
- GEN_n is the set of data flow facts that are *generated* by block n.

In case of the reaching definition analysis only assignment blocks actively manipulate the data flow. The $KILL_n$ and GEN_n sets are defined as follows:

$$
\begin{aligned}
KILL_n &= \begin{cases} \{(a, m) \in vars \times labels \mid a \text{ is defined in } m \text{ and } n\} & n \text{ is assignment} \\ \emptyset & \text{otherwise} \end{cases} \\
GEN_n &= \begin{cases} \{(a, n) \in vars \times labels \mid a \text{ is defined in } n\} & n \text{ is assignment} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

An assignment block kills all previous definitions of the assigned variable and generates the tuple of that variable and the assignment label.

The reaching definition analysis is a classical example of a *bitvector* analysis. It is called a bitvector analysis because its data flow facts can be encoded using bitvectors. Bitvectors allow for fast set union and intersection computation as bitwise operations [5]. Some bitvector analyses use intersection instead of union to combine (meet) data flow facts, run backwards, or both. All of them necessarily have finite data flow information, something that will turn out to be a useful property later on.

We learned that data flow analyses can be used to gather information about programs for optimization purposes or debugging. We also saw how simple analysis problems such as the bitvector analyses can be described with data flow equations by defining GEN and $KILL$ sets.

2.2 Data flow Frameworks

We will now introduce some of the concepts needed to show that there can be a solution to an analysis problem that can be computed algorithmically. Our goal is to arrive at a simple framework that can be used to formulate data flow problems for which we know that an algorithm to solve them correctly also terminates.

If a set of data flow equations can be used to describe how information propagates through the CFG of a program, then solving this set of equations solves the data flow analysis problem [5].

Data flow frameworks can be used to categorize analysis problems. Different kinds of analyses require different levels of complexity in the algorithms that solve them.

The relation between data flow facts and the data flow equations in the reaching definition analysis in Figure 2.1 are relatively simple. Consider the data flow facts in Figure 2.2.

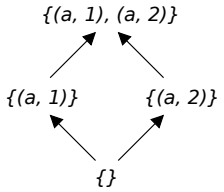

Figure 2.2: Complete lattice for reaching definition example in Figure 2.1

The way in which data flow facts in Figure 2.2 are arranged, represents the order in which they appear in the analysis in Figure 2.1. At the beginning of the program there is no data flow information present, so the analysis starts at the bottom with an empty set. With each assignment there is either a *new* definition (as is the case in block 1), or another definition is *overridden* (block 2). In the first case it causes the data flow information to be extended which correlates to going up in Figure 2.2. In the second case it causes some data flow fact to be replaced which correlates to staying at the same level in Figure 2.2.

When different sets of data flow facts meet in the IN set of a block, this also correlates to going up in the lattice, if they are different or staying at the same level, if they are equal.

It will later become relevant that at no point in the analysis data flow information is reduced.

For now it is sufficient to realize that the subset operator $\subseteq$ induces a *partial order* over the data flow facts in the reaching definition analysis of Figure 2.1.

Partially Ordered Set. A *partially ordered set* $(P, \sqsubseteq)$ is a set P with a relation $\sqsubseteq$ over $P \times P$ that is a partial order, meaning it is:

1. *Reflexive.* For all $d \in D : d \sqsubseteq d$
2. *Transitive.* For all $d_1, d_2, d_3 \in D : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ implies $d_1 \sqsubseteq d_3$

3. *Anti-symmetric.* For all $d_1, d_2 \in D : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ implies $d_1 = d_2$

All of the conditions of a partial order hold for the subset $\subseteq$ relation, but there can also be partially ordered sets with other relations. The subset operator $\subseteq$ induces a partial order not only on the data flow facts in Figure 2.2 but on any power set 2^D. Within a partially ordered set $(2^D, \subseteq)$ there is also the notion of the *least upper bound* and *greatest lower bound*.

Least upper bound / Greatest lower bound. For a partially ordered set $(P, \sqsubseteq)$ an *upper bound* (*lower bound*) of a subset $S \subseteq P$ is an element $s_1 \in S$ such that for all $s_2 \in S$: $s_2 \sqsubseteq s_1$ ($s_1 \sqsubseteq s_2$). The *least upper bound* (or *join*) of a subset $S \subseteq P$ is $\bigsqcup S$ an upper bound $s_1 \in S$ such that for all other upper bounds $s_2 \in S$: $s_1 \sqsubseteq s_2$. The *greatest lower bound* (or *meet*) of a subset $S \subseteq P$ is $\bigsqcap S$ a lower bound $s_1 \in S$ such that for all other lower bounds $s_2 \in S$: $s_2 \sqsubseteq s_1$.

Within the partially ordered set $(2^D, \subseteq)$ the least upper bound of a subset $S \subseteq 2^D$ is actually $\bigcup S$, the union of all sets in S and the *greatest lower bound* of a subset $S \subseteq 2^D$ is $\bigcap S$, e.g. in the case of the example in Figure 2.2, the least upper bound of $\{\{(a,1)\}, \{(a,2)\}\}$ is $\bigcup\{\{(a,1)\}, \{(a,2)\}\} = \{(a,1), (a,2)\}$.

The reason we need the least upper bound and the greatest lower bound is to show that the power set of data flow facts is not only a partially ordered set, but also a *complete lattice*.

Complete Lattice. A partially ordered set $(P, \sqsubseteq)$ is a *complete lattice* iff for all subsets $S \subseteq P$, both the least upper bound $\bigsqcup S$ and the greatest lower bound $\bigsqcap S$ are in P.

The example in Figure 2.2 is a complete lattice, although a trivial case. More generally, all power sets $(2^D, \subseteq)$ of data flow facts with the subset relation are complete lattices.

We already hinted that the data flow equations cause the data flow information in the sets to "go up" in the lattice. Actually the bottom element $\bot$ in its lattice is the most *accurate* statement an analysis can make at any point. The top element $\top$ is the most inaccurate or approximated.

What accurate means in this context is not necessarily that something is correct, but that there is no ambiguity. Just giving the top element $\top$ as answer for all points in the program, always gives a *correct* solution to the analysis problem which, however, is inaccurate or ambiguous.

The above is relevant because complete lattices have a property that is useful for data flow analyses: If you know "the ways that only go up eventually lead to a fixpoint" and "you only go up" you can prove that you eventually arrive at a solution. The first condition is called the *ascending chain condition*.

Ascending Chain Condition. A partially ordered set $(P, \sqsubseteq)$ satisfies the *ascending chain condition* iff every *ascending chain* $p_1 \sqsubseteq p_2 \sqsubseteq \ldots$ ($p_i \in P$) stabilizes, meaning there exists an n such that for all $m > n$ $p_n = p_m$.

The ascending chain condition trivially holds for finite complete lattices because there is no element in the lattice that is bigger than the least upper bound of all elements.

The second condition that needs to be met for an analysis to terminate is that data flow facts may only "go up" or "stay at the same level" in the lattice throughout the analysis. A formal way of expressing this, is by saying that data flow functions have to be *monotone*.

Monotone Function. A data flow function $f : 2^D \to 2^D$ is called *monotone* iff for all $d_1, d_2 \in 2^D$: $d_1 \sqsubseteq d_2$ implies $f(d_1) \sqsubseteq f(d_2)$.

If f is monotone then $f(d_1 \sqcap d_2) \sqsubseteq f(d_1) \sqcap f(d_2)$.

What it means for a data flow function to be monotone is that applying that function to a set of data flow facts cannot make them more specific. An even stronger statement is to say that a data flow function is distributive.

Distributive Function. A data flow function $f : 2^D \to 2^D$ is called *distributive* iff for all $d_1, d_2 \in 2^D$: $f(d_1 \sqcap d_2) = f(d_1) \sqcap f(f_2)$.

If f is distributive, it is also monotone.

If a data flow function is distributive, it does not matter whether you put two sets of data flow facts through the manipulating effects of a block (such as an assignment) and let them meet after the block, or let them meet first and then apply the data flow function of the block.

All bitvector analyses are examples of distributive analysis problems, because of the way their IN sets are defined using the meet of previous data flow information [5]. Distributive analysis problems have the advantage that it is actually possible to calculate their *precise* solution with low complexity [6].

After discussing complete lattice, ascending chain condition and distributive functions, we can use these concepts to define a framework for analysis problem for which it is always possible to compute a solution, by demanding that an analysis

- has to have a finite set of data flow facts,
- needs to be distributive and
- has data flow facts that form a power set with the meet operator either being set union or intersection.

FDS Instance. The instance of a *finite distributive subset* framework is a tuple $(G, D, F, M, \sqcap)$, where

1. $G = (V, E)$ is the CFG of a program,
2. D is a finite set of data flow facts,
3. $F \subseteq 2^D \to 2^D$ is a set of *distributive* data flow functions,
4. $M : E \to F$ maps each edge of G to a distributive data flow function,
5. The meet operator $\sqcap$ is either union or intersection.

Looking at the reaching definition analysis example in Section 2.1 as an FDS instance we get:

- G is the CFG visible in the Figure 2.1,
- $D = \{(a, 1), (a, 2)\}$,
- F is set of functions $f(X) = (X \setminus KILL) \cup GEN$ with $KILL, GEN \in 2^D$,
- M maps outgoing edge of block n to OUT_n,
- The meet operator $\sqcap$ is union $\cup$.

There are more general formulations of distributive frameworks than FDS. There are also even more general *monotone* frameworks that can be used to describe non-distributive data flow problems such as *constant propagation* [5]. The FDS framework serves well enough as a general basis to build upon to develop the more complex analysis problems presented throughout the rest of the thesis.

2.3 Meet-over-all-Paths Solution

The previous sections give an intuition on what the result of a data flow analysis is. We will now actually define the *precise* solution to an FDS instance and show how it can be computed with an iterative algorithm.

The question of how data is manipulated from the beginning of program execution up to a point in the program, answered for all points in the program, is the solution to a data flow analysis problem. From the perspective of a single program point, finding the solution means finding all the *paths* in the control flow that could possibly lead to that point and figuring out how the blocks on that path manipulate the data. The results are then combined using the meet operator of the analysis, producing the *meet-over-all-paths* (MOP) solution. In the case of node `R` in Figure 2.1, the MOP solution is

$$MOP_R = \{(a, 1), (a, 2)\} = \{(a, 1)\} \cup \{(a, 2)\} = ...$$

There are only two paths that lead to `R` and manipulate the data flow facts so that they are $\{(a, 1)\}$ for one path and $\{(a, 2)\}$ for the other. The meet operator in the reaching definition analysis is the set union $\cup$. Note that the MOP solution, there are no IN sets like we saw in the data flow equations. The argument to the data flow function is not $IN_n = \bigcup_{p \in pred(n)} OUT_p$ in this context, but $X_n = f_p(X_p)$ for the single predecessor p along the path.

To formally define the MOP solution, we first need to define what a path actually is:

Path. A *path* $\pi = (e_1, ..., e_n)$ from node v to v' is a sequence of edges, where v is the source of e_1, v' is the target of e_n and for each $1 \leq k \leq n - 1$, the target of e_k is the source of e_{k+1}. $\pi(k)$ is the kth edge of the path π. We denote the set of all paths from the `Start` block to n by $P(n)$.

To define how the data flow behaves through a single path, we define the *path function*. It is just the data flow function applied to every single node on the path based on the result of the previous node.

Path Functions. The *path function* that corresponds to a path π, is the function $PF_\pi = f_n \circ ... \circ f_2 \circ f_1$, where for all k, $1 \leq k \leq n$, $f_k = M(e_k)$, the data flow function on the according edge. The path function for the empty path is the identity function.

To get the general *meet-over-all-paths* solution to an analysis problem, we not only need a single node and all of its paths. The path function of every path from program start to every node in the CFG needs to be considered.

MOP Solution. The *meet-over-all-paths* solution to an FDS instance $A = (G, D, F, M, \sqcap)$ is a map $MVP_n : V \to 2^D$:

$$MOP_n = \bigsqcap_{\pi \in P(n)} PF_\pi(\bot) \text{ for each } n \in V$$

For intraprocedural analyses the MOP solution is the *precise* solution [6].

A straight forward way of implementing an analysis that computes the MOP solution would be to go through all paths for all nodes and computing the path functions separately. This approach has high complexity and is not efficient enough in most cases.

A better strategy is to go through the control flow graph and iteratively calculating the IN and OUT sets of blocks on the basis of what is already known until the content of all sets stop changing and a fixpoint is reached. An efficient way of implementing such a strategy is by using a *worklist* algorithm.

```
WL := nil;
forall the (l, l') ∈ edges(CFG) do
 | WL := cons((l, l'), WL);
end
forall the l ∈ blocks(CFG) do
 | IN_l := ⊥;
end
while WL ≠ nil do
 | (l, l') := head(WL);
 | WL := tail(WL);
 | if IN_l' ≠ OUT_l then
 | | IN_l' := IN_l' ⊎ OUT_l;
 | | forall the l'' with (l', l'') ∈ edges(CFG) do
 | | | WL := cons((l', l''), WL);
 | | end
 | end
end
```

Algorithm 1: Worklist Algorithm

The worklist algorithm has a list of edges (the worklist) on which it operates until it is empty. In the beginning, all edges of the CFG are put into the worklist. Each iteration the algorithm computes the data flow through the node that is the target of the edge at the head of the worklist and if the result in the OUT set differs from what it previously was, all edges pointing to successor nodes are added at the back of the worklist. The worklist will only ever be empty, if a fixpoint is reached.

The result computed by this algorithm is not the MOP solution, because data flow facts meet at the IN set of a node in every iteration and not just before the node. The solution computed by an iterative approach is called the *maximum-fixed-point* solution for historical reasons [5], and in the context of how data flow analyses were introduced within this thesis should rather be called smallest-fixed point, because it is the most imprecise fixpoint solution.

MFP Solution. The *maximum-fixed-point* (MFP) solution is the smallest fixpoint of the data flow equations in a lattice where the most accurate solution is at the bottom and solutions are approximated by moving upwards. For the instance of a monotone framework as its input, the worklist algorithm always terminates and computes the MFP solution to the instance.

The MFP solution safely approximates the MOP solution for monotone frameworks ($MFP \sqsubseteq MOP$), because $f(d_1 \sqcap d_2) \sqsubseteq f(d_1) \sqcap f(d_2)$. For distributive frameworks the MOP solution is equal to the MFP solution ($MFP = MOP$), because $f(d_1 \sqcap d_2) = f(d_1) \sqcap f(d_2)$ [5]. Therefore, the worklist algorithm can be used to compute the precise solution of distributive frameworks such as FDS frameworks.

We now have all the tools to explore the problems of asynchronous program analysis and also saw a practical example of what data flow analysis could generally be used for.

3 Analysis of Asynchronous Programs

In this section, we will describe the problems that arise when analyzing asynchronous programs, specifically those that use boolean flags to manipulate execution order.

3.1 Asynchronous Programs

In regular procedural programs, whenever a (synchronous) call is made, the execution jumps into the called procedure directly from the call site. In contrast to that, in asynchronous programs, procedures can be called asynchronously causing them to not be executed from the call site, but scheduled in a task queue to be executed later. Asynchronously called procedures will therefore be called *tasks*. Tasks are always run to completion, after which a non-deterministic scheduling mechanism choses the next task from the queue. This execution model results in the fact that in asynchronous programs, global variable values can change between the call site and the jump into a procedure, a relevant difference to regular procedural programs when concerned with data flow analysis.

The fact that asynchronous programs are an extension of regular interprocedural programs will be ignored in this thesis. We will not consider synchronous calls at all, because this adds unnecessary confusion to an already complex problem.

The concept of asynchronous calls permits for some of the advantages of parallel programming, without having to worry about synchronization, e.g. to support non-blocking I/O-intensive operations. Asynchronous programs also form the basis of event-driven programming, the paradigm in which external events influence the execution of programs. In asynchronous programs, this behavior can be modeled through callback tasks, which are scheduled to respond to external events.

Consider the example program in Figure 3.1. There are three tasks: `main`, `prepare` and `send`. It is safe to assume that an actual execution of the program would start by dispatching the `main` task. Within the `main` task, the two other tasks are posted to be executed later on, after the end of the `main` task. After the `End` node, there is no deterministic way of knowing how execution will continue. Either `prepare` is dispatched first and run to completion after which `send` would be the only task left to be executed or the other way around. The dotted arrows that connect all `End` nodes of the individual tasks with all `Start` nodes describe the control flow between tasks.

The intraprocedural CFG of each task can be collapsed and what remains is a complete directed graph. This observation is valid for any asynchronous program and directly results from the fact that the most general assumption we can make about the dispatcher is, that tasks are executed in an arbitrary order.

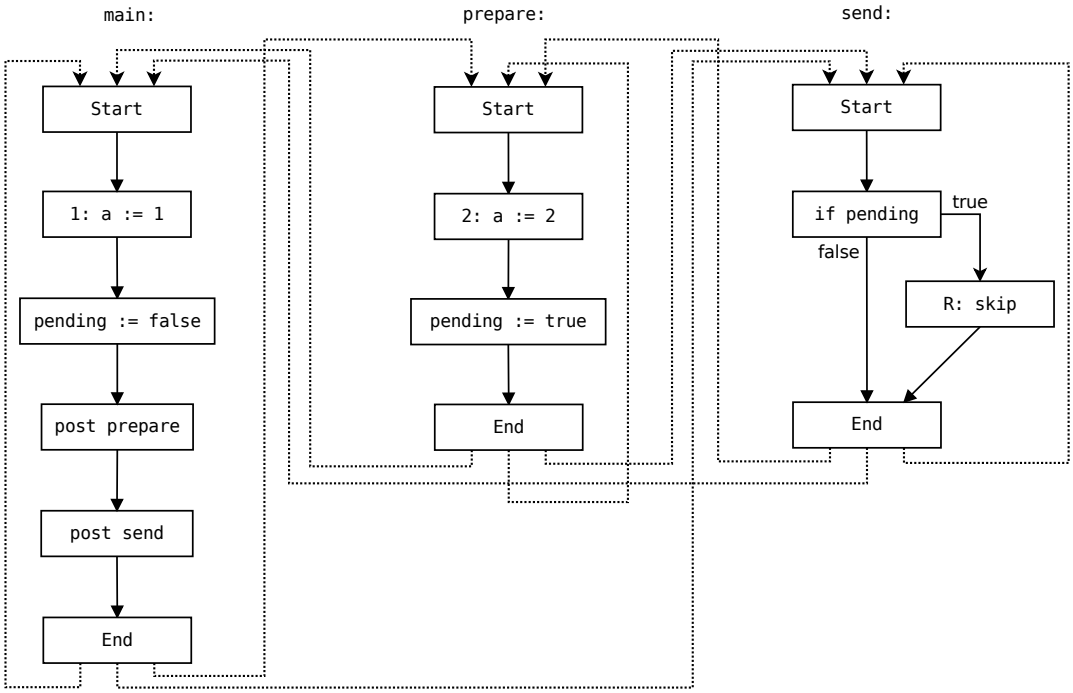

Figure 3.1: Asynchronous Program

Asynchronous Control Flow Graph. An asynchronous program is represented by a directed graph $\widehat{G} = (\widehat{V}, \widehat{E})$ called *asynchronous control flow graph* (ACFG). $\widehat{G}$ consists of a collection of CFGs $G_1, G_2, ...$, one for each *task*. In addition to the intraprocedural edges, $\widehat{G}$ contains interprocedural *asynchronous dispatch* edges from every `End` node to every `Start` node in the graph.

The problem of the ACFG in the context of data flow analysis is that with the straight-forward approach of treating the ACFG like a CFG and the analysis as an intraprocedural one, the results will be very inaccurate [6]. This is true for interprocedural analysis in general, but even more so for asynchronous programs, because they are so highly connected interprocedurally [4]. In order to obtain a precise analysis result, an addition to the analysis is necessary that adds a sense of context to data flow information based on the asynchronous calls that were made.

The general difficulty with asynchronous programs is, that there is no deterministic connection between the asynchronous calls and the interprocedural execution order. Ignoring this difficulty, leads to unsound results [4]. Not ignoring it leads to the necessity of an analysis that is much more complex than what we have seen so far.

3.2 AFDS Framework

We will now show how the analysis of asynchronous programs can be made *context-sensitive* to allow only valid execution orders. Context-sensitivity makes the important step towards accurate data flow analysis for asynchronous programs by giving data flow information a sense of interprocedural context. Specifically, the goal is to keep track of the calling context in the paths of the MOP solution, to be able to ignore interprocedurally *invalid* paths.

For regular interprocedural programs, context-sensitivity can be achieved by using a Dyck language (a language with balanced parentheses labeled according to calling context) to check whether all called procedures are returned from to the correct procedure. Normally, this feature needs to be included in the interprocedural analysis of asynchronous programs, because they support synchronous calls as well [4]. We will ignore synchronous calls entirely, for reasons of simplicity.

Context for asynchronous calls is in a way simpler than that of synchronous calls, because there are no interprocedural return edges in the control flow and therefore there is no need to check for a balance between call and return edges. But context for asynchronous calls is in another way also more complicated, because asynchronous execution order is non-deterministic and there is no simple way of formulating valid interprocedural context with a context-free language [6]. One way to achieve context-sensitive interprocedural analysis for asynchronous programs is to use counters that track how often each task has been posted [4].

Consider the example in Figure 3.2, which shows a reaching definition analysis with counters. There are now three numbers in addition to the data flow facts of the reaching definition analysis on each edge. These numbers are counter that indicate how often each of the tasks is scheduled to be executed respectively (the first number for `main`, the second for `prepare` the third for `send`).

The analysis starts with the counter for the `main` task being at 1 and all other counters at 0, i.e. `main` is the task that is dispatched and executed first in any case. Within the `main` task, the two asynchronous calls cause the respective counters of the called task to be increased.

After the `End` block of the `main` task, data flow information is propagated to the `Start` blocks of all three tasks according to the interprocedural edges. In the case of `main`, data flow information is ignored, because the respective counter is zero. In the other two tasks data flow information is propagated through the `Start` block, and the respective counter is decreased to indicate that the task has been dispatched. The rest of the analysis continues as expected while some of it is left out in Figure 3.2 in order to keep it clear.

It is relevant to notice, that two different data flow facts reach `R`, when actually only one of them would in a real execution of the program. We define the framework for the category of problems that can be solved with a counter analysis as the *asynchronous finite distributive subset* framework.

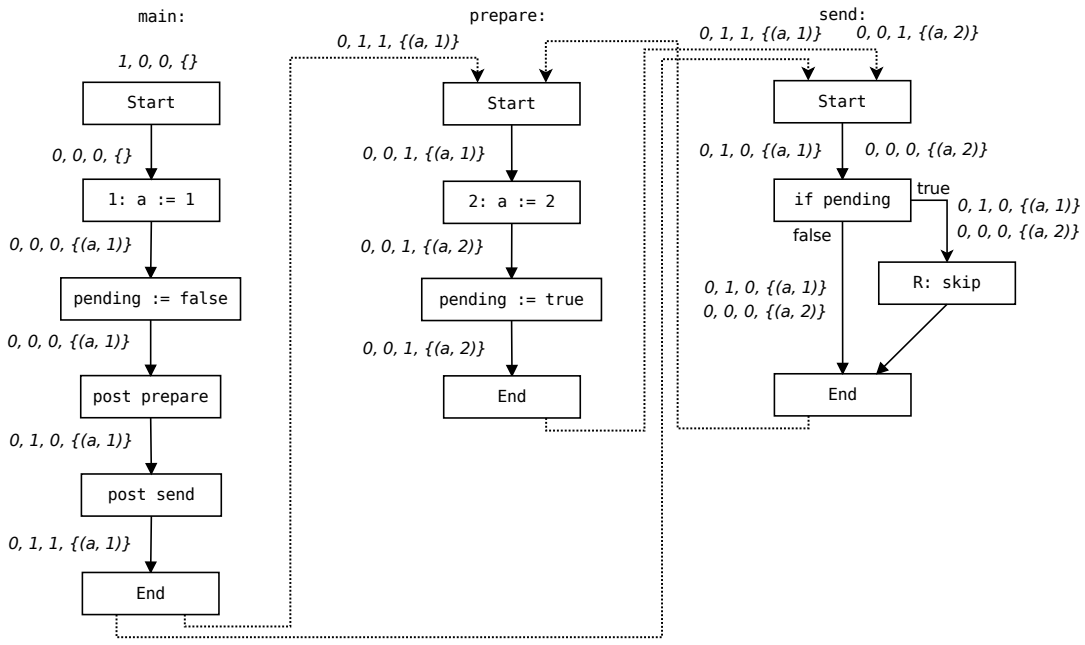

Figure 3.2: Asynchronous Program with Reaching Definition and Counter Analysis

AFDS Instance. An instance of an *asynchronous finite distributive subset* problem is a tuple $A = (\widehat{G}, D, F, M, \sqcap)$ where:

1. $\widehat{G}$ is the asynchronous control flow graph $(\widehat{V}, \widehat{E})$ of an asynchronous program,
2. D is the finite set of global data flow facts,
3. $F \subset 2^D \to 2^D$ is the set of distributive data flow functions,
4. $M : \widehat{E} \to F$ maps each edge in $\widehat{G}$ to a data flow function,
5. $\sqcap$ is the meet operator, which is either set union or intersection.

In contrast to the analysis problems it can be combined with, the counter analysis itself is not an FDS instance, because it is not finite. The fact that the counter analysis combined with an AFDS instance terminates is also not as straight forward as it is for regular FDS instances.

There is a proof by Jhala and Majumdar, that their AFDA algorithm solves AIFDS instances which are the same as AFDS instances except they include synchronous interprocedural analysis [4]. The AFDA algorithm executes two variations of the counter analysis multiple times for increasing values of an integer k, starting with $k = 1$. In one analysis the counters count to k and then stop to increase. In the other analysis the counters count to k, then jump to ∞ and never decrease

afterwards. There will be a value of k, for which both of these analyses produce the same solution. For this fixpoint, the analysis is shown to be correct. The AFDA algorithm that computes the solution is EXPSPACE-hard [4].

Further details of how exactly the counter analysis works are not relevant to understanding the problem solved in the context of this thesis. All we need to know is, that there is a way of computing the precise solution.

To define the precise solution to an interprocedural analysis of asynchronous programs, we first need a formal notion of what an interprocedurally valid path is. We will then define the solution as a meet-over-all-*valid*-paths solution. For a path to be interprocedurally valid it has to have a *schedule*.

Schedule. $\phi : \mathbb{N} \to \mathbb{N}$ is a schedule for a path π iff ϕ is one-to-one and for each $1 \leq k \leq n$, if $\pi(k)$ is an asynchronous dispatch edge to task t then:

- $0 \leq \phi(k) \leq k$ (meaning the node in which the dispatched task was posted must have appeared on the path) and
- the source of $\pi(\phi(k))$ is an asynchronous call (post) to task t.

We denote the set of all paths from the `Start` block in the `main` task to n for which there exists a schedule by $ScheduledP(n)$ and call them *valid* paths in the context of asynchronous programs.

In the example in Figure 3.2 a schedule for the path from the `Start` block in the `main` task, to the `End` block of the `prepare` task would have $\phi(e) = n$ with e being the edge from the `End` block of `main` to the `Start` block of `prepare` and n being the index of the edge that has the block with `post prepare` as a source within the path.

Given the above definition of a schedule, we can now define the solution to an AFDS instance as.

MVP Solution. The meet-over-all-valid-paths (MVP) solution to an AFDS instance $A = (\widehat{G}, D, F, M, \sqcap)$ is a map $MVP_n : \widehat{v} \to 2^D$:

$$MVP_n = \prod_{\pi \in ScheduledP(n)} PF_\pi(\bot) \text{ for each } n \in \widehat{V}$$

For asynchronous programs, the MVP solution is the *precise* solution [4].

All of what we have arrived at so far does not change anything about the fact, that the data flow facts in Figure 3.2 at label `R` are not completely accurate. If there was not just a skip statement at `R` but another `post main` or more, there could be much longer interprocedurally valid paths that are *infeasible* because they would not occur in an actual execution.

The problem of infeasible paths is a general one, but it is especially relevant for asynchronous programs. Using intraprocedural control structures is the only way of enforcing certain interprocedural execution orders when multiple tasks are scheduled. Figure 3.2 demonstrates the use of a global boolean flags to "guard" a

task, which causes it not to be executed if certain conditions have not been met (in this case if `pending` is not true).

The key to the problem we just described lies in making the analysis *path-sensitive*. Similar to how the counter analysis is combined with the reaching definition analysis to keep track of interprocedural context, we can add an additional analysis that correlates paths to intraprocedural control structures making the combined analysis a path-sensitive one. Because boolean values are easy to keep track of and boolean guards are a powerful way of manipulating the control flow, we will focus on utilizing them to obtain a path-sensitive analysis.

We learned that the analysis of asynchronous programs is more complex than that of regular interprocedural programs and that an additional effort to eliminate infeasible paths has to be made. Although there is already a good approach available, in some cases there can be many infeasible paths left even in a context-sensitive MVP solution, because it ignores the influence of enforced execution orders through control flow structures.

4 Adding Path-sensitivity

In this section, we are going to present a technique that attempts to contribute to the quality of interprocedural analyses for asynchronous programs. Specifically, we will introduce a boolean value analysis to make the analysis of asynchronous programs path-sensitive. We will also provide a more general approach to formulating path-sensitive analyses for asynchronous analysis problems based on the *conditional-subset meet-over-all-paths* solution.

4.1 Boolean Analysis

We will now describe the boolean value analysis approach for path-sensitive interprocedural analyses of asynchronous programs. We argue that this approach makes data flow analyses for asynchronous programs not only more accurate, but also more efficient in some cases.

The idea behind path-sensitivity is to capture a relation between data flow information and control structures in a program. What this actually means is: Only data flow information for which a condition may be true, is propagated along the according branch in an if-condition or a loop. The same holds for the data flow information for which a condition may be false.

In the case of a boolean analysis, we want to keep track of the value of a couple of global boolean flags that are used specifically for the purpose of ordering task executions. The goal is to eliminate infeasible execution traces depending on conditions that involve the global boolean flags. If booleans that used as guards are explicitly separate from other values in the program, such an analysis can be very efficient, because it causes only very little overhead while increasing the accuracy of the result significantly.

Figure 4.1 shows what a boolean analysis looks like for the example program in the previous section. The set of data flow facts for a boolean value analysis represent all possible value combinations (*valuations*) the boolean flags in the program could have. There is only one boolean value in the program, so the data flow facts are also just either `true` or `false`.

In fact `pending` starts out with either being `true` or `false`. After the first assignment involving `pending`, the data flow information is reduced so the value of `pending` becomes definite. The effect of the boolean value analysis on the reaching definition analysis can be seen when looking at the data flow information that is propagated from the `main` task to the `send` task. It does not go down the branch of the if-condition that leads to `R`, because it is clear that at this point in the program

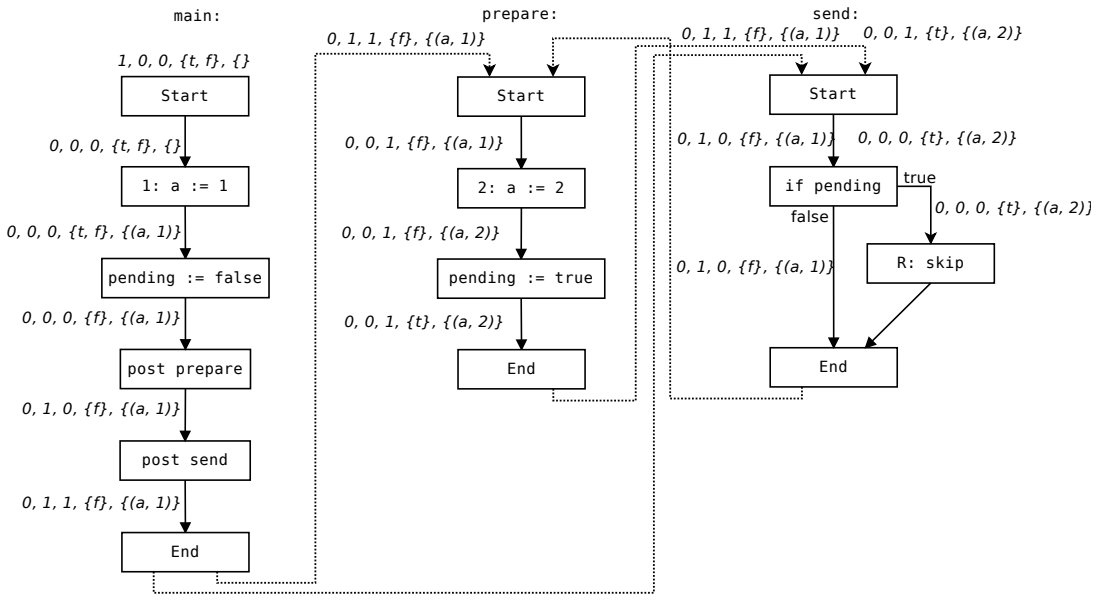

Figure 4.1: Asynchronous Program with combined Count, Boolean and Reaching Definition Analysis

`pending` is `false`. Because `pending` is only set to `true` in the `prepare` task, only data flow information that has passed through that task will ever reach `R`.

For programs with multiple global boolean flags, flags are assigned an ordering that does not change throughout the analysis. A bitvector x can then be used to assign a set of boolean values to all the flags in the program, where x_i is the value assigned to the ith flag in the ordering. The set of all possible data flow facts D are all $2^{|x|}$ permutations of bitvectors of length $|x|$, with $|x|$ being the total number of flags in the program.

Since one bitvector $x \in D$ presents one possible valuation of flags, the actual data flow information can be any set of permutations $S \subseteq 2^D$. The power set of data flow facts 2^D and the subset relation $\subseteq$ form a complete lattice $(2^D, \subseteq)$ in which $\bot =$ set of all permutations and $\top = \emptyset$.

The relevant blocks in the case of a boolean value analysis are similar to those of a reaching definition analysis: First of all, assignments have to be considered, because they manipulate the values of the flags and that is what the boolean analysis is supposed to keep track of. If-condition and loops are also relevant, because they involve conditional splitting of the control flow.

Different control structures can be used to influence the flow of a program, but they all involve a condition and multiple successors in the CFG. We will focus only on if-condition, because loops, switches etc. can all be handled in the same way.

In each assignment block, only one boolean flag is assigned a new value. The new

value is the result of the evaluation of some formula with a boolean outcome, that can consist of propositional elements but may also involve relations over numeric values or functions. Since the analysis only keeps track of boolean values, we cannot say anything about the results of relations over numeric values or the return value of functions. We will focus only on the propositional parts of the formulas, substituting the non-propositional parts by variables of which we do not know the value.

In the boolean analysis different data flow facts, bitvectors representing different valuations of flags, are manipulated by assignments one-by-one. For every bitvector, the formula in the assignment has to be evaluated and the original bitvector will be replaced with all possible bitvectors in which the newly assigned value is replaced. The following example illustrates this process.

Imagine a program with two flags `flag1` and `flag2`, and an integer variable `a`. Consider following assignment of `flag1`:

```
flag1 := flag2 or (a > 2)
```

If we know that before the assignment `flag2 = true`, then we know that after the assignment `flag1 = true`. If before the assignment `flag2 = false`, then we do not know what value `flag1` has after the assignment and we have to consider all possibilities by splitting the existing data flow information into two separate sets, one for each of the possible scenarios. This splitting of the data flow is essentially, what path-sensitivity is all about. What happens in the case of a conditional control flow split illustrates the behavior of a path-sensitive analysis in contrast to that of a regular flow-sensitive analysis even better.

A condition is also a formula with a boolean outcome, just as the formulas that are used for assignments. The difference between an assignment block and an if-condition is basically just, that the assignment splits the data flow information and propagates both parts to its one successor in the CFG, while the if-condition splits the data flow information and propagates each part to one of its two successor branches. The if-condition is simpler, because it does not actually manipulate the data flow facts like the assignment does.

All data flow facts for which the condition formula could possibly evaluate to `true`, are propagated along the "then"-branch of the if-condition and all data flow facts for which it could possibly evaluate to `false`, are propagated along the "else"-branch. Some data flow facts may be propagated along both branches, because the formula could evaluate to either `true` or `false` for them just like the case that `flag2 = false` in the assignment example above.

To define the boolean analysis using data flow equations, we first need to make some extensions to the CFG in order to be able to describe the behavior of propagating two different sets of data flow information in the case of if-conditions. The easiest way to differentiate between the "then"-branch successor and the "else"-branch successor of an if-condition block, is to label all the edges in the graph with either `true` or `false`.

Conditional ACFG. An asynchronous program is represented by a *conditional ACFG* (CACFG) in the same way as by a regular ACFG, with the following additions: Control flow edges are labeled with either *true* or *false*, depending on their position in the control flow. All edges are labeled *true* except those, that are outgoing from conditions for the case that the condition is false.

$pred(n)$ is now defined as the set of tuples (p, b), the predecessor labels of block n and the label of the edge that has p as a source and n as a target.

The changes made to the CFG do not change anything about the arguments that can be made to prove that an analysis terminates. The conditions an analysis has to satisfy are still such that all data flow functions need to be monotone. The only difference is, that the path function will now depend on what labels are on the edges of the path and the used data flow functions in the chain of function compositions is chosen accordingly.

It is possible to transform the CACFG into a regular ACFG by replacing an if-condition with two separate nodes with two different data flow functions. As long as both of these data flow functions are monotone, the analysis terminates.

Aside from the data flow functions for different kinds of blocks, an analysis also needs a meet operator to combine data flow information. There are two approaches in the context of a path-sensitive analysis that combine a boolean value analysis with some other analysis (like reaching definition). One way is somewhat more intuitive, but has the issue of causing the analysis not to be distributive. An analysis that is not distributive is troublesome when *arguing* about the accuracy of the solution computed with a worklist algorithm. The other approach is straight forward but might not be as efficient.

Consider the following three scenarios:

- Two paths meet, for which the set of possible boolean values are the same: In this case, the data flow facts of the underlying analysis are just combined with their meet operator and the two paths can be combined into one. Notice that we only need to keep track of different paths, if we actually have means to tell how they differ. If the boolean values in both paths are equal, we cannot differentiate between them and can just recombine the data flow facts of the underlying analysis.

- Two paths meet, for which both the set of possible boolean values and the underlying facts are different from each other: In this case, the two paths are simply kept separate by building a set with two tuples that contain the set of possible boolean values and the underlying facts for the two paths respectively. This way of keeping data flow facts split up in a superset, depending on some other data flow facts (in this case the boolean value facts) is the actual path-sensitive aspect of the combined analysis.

- Two paths meet, for which the set of boolean values differ, but the underlying facts are equal: In this case, there is no need to keep the two paths separate, because they manipulated the underlying data flow information in the same

way. The intuitive approach would be, to combine the two paths into one, and just widen the set of possible boolean values. This approach would cause the boolean analysis not to be distributive and will therefore not be discussed any further, although it may be viable in practice. We will treat this case as if the underlying facts were different.

Data flow facts in a combined analysis consist of tuples $\langle x, \rho \rangle$ of bitvectors x and data flow facts of the underlying analysis ρ. $\sqcap$ is the meet operator of the underlying analysis. The meet operator $\uplus$ of the combined path-sensitive analysis is defined as

$$\begin{aligned} X \uplus Y \quad = \quad & \{\langle z, \rho \sqcap \sigma \rangle \mid \langle z, \rho \rangle \in X,\ \langle z, \sigma \rangle \in Y\} \cup \\ & \{\langle x, \rho \rangle \mid \langle x, \rho \rangle \in X,\ \langle x, \sigma \rangle \notin Y\} \cup \\ & \{\langle y, \sigma \rangle \mid \langle y, \sigma \rangle \in Y,\ \langle y, \rho \rangle \notin X\}. \end{aligned}$$

To define the combined boolean analysis in terms of data flow equations, we call C_i the if-condition block with label i, where BF_{C_i} is the formula that describes the condition. BF_{C_i} can be evaluated for a given set of flag values x: $BF_{C_i}(x)$.

A_i is the assignment block with label i, where $x_{a_i} := BF_{A_i}$. a_i is therefore the index of the flag, that is assigned in A_i, in the bitvectors that describe the flag valuation of the program.

Both, the boolean value analysis and the underlying analysis have data flow facts that form a complete lattice. $\bot_b$ = set of all possible valuations and $\top_b = \emptyset$ are bottom and top element of the lattice for the boolean analysis as explained earlier. $\bot_\upsilon$ and $\top_\upsilon$ are the bottom and top element of the lattice for the underlying analysis and f_n is its flow function at block n.

The combined boolean analysis is finally defined by the following data flow equations:

$$IN_n \quad = \quad \begin{cases} \langle \bot_b,\ \bot_\upsilon \rangle & \text{block } n \text{ is } \texttt{Start} \text{ of the } \texttt{main} \text{ task} \\ \biguplus_{(p,b) \in pred(n)} OUT_p^b & \text{otherwise} \end{cases}$$

$$OUT_n^b \quad = \quad \begin{cases} \{\langle x, f_n(\sigma) \rangle \mid \langle x, \sigma \rangle \in X, BF_{C_i}(x) = b\} & n \text{ is } C_i \\ \{\langle (x_0, ..., x_{a_i - 1}, BF_{A_i}(x), x_{a_i + 1}, ..., x_{|x|}), f_n(\sigma) \rangle \mid \langle x, \sigma \rangle \in X\} & n \text{ is } A_i,\ b \text{ is } \mathit{true} \\ \{\langle x, f_n(\sigma) \rangle \mid \langle x, \sigma \rangle \in X\} & b \text{ is } \mathit{true} \\ \emptyset & \text{otherwise} \end{cases}$$

We argue that a combined boolean analysis can be formulated as an AFDS instance, as long as the underlying analysis is the instance of a FDS framework. Any two data flow analyses with lattices can be combined by constructing a complete lattice out of the product set of the power sets of data flow sets of the individual analyses.

We just need to show that the boolean value analysis is a monotone framework:

1. The power set 2^D of the set D of all $2^{|n|}$ bitvectors of length n and the subset relation $\subseteq$ are a partially ordered set. $(2^D, \subseteq)$ is a complete lattice, because for any subset $S \subseteq 2^D$, $\bigcup S$ and $\bigcap S$ are in 2^D ($\bot = D$, $\top = \emptyset$).

2. The ascending chain condition trivially holds for the lattice $(2^D, \subseteq)$ because it is finite.

3. All data flow functions are monotone, because
   - if-condition blocks only filter out some of the data flow facts for a branch, that do not satisfy the condition and
   - assignments only ever change one position in every bitvector in a set of bitvectors, which can only cause the set to decrease in size, because bitvectors can turn out to be equal that were not equal before. One bitvector cannot turn into two different ones, without causing the set to split.

   If-conditions and assignments are the only two relevant manipulating blocks, because for all other blocks the flow function is the identity.

The combined analysis is distributive, because the meet operator $\uplus$ is applied before each block and can therefore easily be formulated as an AFDS instance and its MVP solution can be calculated with the AFDA algorithm [4].

We saw, that a boolean value analysis is a straight forward way to splitting data flow information according to control structures in order to eliminate infeasible paths. There are other options, e.g. integer value analyses could be utilized in case counters are used within an asynchronous program, to check that a task is executed a certain number of times.

4.2 Conditional-Subset MVP Solution

We will take the path-sensitive approach a step further and generalize it based on the ideas of Ball and Rajamani [3]. Our goal is to formulate a formal definition of the path-sensitive solution of an interprocedural analysis for asynchronous programs.

In the first example of a reaching definition analysis in Figure 2.1, there is a condition `a < 2` that in an actual execution would cause the "else"-branch never to be executed. In a path-sensitive analysis that considers integer variables instead of just booleans, the infeasible path would be eliminated and the set of reaching definitions at `R` would just be $\{(a, 1)\}$.

The general idea behind value analyses is, to figure out all the possible values (intervals) a variable could have at any given point in the program, without actually executing the program itself. Boolean values can only accurately describe boolean variables or very roughly approximate facts about numeric values. The less approximations have to be made, the better the correlation between values and control structures. Whenever a control structure in the program depends on the value of

a variable, it is possible to split the data flow and make it path-sensitive based on that information. In essence, we can say that the more accurate the information in a value analysis, the higher the quality of path-sensitivity that can be achieved using this value analysis.

Based on the above observation that there is a more general approach to path-sensitivity than through boolean values, we can define the general *conditional-subset meet-over-all-valid-paths* solution, an extension of the MVP solution that contains only sets of data flow facts for which the path functions are path-sensitive.

CSMVP Solution. The conditional-subset meet-over-all-valid-paths (CSMVP) solution to an AIFDS instance $A = (\widehat{G}, D, F, M, \sqcap)$ and a set C of path correlating data flow facts with flow functions c_i, is a map $CSMVP_n : \widehat{V} \to 2^{2^D}$:

$$CSMVP_n = \biguplus_{\pi \in ScheduledP(n)} PF_\pi(\bot) \text{ for each } n \in \widehat{V},$$

with $F \subseteq 2^{2^D} \to 2^{2^D}$ the set of path-sensitive data flow functions.

The difference between MVP solution and CSMVP solution is mainly, that data flow information in the CSMVP solution is *sets-of sets-of* data flow facts, instead of just *sets-of* data flow facts. The inner sets are now correlated to the individual paths, i.e. every path in the path-sensitive solution corresponds to one subset solution $S \subset 2^D$ and the overall solution is a set of path solutions [3].

Ball and Rajamani go into further detail, by describing how binary decision diagrams can be utilized in algorithms that compute this sort of path-sensitive solutions [3]. There is also a relationship to "tri-vectors" that can be useful for the efficient implementations of a boolean value analysis [7]. Although interesting, these topics go beyond the scope of this section and we will not go into more detail about them.

We have seen that there is a general way of defining path-sensitive analysis solutions. Different approaches can be taken to gain accuracy, which is especially relevant with asynchronous programs. Experimentation will be necessary to figure out how feasible the different approaches turn out to be.

We think the boolean analysis provides a straight-forward way of achieving a degree of path-sensitivity that actually pays off in practice.

5 Evaluation

In this section we will talk about the implementation of the boolean analysis that was developed in the context of the thesis work. The boolean analysis was implemented as an extension of an existing analysis toolchain targeted at the nesC programming language. We will also see how the analysis performed in practice when applied to a set of programs that use boolean flags as guards and we will argue about its usefulness in applications.

5.1 Implementation

We will now introduce the toolchain that was extended to be path-sensitive in the context of this thesis and talk about what parts where extended in what ways.

The toolchain itself is written in Haskell, a programming language that because it is functional, allows for a very direct translation of data flow equations within the implementation. The target language is nesC.

In Section 3 we talked about how asynchronous programs can be used to describe the execution model of event-driven languages. One use case of event-driven programming is sensor-based devices, in which external sensor input is seen as an event that schedules a task to handle this input.

nesC extends the C programming language with interfaces to facilitate in developing sensor network applications on the TinyOS platform. The nesC execution model is event-driven and callback tasks are realized as part of the different interfaces that provide abstractions to hardware components and can be used to access sensors.

The execution model of nesC is relatively complex and only its application layer falls into the category of asynchronous programs. Furthermore, nesC differs from the notion of asynchronous programs described in the previous parts of the thesis in that it actually supports three different event types. Its task queue also allows only one instance of a task to be scheduled at one time, which makes for a different execution behavior than the one that was discussed so far.

All of the above makes for a good reason to use nesC as a testing ground of how our ideas apply to real world examples because it is not just a direct implementation of the boolean analysis. Also, the use of global boolean flags as guards for events is common in nesC and the example used in the previous sections is actually inspired by a program that is part of the TinyOS example programs library.

The toolchain and its analyzer work autonomously from the nesC compiler and it actually does not support real nesC programs yet, but a similar language that

contains all of nesC's relevant event system features. Figure 5.1 illustrates its design.

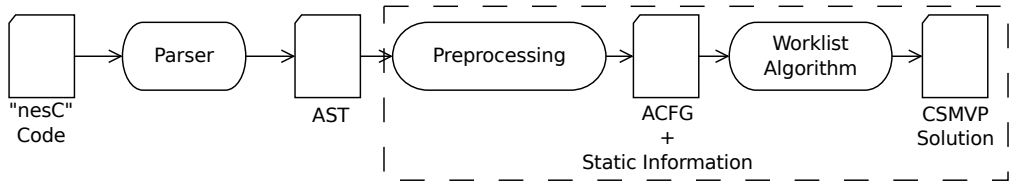

Figure 5.1: Analysis Toolchain

From left to right, the role of each symbol in Figure 5.1 and how it was extended in the context of this thesis can be described as the following:

- nesC Code: The toolchain is targeted at the nesC programming language, it does not fully support the real nesC grammar yet, just some of nesC's relevant features, most notably its three different categories of events.
- Parser: The parser is implemented using Alex and Happy, two Haskell tools that work as a parser generator. The extension that was made in this part was to parse global boolean variables as flags explicitly for the use in the boolean analysis. The idea behind handling them separately from other variables is to allow developers to achieve more accuracy in their analyses without any overhead by just using a certain naming convention.
- AST: The parser generates an abstract syntax tree. Up to here everything is just the same as what usually happens in a compiler, making the parts outside the dashed box in Figure 5.1 somewhat redundant. The nesC compiler is different from what one might expect in that it generates actual C code and uses a C compiler to compile the generated code with optimization.
- Preprocessing: The preprocessing step involves generating the asynchronous control flow graph from the abstract syntax tree and obtaining all necessary information for the analysis. This part of the toolchain was extended to actually generate the *conditional* asynchronous control flow graph and to count and order the global boolean flags as necessary for the boolean analysis.
- The worklist algorithm was extended to work on data flow triples consisting of counter facts, boolean value facts and facts for the actual analysis (e.g. reaching definition analysis). This is also the part where the actual data flow equations of the boolean analysis were implemented.
- The result is the CSMVP solution based on the assumption of a distributive analysis. In the case of this toolchain the output is not processed or used for any further computations. The result could be used for code optimization or debugging purposes in the future.

Haskell provides QuickCheck, a library for random testing of program properties. We used QuickCheck to test some of the most important specifications and properties of the boolean analysis such as the distributivity of the flow equations. Because Haskell is a functional programming language the formulation of mathematical properties is a straight-forward process.

Based on the property tests and a growing library of test-program and analysis result pairs we trust the extended toolchain to be a correct implementation of the boolean analysis formulated within this thesis without further verification of the code.

5.2 Test Suite

We will now introduce the test suite that was used to test the actual analysis based on the assumption that the implementation is a correct encoding of the algorithm that computes the solution to the analysis.

The focus in the evaluation of the toolchain was on programs that actually use global boolean flags to influence interprocedural control flow. Specifically, the programs we evaluated always consist of a `main` task that posts a certain number of other tasks that use boolean flags to enforce only one legal order of execution. The chain of tasks is enforced by having a boolean flag for every task, that has to be `true` in order for its "actual" control flow to be executed. If the guarding flag is `false` and the task is dispatched, it just gets re-posted, which causes it to "wait" for its guarding flag to finally turn `true`.

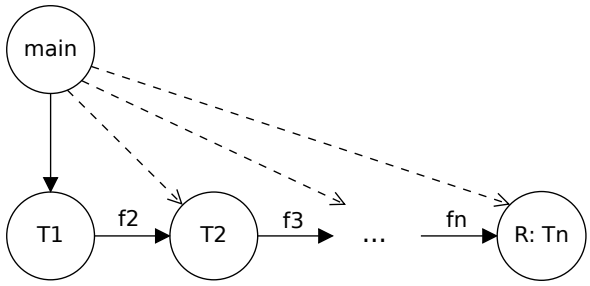

Figure 5.2: Test Suite

Figure 5.2 illustrates the enforced execution order using regular edges. The dashed edges picture the asynchronous calls made from the `main` task. A behavior like this can be enforced when the individual tasks look somewhat like the following piece of code:

```
task Ti:
  if fi then
    f(i+1) := true
    Ri: ...
  else
    post Ti
```

We analyzed programs with different amounts `n` of tasks and flags put together in a chain in the way described above in order to see how the "depth" of an invalid path influences the analysis result. In addition to the number of paths arriving at the label of interest `Rn` in the last task of the chain `Tn`, we also counted the number of iterations the worklist algorithm took until termination in order to be able to argue about efficiency in those cases.

The test suite is one specifically chosen to test the properties of the boolean analysis and is not necessarily a valid representation of asynchronous programs in general. The tested programs are a valid representation of the parts of a program that use what the boolean analysis is aimed at. We think that our test suite is sufficient and useful because for those parts of a program that do not involve boolean flags the overhead is negligible.

5.3 Results

We will now present the solution to our experiments and discuss its results. The following table compares the number of paths that are considered at a label `R` for an analysis over a program with a certain number of tasks executed in chain, with `R` being in the last task of the chain. The task chains are constructed exactly the way described in the previous section.

Tasks	1	2	3	4	5	6	7	8	9
Paths to `R`	1	2	4	8	16	32	64	128	256
Paths to `R` (path-sensitive)	1	1	1	1	1	1	1	1	1

The number of paths considered at the label of interest `R` is as expected from the "completeness" of the ACFG. For the analysis without path-sensitivity it grows exponentially compared to the length of the task chain. For the path-sensitive analysis only the one feasible path is considered in any case.

The next table contains the number of iterations a worklist algorithm requires to terminate for the analysis of an asynchronous program with a task chain of a certain length.

Tasks	1	2	3	4	5	6	7	8	9
Iterations	13	32	61	100	149	208	277	356	445
Iterations (path-sensitive)	12	28	54	89	133	186	248	319	399

The results in this table are more interesting because they might not be as expected. Usually, there is a trade-off between efficiency and accuracy in data flow analyses [3]. Just as the first table confirms, the boolean analysis is more accurate, so one would normally expect it to be less efficient.

The experiment shows it to be otherwise at least for the category of programs that were tested. We assume this to be the case because some paths could be eliminated very early on, causing some of the longer invalid paths not having to be analyzed at all.

Because of the way the analysis of asynchronous programs works the insight that a gain of efficiency is possible may turn out to be especially valuable. As we explained in Section 3, in order to reach a fixpoint the analyses of asynchronous programs may have to be computed multiple times for increasing values of k such that the counters count to k. Eliminating invalid paths early on is especially valuable in this context because some of the invalid paths could cause for an analysis to arrive at a fixpoint very late in the worst case, due to the fact that counters would have to count to high values of k in those cases.

An extreme example is the value analysis in which an integer is only increased by one on a path on which a task posts itself. The increased value of the integer would cause changes in the data flow facts to be re-propagated to that task over and over again. If such a path is invalid it must be eliminated early on. The above is a very extreme example of how a path-sensitive analysis can become very useful and a case in which one might consider adding it just for the purpose of allowing the analysis to terminate at all.

Overall the results from the test suite have satisfied our expectations. We should notice that the test suite and the results in this section are not meant to represent the application of the boolean analysis to actual nesC programs.

6 Related Work

In this section, we will explain how the boolean analysis and the path-sensitive approach presented in this thesis fit into a context of prior work.

Although the aspect of interprocedurally valid paths for synchronous interprocedural calls was ignored in this thesis, the work by Reps, Horwitz and Sagiv on precise interprocedural data flow analysis via graph reachability [6] can be seen as its foundation. In their paper, Reps, Horwitz and Sagiv introduce the interprocedural finite distributive subset IFDS framework and an algorithm for computing solutions for IFDS instance via context-free graph reachability. They define the meet-over-all-valid-paths solution for interprocedural programs.

Based on the MVP solution to IFDS instances Jhala and Majumdar present their MVP solution to asynchronous IFDS (AIFDS) programs [4]. The AFDS framework that is presented within this thesis is based on the AIFDS framework introduced in their work and is equivalent except that it ignores synchronous calls. Jhala and Majumdar also describe the counter analysis and the AFDA algorithm that can be implemented on top of the RHS algorithm and, thereby, extend the work of Reps, Horwitz and Sagiv for asynchronous programs.

The approach to path-sensitivity taken in this thesis is inspired by the Bebop model checker for boolean programs [3, 1]. Ball and Rajamani present a path-sensitive version of the RHS algorithm and also extend on the work of Reps, Horwitz and Sagiv. The conditional-subset meet-over-all-paths solution is introduced in their work and this thesis applies this to the interprocedural case for asynchronous programs. Ball and Rajamani also utilize binary decision diagrams for more efficient path-sensitive approaches, a technique we also used in the implementation of the boolean algorithm.

The approach to path-sensitivity presented in this thesis is the attempt of effectively combining the ideas of Jhala and Majumdar with those of Ball and Rajamani by solving the AIFDS instance not with the RHS algorithm, but an algorithm that is similar to the SP_{rhs} algorithm in "Bebop: A Path-sensitive Interprocedural Dataflow Engine" [3].

7 Conclusion

The goal of the thesis was to improve an existing interprocedural analysis for asynchronous programs by considering global boolean flag values to make it path-sensitive and eliminate invalid data flow.

We discussed the problem of infeasible paths in interprocedural analyses of asynchronous program and presented a solution. By combining analyses with a simple boolean value analysis we have made them path-sensitive and managed to achieve the overall gain in performance we desired.

We saw that despite the common trade-off between precision and efficiency a path-sensitive approach to interprocedural analysis can actually make for a gain in both for some asynchronous programs.

The boolean analysis approach is only useful for asynchronous programs that actually use global boolean flags to control interprocedural execution order. We did not test memory usage for real world applications.

There are several next steps that can be taken in future work.

We think it is necessary to do some further experimentation to see how a path-sensitivity of the boolean analysis influences the ADFA algorithms behavior for deep call counter values, especially concerning memory usage. Such experiments could possibly deliver interesting results based on the reasoning provided in Section 5.

Since other value analyses can also be used for path-sensitivity it would be interesting to see how they perform in combination with other analyses, especially with tools that perform value analyses anyways.

There is also the approach of translating interesting properties of a program into boolean values as used by the Bebop model checker [2]. An example of this would be formulating a *points-to-null* flag for pointers in a C program. The boolean analysis presented in this paper could be utilized to do model checking over asynchronous boolean programs.

Bibliography

[1] Ball, T., and Rajamani, S. K. Bebop: a symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification* (London, UK, UK, 2000), Springer-Verlag, pp. 113–130.

[2] Ball, T., and Rajamani, S. K. Boolean programs: a model and process for software analysis. Tech. rep., Microsoft Research, Feb. 2000.

[3] Ball, T., and Rajamani, S. K. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), PASTE '01, ACM, pp. 97–103.

[4] Jhala, R., and Majumdar, R. Interprocedural analysis of asynchronous programs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 339–350.

[5] Khedker, U., Sanyal, A., and Karkare, B. *Data flow analysis: theory and practice*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[6] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 49–61.

[7] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), POPL '99, ACM, pp. 105–118.