A Path-Precise Analysis for Property Synthesis

Sean McDirmid and Wilson C. Hsieh

UUCS-03-027

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

December 1, 2003

Abstract

Recent systems such as SLAM, Metal, and ESP help programmers by automating reasoning about the correctness of temporal program properties. This paper presents a technique called **property synthesis**, which can be viewed as the inverse of property checking. We show that the code for some program properties, such as proper lock acquisition, can be automatically inserted rather than automatically verified. Whereas property checking analyzes a program to verify that property code was inserted correctly, property synthesis analyzes a program to identify where property code should be inserted.

This paper describes a path-sensitive analysis that is precise enough to synthesize property code effectively. Unlike other path-sensitive analyses, our intra-procedural **path-precise analysis** can describe behavior that occurs in loops without approximations. This precision is achieved by computing analysis results as a set of **path machines**. Each path machine describes assignment behavior of a boolean variable along all paths precisely. This paper explains how path machines work, are computed, and are used to synthesize code.

A Path-Precise Analysis for Property Synthesis

Sean McDirmid and Wilson C. Hsieh
School of Computing, University of Utah
50 S. Central Campus Dr.
Salt Lake City, Utah USA
{mcdirmid,wilson}@cs.utah.edu

ABSTRACT

Recent systems such as SLAM [3], Metal [11], and ESP [8] help programmers by automating reasoning about the correctness of temporal program properties. This paper presents a technique called **property synthesis**, which can be viewed as the inverse of property checking. We show that code for some program properties, such as proper lock acquisition, can be automatically inserted rather than automatically verified. Whereas property checking analyzes a program to verify that property code was inserted correctly, property synthesis analyzes a program to identify where property code should be inserted.

This paper describes a path-sensitive analysis that is precise enough to synthesize property code effectively. Unlike other path-sensitive analyses, our intra-procedural **path-precise analysis** can describe behavior that occurs in loops without approximations. This precision is achieved by computing analysis results as a set of **path machines**. Each path machine describes assignment behavior of a boolean variable along all paths precisely. This paper explains how path machines work, are computed, and are used to synthesize code.

Keywords

Path-sensitive analysis, data-flow analysis

1. Introduction

Many program verification tools [3, 6, 8, 9, 10] have recently been designed to help programmers reason about the correctness of temporal program properties. Examples of such properties are adherence to synchronization, consistency, and resource allocation protocols. Many of these properties, such as proper lock acquisition, are also ideal candidates for automatic synthesis of code. In other words, given a description of where locking code should be inserted, compiler analyses can be used to insert the code automatically. In this paper we describe *property synthesis*, which can be viewed as turning automatic property checking on its head.

```
I = 0; J = 0; K = false;
while (true) {
 if (!hasNext(J)) break;
 J = next(J);

 Y = process0(J);
 if (J > I)
 { I = J; K = Y;}
}
if (K) process1(I);

return I;                      (a)
```

```
I = 0; J = 0; K = false;
while (true) {
 if (!hasNext(J)) break;
 J = next(J);
 if (!K) acquire(LK);
 Y = process0(J);
 if (J > I)
 { I = J; K = Y;}
 if (!K) release(LK);    }
if (K) process1(I);
if (K) release(LK);
return I;                      (b)
```

Figure 1: A code fragment (a) and an implementation of a lock acquisition property in this code fragment (b).

Property synthesis requires a program analysis that is comparable to a programmer's "best effort." As an example, consider the code in Figure 1 (a), which is not synchronized. This code traverses a list, processes elements, and keeps track of the largest element for future processing. Inserting synchronization code results in the code in Figure 1 (b). The locking property requires that lock `LK` be held for each element from before a `process0()` call until after a matching `process1()` call. If `process1()` will not be called for the element, then the lock can be released after the `process0()` call is made. A good implementation of this property releases the lock as soon as it is known that `process1()` will not be called on an element that was already processed. This example demonstrates that property synthesis is general enough to handle arbitrary looping behavior.

The tricky part about this example are the loop-carried dependencies created by assigning `I` and `K`. Whenever `I` and `K` are assigned to `J` and `Y`, the lock can be released if `K` happens to be false. For property synthesis to be effective, this behavior must be identified via analysis. However, we know of no existing analysis that can identify lock release opportunities this precisely.

Our technique for property synthesis is performed in three steps. The first step automatically adds annotations that setup property-specific analysis problems, as well as property-specific code whose reachability depends on solutions to these analysis problems. A generic description of these annotations is expressed in a meta-programming language that is outside the scope of this paper: consider it to be similar to AspectJ [13]. The second step solves the analysis problems specified in the annotated program. In the final step, solutions computed during analysis are used to determine when property-specific code added in the first step is reachable, which results in a modified program that implements the property.

This paper describes the analysis step of property synthesis, which must compute results with enough precision to implement a property effectively. Our analysis technique focuses only on examining the scalar boolean assignment behavior of a procedure. As noted by Ball and Rajamani, reasoning about behavior related to a temporal safety property can be reduced to boolean assignment problems that are cheaply and precisely analyzed [2]. The annotations in the first step are expressed as boolean variables that are assigned in strategic places in the program. Our annotations can even express backward analysis problems.

We have developed an intra-procedural "path-precise analysis" (PPA) that solves boolean assignment problems. PPA is similar to other path-sensitive analyses [5, 8, 12, 14, 15, 17] that qualify results to distinguish between the behaviors of different execution paths. Traditional meet-over-all-paths data-flow analyses (DFAs) are limited to describing behavior according to points in a control-flow graph (CFG). Path-sensitive analyses improve the precision of DFA with simple boolean path predicates that can distinguish between some paths in a CFG. However, these predicates are not powerful enough to distinguish between the different behaviors of multiple looping execution paths.

PPA handles looping paths by computing analysis results as a set of *path machines*, which are finite-state machines that each describe boolean assignment behavior for one variable. A path machine is a state-transition system that can describe arbitrary looping execution paths in a procedure. By specifying states where boolean variables are true or false, path machine effectively describe the control-flow of boolean assignment behavior, rather than describing such behavior according to a CFG. The worst-case time complexity of PPA is $O(E^L)$, where E is the number of boolean assignments being analyzed and L is the level of loop nesting in a procedure.

In this paper, we use proper lock acquisition as the primary example of a property that can be synthesized. However, property synthesis is applicable to other kinds of properties as well. Error-handling protocols are properties that require reasoning about potential failures that occur between how some data is generated and is used. For example, data from user input must be validated before it can be used to update a database. Consistency protocols are properties that require reasoning about when some operation has occurred so some other operation can invalidate or update assumptions. For example, repainting of a GUI widget must occur after its visible state has been updated. Code for both error handling and consistency protocols can be addressed by property synthesis.

The rest of this paper is organized as follows. Section 2 describes the four steps of property synthesis. Section 3 describes an algorithm for path-precise analysis that operates over boolean assignment procedures to compute path machines. Section 4 discusses issues related to property synthesis and path-precise analysis. Related work and our conclusions are presented in Sections 5 and 6.

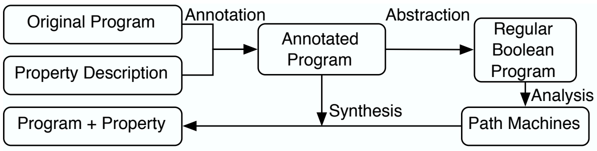

Figure 2: An overview of property synthesis steps.

2. Property Synthesis

Our technique for property synthesis is a multi-step process that is illustrated in Figure 2. The annotation step takes a program and a property description and produces a version of the program annotated with property details, which is suitable for analysis but not execution. The abstraction step transforms the annotated program into a regular boolean program, which only describes the annotated program's control flow and boolean assignment behavior. In the analysis step, the regular boolean program is analyzed to compute a model of path machines, which precisely describes the boolean assignment behavior of the annotated program. Finally, the synthesis step uses the model of path machines to transform the annotated program into an executable program that implements the property.

2.1 Annotation

Property descriptions consist of two parts: annotations that express an analysis problem and property code whose reachability depends on a solution to the analysis problem. Annotations are added to a program according to a description of the property expressed as a set of rules. The rules specify how annotations are added according to landmarks that can be identified in a program using a flow-insensitive analysis. For example, a generic description of the lock acquisition property implemented in Figure 1 (a) can be expressed as the following two rules:

1. Before procedure `process0()` is called, the lock `LK` must be acquired if it is not held already;

2. Procedure `process1()` can be called after a `process0()` call with the same index only if lock `LK` is held over the span of the two calls.

A meta-programming language for expressing property descriptions is not described in this paper. Such a language needs to associate annotations with easily identified points in the program, in a manner similar to how advice is added at join points in the aspect-oriented programming language AspectJ [13].

Property code added during annotation specifies actions that should be performed in certain situations: acquire a lock before a call to `process0()` when the lock is not already held. Situation descriptions can take into account the location in code, such as before a `process0()` call site, or invariants over program behavior, such as whether a lock is held. Whether an invariant is satisfied is determined by testing *annotation variables*, which are boolean variables tested and assigned for analysis purposes only.

Lock `LK` can be released when no value used in a `process0()` call can flow to a `process1()` call. Therefore, annotation variables are needed to track how values flow through the code. Adding such annotations to the code in Figure 1 (a) results in

```
    I = 0; J = 0; K = false;
RA  if (!?USED && LK_H)
    { LK_H=false; release(LK); }
    while (true) {
X    X = hasNext(J); if (!X) break;
RB   if (...) {LK_H=false;release(LK);}
N    J = next(J);
Q    if (!LK_H) {LK_H=true;acquire(LK); }
Y    Y = process0(J); J_IS = J;
RC   if (...) {LK_H=false;release(LK);}
Z    Z = I > J;
RD   if (...) {LK_H=false;release(LK);}
M    if (Z) { I = J; K = Y; I_IS = J_IS;} }
RE  if (...) {LK_H=false;release(LK);}
P   if (K) { process1(I); USE = I_IS; }
RF  if (...) {LK_H=false;release(LK);}
RT  return I;
```

Figure 3: The code from Figure 1 (a) with lock property annotations. Labels for each line are specified on the left.

the code of Figure 3. For illustration purposes, the lines are labeled in this code, and the labels will be used to refer to this code throughout the synthesis process.

The J_IS and I_IS variables keep track of elements that are used in process0() calls. When process0() is called at line ***Y*** with the variable J as an argument, J_IS is assigned to J. Though J is not a boolean value in the program, it will later be considered as a boolean *select value*, which will be explained in Section 2.2. When the value in J is assigned to variable I at line ***M***, the variable J_IS is assigned to I_IS to specify that I is now an alias of J. The variable USE tracks when elements are used in a process1() call. The I_IS variable is assigned to the USE variable whenever the process1() call executes.

Figure 3 also shows the lock release and acquire property code adding during annotation. The annotation variable LK_H tracks whether lock LK is held. At line ***Q***, just before process0() is called, annotation code tests the LK_H variable, acquires the lock if it is not held, and sets LK_H to true. Lines ***RA***, ***RB***, ***RC***, ***RD***, ***RE***, ***RF***, and other lines not illustrated in Figure 3 release the lock. Because a lock release might occur anytime the specified invariants are satisfied, the code is inserted between every statement of the program. During synthesis, most of the lock release code will be eliminated because they are never reachable during execution. In Figure 3 the lock release condition is expressed at line ***RA*** and abbreviated for other lock release lines. Lock release occurs when the lock is held (LK_H), and when no values used in previous process0() calls can be used in future process1() calls. This last condition is expressed as !?USE, which is true only if variable USE cannot be assigned to true in the future. The future operator ? enables reasoning about the possibility of an event occurring in the future via "reachability" relationships, which can be computed by a static program analysis.

```
BGN  if (L0)                         L0 = 0, L1 = 1;
  X  if (L1)                  X = *, L1 = 0, L2 = 1;
  N  if (L2 && X)             J = @, L2 = 0, L3 = 1;
  Y  if (L3)         Y = *, J_IS = J, L3 = 0, L4 = 1;
  Z  if (L4)                  Z = *, L4 = 0, L5 = 1;
  V  if (L5 &&  Z)K = Y, I_IS = J_IS, L5 = 0, L1 = 1;
NOP  if (L5 && !Z)                   L5 = 0, L1 = 1;
  P  if (L2 && !X &&  K)  USE = I_IS, L2 = 0, L6 = 1;
NOP  if (L2 && !X && !K)             L2 = 0, L6 = 1;
 RT  if (L6)                         L6 = 0;
```

Figure 4: An RBP of the annotated code in Figure 3; 0 is true, 1 is false, * is an unknown values, and @ is a select values. Initially all variables except L0 are false. Annotation code line labels for each instruction are listed on the left.

2.2 Regular Boolean Programs

After a program has been annotated with property details, it is abstracted into a *regular boolean program* (RBP). An RBP is similar to a boolean program [2], where the only data type is boolean. Unlike a boolean program, an RBP is a set of unordered guarded assignments to boolean variables. Each procedure in a program is expressed and analyzed separately in an RBP; i.e., RBPs do not support inter-procedural analysis.

The annotated code in Figure 3 is translated into the RBP in Figure 4. Variables are added to the RBP to track control flow. Following each guard is a set of assignments to boolean expressions that can refer to variables and *unknown values* (*). Unknown values in boolean programs [4] are non-deterministically true or false. They express abstracted details about a program's behavior that are either unknown statically, or are otherwise computationally too complex to be expressed in an RBP; e.g., general arithmetic. In Figure 4, unknown values are used to describe the unknown behavior of calls to hasNext() and process0(), and the expression j > i, which are assigned to variables X, Y, and Z.

Assignments in RBP instructions can also occur to *select values*, which are similar to unknown values in that they are non-deterministically true or false. However, a select value will be true exactly once per instruction it is assigned. Select values are used to describe non-boolean values whose identities are significant even when they are created in loops; e.g., the next() result in Figure 3. Values created in a loop can have relationships with each other that are not shared across multiple loop iterations. For example, the next() result in Figure 3 is related to the process0() result created in the same loop iteration. If the loop breaks and some next() result is assigned to I, then the process0() result from the same iteration is assigned to K. Therefore, whether a process1() call is reachable for some next() result depends on the process0() result in the same iteration being true. To maintain these relationships, a select value is true only for some arbitrary loop iteration during analysis, which can then be generalized to all loop iterations. How analysis later handles select values is crucial to reasoning about behavior in loops, and will be described in depth in Section 2.3.

Guards are the only way to represent control flow in an RBP. Control flow of an RBP is very simple and can be described with respect to a clock. On every tick of the clock, every instruction is executed, although the guard for only one instruction is ever satisfied per clock tick. Each instruction is guarded by a *CFG variable*, which corresponds to when the RBP instruction executes in the annotated code's CFG. In Figure 4, all variables prefixed by an L are CFG variables. Initially, all

CFG variables are false except for L0, which acts as the entry point. Each RBP instruction is also associated with a line label from the annotated code, which we use to identify an RBP instruction. The last two assignments of each RBP instruction in Figure 4, except for instruction ***RT***, assign one CFG variable to false and assign another CFG variable to true to implement control flow. By incorporating CFG information into an RBP through assignment and guards, control-flow analysis is combined with assignment analysis. To represent procedure termination, all CFG variables are set to false, which is what occurs at instruction ***RT***.

Since an RBP can implement arbitrary non-recursive branching (e.g., gotos), transforming a procedure into an RBP is simple. Using a flat representation of a procedure's code, such as in a register transfer language (RTL), all instruction labels are transformed into CFG variables that are always assigned to false unless the instruction should execute. All registers or variables in the RTL become variables in the RBP. All boolean expressions that are not formed using logical operators are translated into unknown values. Non-boolean expressions are translated into select values.

RBPs are useful because their analysis semantics can be described according to a finite state-transition system that processes strings of unknown values. Using the clock analogy to describe RBP behavior, every clock tick is associated with an arbitrary boolean value that determines whether an unknown value assigned on that clock tick is true or false. These clock ticks form the basis of a simple boolean-state transition system. The truth values of transitions in an execution path describe what assumptions execution paths depend on about unknown value behavior, which do not exeactly correspond to branches taken.

2.3 Path Machines

A *path machine* describes boolean variable assignment behaviors in an RBP through a state-transition system. Intuitively, a path machine describes assignment behavior with respect to feasible execution paths through the code being analyzed. Path machines are computed by analyzing the RBP representation of a procedure, and are precise with respect to the RBP. The path-precise analysis that computes path machines is described in Section 3.

One path machine describes one variable in the RBP. Path machines are deterministic binary-transition finite-state machines (FSMs) whose accept states specify where in a program's execution a variable is true and non-accept states specify where a variable is false. Path machines support all the basic FSM operations, which correspond to boolean operations over variables being described: FSM complement is logical negation, FSM union is disjunction, and FSM intersection is conjunction. For example, the expression !?USE || LK_H is described by a path machine that is formed by complementing the path machines for variable ?USE, and unioning it with the LK_H path machine. Path machines can also be minimized using traditional FSM minimization algorithms.

A path machine is shown in Figure 5 that describes the assignment behavior of variables K. For illustration purposes only, if both of the transitions from one state go to the same state, the transitions are illustrated as one unlabeled transition. Branching transitions are labeled with + for true or - for false. Also, states are labeled according to the RBP instructions whose guard is satisfied, or executes, on the state's transitions. RBP instructions are labeled according to line labels in the annotated code of Figure 3; e.g., the start state is labeled ***BGN*** because on the first transition the assignments of instruction ***BGN*** will be evaluated. Branching occurs from a state in a path machine when path behavior diverges on the basis of an unknown value created at the state. For example, because the Y variable can be assigned to K and the path machine tracks K's value, the unknown value computed at state ***Y***$_0$ can change the value of K. The K path machine describes how K is initially true and becomes false after ***N***$_0$ executes. In order for K to go from true to false in one loop iteration, the call to hasNext() and evaluation of i > J must be true at states ***X***$_0$ and ***Z***$_0$, and a call to process0() at state ***Y***$_0$ must be false.

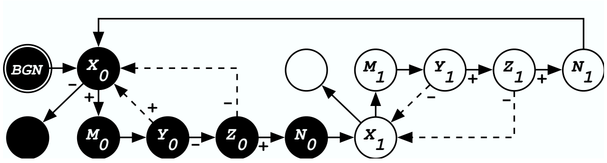

Figure 5: A path machine that describes how variable K is assigned in Figure 4. States where the variables are true are black background with white foreground, and the start state has an extra circle around it. States are labeled according the RBP instructions they correspond to and are subscripted when labels are duplicated; unlabeled states illustrate behavior that continues forever and transition to themselves. Dashes abbreviate multiple transitions that do not affect behavior.

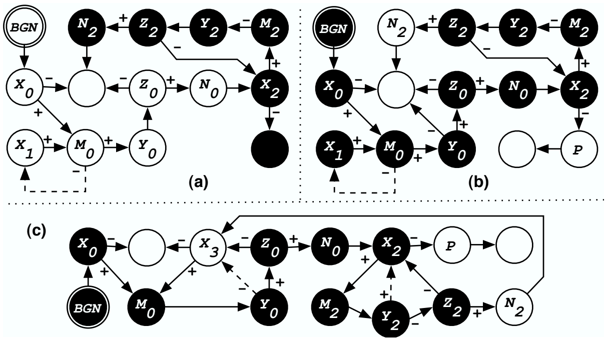

Figure 6: Path machines for variables I_IS (a) and ?USE (b), and the path machine ?USE (c) after it undergoes select value elimination.

The path machines in Figure 6 demonstrate the semantics of select values and the future operator ?. A select value is similar to an unknown value except that it accepts exactly one true transition per label. In Figure 4, a select value is created at instruction ***M***. Any path that passes through states with the ***M*** label must contain exactly one true transition from one of these states. The I_IS path machine in Figure 6 (a) describes I_IS assignment behavior, where I_IS is assigned to a select value at instruction ***N***. Because state ***M***$_0$ creates a select value that determines whether I_IS becomes true, a branch is made. If the select value is false, then the path loops through the ***M***$_0$ state until the select value is true. The false transition from state ***X***$_1$ is not part of a feasible path, because then the loop would exit and the select value could not be true exactly once. If hasNext() never returns true, indicating that the list is empty, then the false transition occurs from state ***X***$_0$ and state ***X***$_1$ is never reached.

```
    I = 0; J = 0; K = false;
RA  if (!?USE&&LK_H) release(LK);      false
    while (true) {
 X   X = hasNext(); if (!X) break;
RB   if (!?USE&&LK_H) release(LK);     false
 W   W = next();
 Q   if (!LK_H) acquire(LK);           !K
 Y   Y = process0(W);
RC   if (!?USE&&LK_H) release(LK);     !K&&!Y
 Z   Z = compare(V,W));
RD   if (!?USE&&LK_H) release(LK);     !K&&Y&&!Z
                                       ||K&&!Y&&Z
 V   if (Z) { I=J; K=Y;}             }
RE  if (!?USE&&LK_H) release(LK);      false
 P  if (K) { process1(I);    }
RF  if (!?USE&&LK_H) release(LK);      K
RT  return V;
```

Figure 7: Annotated code from Figure 3 without LK_H variable assignments; the right column specifies under what conditions annotation code executes using a boolean expression.

Once the true transition through M_0 is taken and if the evaluation of `i > j` at state Z_0 is true, then `I_IS` is assigned to true at state N_0. The path can continue through state M_2. Since a true transition for a state with the label M was already considered, only a false transition can be considered from state M_{2b}. If state N_2 is reached, then the select value assigned to `I_IS` at that state will always be false.

The visual structure of the `I_IS` path machine in Figure 6 (a) shows that `I_IS` is only contiguously true, so only one assignment of `I_IS` to one `next()` result is described. Other assignments of `I_IS` to a `next()` result are false because the select value is only true once. As a result, the `I_IS` path machine describes when a `next()` result in an arbitrary iteration is assigned to variable `I`, and how long it remains assigned to `I` in successive loop iterations. It does not describe whether `I_IS` is assigned to "some" `next()` result.

The path machine for variable `?USE` is shown in Figure 6 (b), and has a structure similar to the `I_IS` path machine. Variable `?USE` is true in states that can reach state P, which calls `process1()` while `I_IS` is still true. The non-determinism of a select value must be eliminated from a path machine before it is used for synthesis. A select value is eliminated by unioning the behavior of path segments where the select value is assumed true into the behavior of path segments where the select value is assumed false. Intuitively, the result is a path machine that specifies behavior about an expression value for all loop iterations, not just an arbitrary iteration. The `?USE` path machine with its select value eliminated is shown in Figure 6 (c). This path machine is true when a `next()` result created in state M_0 or M_2 can reach state P with a true `process0()` result created at state Y_0 or Y_2 in the same iteration. Otherwise, the path machine is false and the lock can be released.

2.4 Synthesis

The synthesis step uses path machines to determine when property code is reachable during execution. Property code can be resolved to one of three conclusions: it always executes, in which case it is left in the program; it never executes, in which case it should be removed from the program; or it executes conditionally, in which case it is left in the program with a dynamic test. Figure 7 lists resolution results for the propery code in Figure 3 in its right column. The Q line of lock acquire code executes when `K` is false, the RF line lock release code executes when `K` is true, and the RA, RB, and RE lines of lock release code never execute. Lock release code at lines RC and RD execute according to expressions over `K`, `Y`, and `Z`.

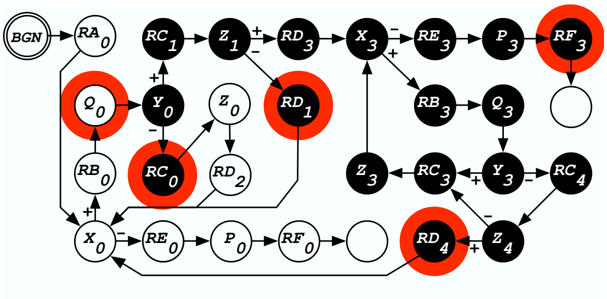

Figure 8: The path machine for the LK_H variable (select values eliminated); Q and R states that correspond to lock release and acquire are highlighted, and occur whenever LK_H goes between true and false; states with labels M and N are not shown.

Conditions under which annotation code can execute are determined by looking at the guard in its controlling `if` test, and implementing the guard expression with path machines. Given property code located at some label, a path machine that describes its guard behavior, and states in the path machine with the same label as the code, the following can occur: the states are all true, in which case the code executes unconditionally; the states are all false, in which case the code never executes; or some of the states are true and some of the states are false, in which case code execution is contingent on conditions known at run time. For the last case, conditions are never derived from annotation variables, which are always removed from the program. However, they can be derived from original variables or new variables added to the code specifically for dynamic testing.

Figure 8 illustrates the path machine for the variable `LK_H`. Lock acquisition occurs only in state Q_0 and lock release occurs in states RC_0, RD_1, RD_4, and RF_3. Instruction labels correspond to static locations in the CFG, so states can always be disambiguated by their labels at run time using CFG information. However, multiple states with the same label cannot be distinguished using the CFG alone. When property code executes in a state that is identified by an ambiguous label in a path machine, it must be guarded to ensure that it executes only in the right states. This condition can be computed by tracking transitions in a path machine at run time. For example, state RC_0 can be disambiguated from states RC_3 and RC_4 by noting that if the true transition from Z_1 is taken, then RC_3 will not execute until the true transition from state Z_4 is taken. This expression corresponds to the current value of the `K` variable, where RC_3 will not execute unless `K` is false. State RC_3 is only entered from the false transition of the Y_0 state, so the expression that guards RC_3 is equivalent to `!K && !Y`. However, because this expression uses multiple variables, it is not feasible

A	`Z = false;`	`if (L0)  Z = 0, L0 = 0, L1 = 1;`
	`while (true) {`	
B	`X = !Z;`	`if (L1)  X = *, L1 = 0, L2 = 1;`
	`while (true) {`	
C	`Y = getY();`	`if (L2)  Y = *, L2 = 0, L3 = 1;`

D	`if (Y == Z)`
	`break;`
		`if (L3&&(!Y&&!Z))`
		`L3 = 0, L1 = 1;`

E	`Z=Y&&X; X=Y;`
	`}}`
		`if (L3&&(Y||Z))`
		`Z = Y&&X,X = Y, L3 = 0, L2 = 1;`

Figure 9: A code fragment with a two-level nested loop structure and its corresponding RBP; line labels are listed in the left column.

to reuse an existing variable to describe it. Instead, new variables and new assignments can be generated automatically to track the transitions that determine when the `!K && !Y` region of the path machine is entered.

Once path machines are used to resolve the reachability of property code, annotation variables are removed from the code. For Figure 7, this results in code that is similar to the code of Figure 1 (b). However, the solution computed by property synthesis is slightly more efficient and less obvious: it can release a lock before the `i > j` expression at line ***Z*** is evaluated. This replaces the one line of lock release code in the loop of Figure 1 (b) with three lines that enable the lock to be released a few instructions sooner.

3. Path-Precise Analysis

Path machines used in property synthesis are computed with a path-precise analysis that does not utilize any of the approximations used in a MOP (meet-over-all-paths) data-flow analysis. The only time approximation occurs in property synthesis is when the annotated code is transformed into an RBP, where unknown values approximate behavior that is either unknown or not expressible through boolean assignments. This section describes how path machines are computed from arbitrary RBPs without any loss of precision.

Each variable in an RBP is associated with one path machine. PPA computes path machines by tracing longer and longer paths through an RBP. The key insight is that a fix-point can eventually be recognized if path machine construction makes compact guesses about paths in the RBP longer than they are precise for. In other words, the path machines that are precise for paths of K length express behavior that is possibly wrong for paths of length $K+1$. When a path machine is refined to be more precise for a longer path, states must be reused to represent this precision via *state recycling* before new states are created, otherwise a fix-point can never be identified. State recycling is similar to FSM minimization, except that equivalence between the original and minimized machines is not as strong. Instead, state recycling only requires that the resulting path machine expresses the same behavior for execution paths for which the original path machines is precise.

As an example, consider the code in Figure 9 and a corresponding RBP, where a call to `getY()` is changed into an unknown value. The code contains two loops, where the outer loop never breaks, and two loop-carried variables, `X` and `Z`. In the outer loop, `X` is always assigned to `!Z`. In the inner loop, `Z` is assigned to `Y && X`, where `Y` is computed from `getY()`, as long as `Y` does not equal `Z`. Otherwise the inner loop breaks into the outer loop. Though this example does nothing useful, its looping behavior is complicated. Real code often exhibits looping behavior that is simpler; this example code demonstrates that path-precise analysis is general enough to handle arbitrary looping behavior.

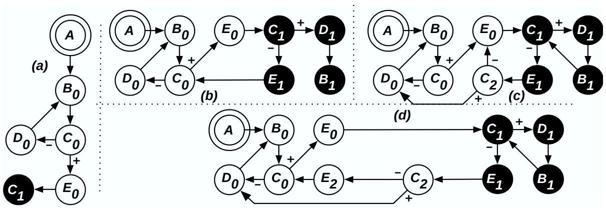

Figure 10: Intermediate steps (a) through (c) in computing the path machine for variable Z in Figure 9, and the final result (d).

Three intermediate steps in computing the path machine for variable `X` in Figure 9 are shown in Figure 10. Figure 10 (a) shows what the `X` path machine looks like after tracing arbitrary paths up to length 4. The trace proceeds as follows: instruction ***A*** is the first instruction to execute, followed by instruction ***B***. The next instruction to execute is ***C***, which creates a result that determines what instruction executes next. Therefore state ***C***$_0$ branches to two choices, state ***D***$_0$ (instruction ***D*** will execute when `Y` is false), or state ***E***$_0$. Following the execution of instruction ***D***, instruction ***B*** will execute again. An existing state is recycled in a path when it executes the same instruction and has the same truth value as a new state in the path would have. Given multiple recycling candidates, a well-defined order that takes into account distance from start and the branching structure of the path machine is used to choose a state. A well-defined order in choosing recycling candidates is important as it ensures no state is skipped when guesses are incorrect. Because `X` is false at state ***B***$_0$, it can be recycled for the transition from state ***D***$_0$.

Figure 10 (b) shows what the `X` path machine looks like after tracing arbitrary paths up to length 6. State ***C***$_0$ is recycled for the transition from state ***E***$_1$. This transition will be known to be incorrect when arbitrary paths of up to length 7 are traced. At this time it will be known that the ***D*** instruction, not the ***E*** instruction, succeeds the ***C*** instruction on a true transition when state ***E***$_1$ was executed two transitions before. When incorrect guesses are identified, path machines must be patched.

Path machine patching finds the next possible guess in the path machine that is precise with respect to K and recycles the most states using a well-defined order. In the case of Figure 10 (b), there is only the one false ***C*** state, so a new false ***C*** state must be created to capture the new path. This occurs in Figure 10 (c) with the new ***C***$_2$ state to express behavior precise for paths of length 7. In general, incorrect guess paths are always the result of assignment behavior that occurs only periodically in a loop. For example, consider a loop where a periodic assignment only occurs every four iterations when two counter-like boolean values are both true. Path machines would recycle three sets of states through the loop's transition path before the incorrect guess was discovered. Patching the path machine would require unrolling the existing cycle in the path machine three times so that special behavior can be considered in the fourth iteration. In Figure 10 (b), periodic behavior is identified on the second iteration, so the incorrect guess is discovered quickly.

Because incorrect guess paths can cycle for multiple iterations before periodic behavior is identified, path machines must be annotated with transition precision information to ensure that incorrect guesses can be recovered from. Each transition must be associated with a precision that specifies how many times it is precise in a path before the next transition taken cannot be guaranteed to be as precise. When a state is initially recycled from some state, the transition between these states is assigned a precision of zero, meaning the next transition is not guaranteed to be as precise. As tracing recycles a state multiple times in a path, the precision of the transition is incremented. Beyond enabling incorrect guess recovery, transition precisions are not used for any other purpose.

The Z path machine in Figure 10 (c) still makes an incorrect guess on the false transition from state $\boldsymbol{C}_2$ to state $\boldsymbol{E}_0$, where the ***E*** instruction is preceded in execution by a false ***C*** instruction, not a true one. When a trace of path length 8 is performed, the path machine is patched to the path machine in Figure 10 (d). This path machine is completely precise for any arbitrary length path, although the algorithm will only terminate after it traces arbitrary paths of length 9 (8 + 1) to discover that the X path machine and path machines for other variables have reached fixed-points. A path of length 8 has encountered all possible states in this RBP. The RBP has four instructions in the loop and two loop-carried variables (X and Z). Therefore, a path machine could duplicate an instruction at most only four times, which results in at most 17 states (including ***A*** outside the loop). PPA is guaranteed to terminate at or before a path length of 18 (17 + 1) for the RBP in Figure 9. Termination occurs at path length 9 because half of the possible states happen to be infeasible or redundant. The example in Figure 9 is an extreme case, where all variables are dependent on each other.

3.1 Fast PPA

An implementation of PPA simply by direct induction on path lengths explores every possible path of increasing lengths until a fix-point is computed. This results in a computationally inefficient algorithm where the number of paths explored is exponentially related to the number of unknown values in an RBP. The basic algorithm is analogous to a naive implementation of data-flow analysis (DFA), where a CFG has only one “dirty” bit and basic blocks are traversed repeatedly until the dirty bit is no longer set. As in a fast DFA algorithm, a fast PPA algorithm should focus on computing local fix-point behavior for semi-independent sections of the CFG, like inner loops, before tracing behavior for successive sections of the CFG. Although nodes will have to be processed multiple times, tracing an exponential number of paths can be avoided.

Control-flow in an RBP is implicit in variable assignment, so control-flow of an RBP is not known in a graph form until PPA is finished. However, a *control-flow machine* (CFM) is constructed and refined during PPA by computing the path machines for the RBP’s CFG variables, and it can guide analysis. A CFM is analogous to a CFG and is formed by unioning the path machines of all CFG variables together. The CFM is dependent on every variable used in control-flow tests, so will contain the complexity of those variable’s path machines. The CFM is not precise until PPA terminates, but even in its imprecise form, it can aid in identifying potential loops to focus analysis on. Because the loops PPA is concerned about occur between states, not between instructions, the CFM will be more precise than a CFG in guiding state traversals. In Figure 10 (d), the Z path machine actually consists of four sequential inner loops and one outer loop. The fast PPA algorithm maintains a set of dirty states relative to the CFM. Initially, only the CFM start state corresponding to the first evaluated instruction is in the dirty set.

On each iteration of the analysis, a state is chosen for processing from the dirty list based on two criteria. The higher priority criterion selects a dirty state based on the current loop being processed according to the CFM’s clique structure. The dirty state in the same loop with the last processed state will be processed before another dirty state. For example, if the CFM resembles the path machine in Figure 10 (a), if state $\boldsymbol{C}_0$ is the last state processed, and states $\boldsymbol{D}_0$ and $\boldsymbol{E}_0$ are dirty, then state $\boldsymbol{D}_0$ will be processed before $\boldsymbol{E}_0$. The lower priority criterion selects the nearest successor state, based on the CFM, of the previously processed state. Otherwise, a dirty state is just chosen at random. Processing traces all imprecise CFM paths through the processed state, patching the path machines according to the results of the traces. If any path machine is patched as a result of the trace, states whose evaluation follows the processed state in the CFM are added to the dirty list. A fix-point has been reached when the dirty list is empty.

3.2 Complexity

The worst-case feasible state space of an RBP is $O(E^V)$, where E is the number of instructions and V is the number of variables in a program. This determines the shortest path that can be traced before a fix-point is identified, which is the same for the fast or basic PPA algorithm. The worst case assumes that all variables V are related to each other’s behavior, which does not occur often in real programs.

The number of arbitrary paths for a given length is related to the number of unknown values U in a program. In the basic PPA algorithm, an instruction will be explored for each path it is in; therefore, the worst-case complexity with respect to state processing is $O(E^U)$. Because of its traversal technique, the fast PPA algorithm has a worst case time complexity of $O(E^L)$, where L is the looping depth in the CFM. Although it is possible to contrive programs where L approaches the size of U, loop nesting in the CFM is usually related to loop nesting in the CFG. However, it is always possible for loops in the CFG to only execute a constant number of times, which means they are not loops in the CFM. Alternatively, variables and branches may be used in a way that creates loops in the CFM that do not exist in the CFG. For the code in Figure 9, the code’s CFM is similar to the path machine in Figure 10 (d) indicates a loop nesting depth of two, which would also be the looping depth in its CFG. Since PPA is not inter-procedural, L is likely to be a small number, the exception being loop-intensive scientific code.

Each path machine describes behavior for only one variable. As a result, PPA avoids the state explosion problem often associated with constructing state machines that describe program behavior. Except for control-flow variables, which are obviously all inter-related, variables are often only loosely related to each other. Therefore, their path machines can be smaller and can arrive at fix-points faster than a single monolithic state machine that describes all variable assignment behavior together. Additionally, this also means PPA will suffer when false depen-

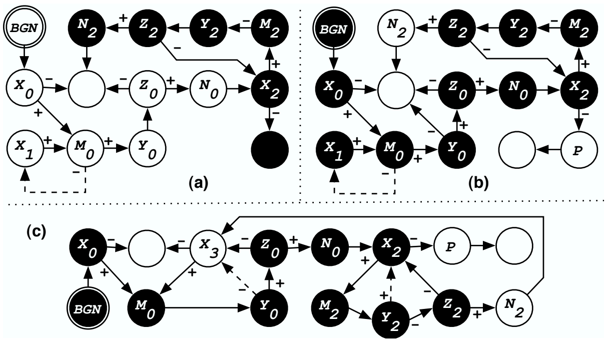

Figure 11: Repeated from Figure 8, the path machines for variables I_IS (a) and ?USE (b), and the path machine ?USE (c) after it undergoes select value elimination.

dencies exist between variables because they are reused for unrelated purposes. Before PPA is performed, renaming variables using a technique such as SSA [7] can potentially improve PPA performance by eliminating easily detected false dependencies.

3.3 Future Variables

Future variables are created when the future operator `?` is applied to a variable. Because of future variables, PPA occurs in multiple passes. On each pass, PPA computes path machines for a set of variables that are not dependent on variables in later passes. Because of this requirement, conventional and future variables can never be mutually dependent on each other. Additionally, future variables are restricted from ever being assigned to unknown values. Three analysis passes are needed to compute path machines for the code in Figure 3, where the second pass computes path machines for the `?USE` expression and the third pass computes the path machine for the `LK_H` variable, while the first pass computes paths machines for all other variables.

All CFG variables exist in the first pass, so by the second pass, which processes future variables, the CFM is already constructed. Whereas path machines for conventional variables are constructed from a start state, path machines for future variables are constructed from a termination state that all paths in the CFM conceptually can reach, even if they cycle forever. Starting from the termination state, assignments to the variable flow backward from their assigning states. An assignment to true flows backward, adding states to the true set, until an assignment to false of the same variable is encountered. After a path machine for a future variable is computed, it is identical in form and function to path machines of conventional variables.

3.4 Select Values

When the path machines are being constructed, select values are treated the same as unknown values that are true exactly once. To implement this constraint, all paths that take a true transition more than once for the same select value are eliminated. Additionally, all paths that take a false transition for a select value is eliminated if they do not eventually take a true transition for the select value. These criteria are identified during path machine construction.

Before a path machine is used to synthesize code, the non-determinism of a select value can and must be eliminated. Select value elimination occurs by unioning sections of the path machine where the value is true with sections of the path machine where the value is false. Select value elimination is a lossy process because it intuitively merges together all behavior about the iterations of an expression value created in a loop. Therefore, select values are only eliminated when a path machine is directly used to reason about the reachability of some property code. Before synthesis, select values enable these correlations between values produced in a loop to be maintained long enough to derive other path machines that are more precise as a result of the correlations. Consider the path machine for variable `I_IS`, which is repeated in Figure 11 (a), and the path machine for variable `!?USE` in Figure 11 (b). Because the select value is not eliminated from the `I_IS` path machine, the structure of the `!?USE` path machine describes cases where the `process1()` call is sometimes unreachable for a value used in a `process0()` call. If the select value was eliminated from the `I_IS` path machine, then the `process1()` call would always be indicated as reachable in the `!?USE` path machine as long as `hasNext` was true at least once. However, when the select value is eliminated from the `!?USE` path machine, as in Figure 11 (c), the `process1()` call is sometimes reachable and sometimes unreachable. As a result, the path machine identifies iterations in the loop where lock `LK` can be safely released, which would not be possible if the select value was eliminated from the `I_IS` path machine before the `!?USE` path machine was computed.

The select value elimination process first identifies the regions in the path machine where the select value transitition is true and regions where it is false. For the `!?USE` path machine in Figure 11 (b), a beginning false region for the ***M*** select value (representing `next` call results) starts at state ***BGN*** and ends at state ***M***$_0$. The single true region ranges from state ***M***$_0$ until state ***M***$_2$, where another false region exists forever. Behavior that occurs in the false regions that surround a true region are unioned with the behavior of that true region. For example, in the later false region of Figure 11 (b), the transition from state ***Y***$_2$ is split in Figure 11 (c) because `!?USE` will always remain true through the next ***X*** state when `process0()` is true.

4. Discussion

In its current form, path-precise analysis is intra-procedural because path machines cannot express the unbounded stack behavior needed to reason about recursion. Enhancing path machines with a stack abstraction would increase their expressiveness to the power of push-down automata. Such path machines would be much more difficult to compute via an analysis. Using the results of these path machines in property synthesis would also be problematic, because a stack would need to be maintained and inspected at run-time to ensure that code executed in the correct context.

Our strategy for making PPA inter-procedural involves abstracting away the call stack and recursion. Annotations provided by a meta-program or inferred via analysis can mitigate the precision lost through abstraction. Annotations can describe a procedure's behavior as a set of boolean assignments, which can replace calls to the procedure in an RBP. The same strategy can be generalized to reason about other kinds of unbounded memory constructs, such as arrays, using techniques like shape analysis [16].

Although we separate our discussion of the annotation and abstraction property synthesis steps in Section 2, they are tightly inter-related. Annotations are added as boolean variables and assignments, which will not be abstracted away during the abstraction step. Any precision lost during property synthesis occurs when expressions are converted into unknown values. Annotations can be used to direct the abstraction process by replacing what would otherwise be a single unknown value with more a detailed set of variables, tests, and assignments. For example, $y > x$ and $y \leq x$ evaluate to mutually exclusive conditions that can be expressed with one unknown value rather than two. Libraries can also express domain-specific relationships that can be used to enhance the precision of an abstraction. In addition to enabling the encoding of property-specific behavior in a generic way, the annotation language can enable encodings that improve precision of the abstraction process.

5. Related Work

Property synthesis follows work in property checking. The multiple steps used in property synthesis are similar to those used in SLAM [3], a verification method based on iterative refinement. In SLAM, a program is annotated, the annotated program is abstracted into a boolean program, and the boolean program is analyzed to determine whether or not a temporal safety property is adhered to in the program. Property synthesis uses these same steps even though the technologies used are different. An additional step in SLAM, which iteratively refines the precision of a boolean program to guide results, is not applicable in property synthesis. Property synthesis abstracts programs into regular boolean programs that are similar to boolean programs created by the tool C2BP [1]. Although regular boolean programs are less expressive than boolean programs because they lack a stack abstraction, they are more appropriate for program synthesis because they form a simple state-transition system that can be inspected during program execution.

Property synthesis is supported by a path-precise analysis that is most similar to Bebop [2], which model checks boolean programs in SLAM. Bebop computes over sets of boolean variable behavior for each statement in a boolean program, which is similar to how path-precise analysis computes path machines. Bebop results are used to reason about the reachability of error states in a program, which is similar to how path machines are used to reason about about the invariants under which property code should execute. Unlike Bebop results, path machines are designed specifically to enable the synthesis of code: path machines can identify every path in a regular boolean program that can reach property code, and can be used to synthesize code that recognizes these paths at run-time in the corresponding program.

Property simulation is a path-sensitive analysis designed to support partial property verification in ESP [8]. Property simulation improves precision over traditional DFA by heuristically tracking branches in a program when they obviously affect the behavior being analyzed. Metal's xgcc [11] finds bugs using a path-sensitive algorithm based on a heuristic that most path are executable and data dependencies are simple. Where ESP and Metal are designed to be scalable, property synthesis requires precision and only processes one procedure at a time. Unlike the analyses used in both ESP or Metal, path-precise analysis can correlate how loop-carried dependencies affect control-flow.

Path-sensitive analyses often work by sharpening data-flow analysis results with a finite set of path predicates [12]. GSA [18] is a variation of SSA that enhances precision by qualifying merge nodes with path predicates. Predicated array data-flow analysis [15] uses path predicates to enhance both compile-time analysis and to introduce run-time tests that guard safe execution of optimized code, which is similar to how path machines are used to derive run-time tests that guard execution of property code. Path-precise analysis innovates on these path-sensitive analyses by describing paths directly in a transition system. As a result, path-precise analysis can disambiguate between paths as they loop, whereas simple path predicates cannot.

An alternative to dealing with loops is to expand the loop k times to recognize loop-carried dependencies that occur across k iterations [14]. However, using the results of this analysis can lead to an exponential blow-up in the program. Additionally, loop-carried dependency distances may not be constant. Path-precise analysis avoids both of these problems with path machines. Although the state structure of a path machine may be expanded to describe periodic loop behavior, this expansion does not carry through to the CFG when the path machine is used to synthesize code. Unlike expansion, path machines can also be used to identify loop-carried dependencies over an arbitrary number of iterations; e.g., path machines can identify a loop-carried dependency that occurs until the loop terminates.

6. Conclusions and Future Work

In this paper we have described property synthesis, a mechanism for automatically inserting code that achieves some property into a procedure. The property, such as lock acquisition, is described as a program analysis problem, such as determining where and when a value is used in a procedure. The correct insertion of property code into a program occurs automatically when an answer to this analysis problem is provided for the program.

We have concentrated on describing the analysis necessary to support property synthesis. Because property code depends on tracking value use, a precise analysis is necessary. We have described PPA, path-precise analysis, which produces a set of path machines. The path machines form a finite-state model of the procedure being analyzed; i.e., it is based on a boolean abstraction of the procedure. Path machines are produced with cycles of state transitions, which enables the precise analysis of loop behavior. Beyond property synthesis, computing analysis results as state machines could also benefit other program analysis domains, such as partial verification of program behavior and optimizing programs.

We are implementing this analysis in the context of a meta-programming system that supports property synthesis. We have implemented early versions of this analysis for Java programs, but we have not yet implemented the version of the analysis described in this paper. The meta-programming system provides a language to describe properties. When this language is combined with the analysis in this paper, the system will enable property code to be described at a high level and inserted automatically.

Besides lock acquisition, property synthesis can be applied to other properties such as error handling and consistency protocols. Property synthesis should be useful for a range of metaprogramming tasks. We are currently exploring how property synthesis can be used to generate and customize software components. Also, we are exploring how property synthesis can accommodate architectural properties that span multiple procedures, such as resource management policies.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 9876117. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] T. Ball, R. Majmdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of PLDI*, June 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. of SPIN Workshop*, pages 113–130, May 2000.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN Workshop*, May 2001.

[4] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proc. of PASTE*, June 2001.

[5] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *Proc. of POPL*, pages 237–251, Jan. 1998.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software: Practice and Experience, 30(7)*, pages 775–802, 2000.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficient computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems, 13(4)*, pages 84–97, Oct. 1991.

[8] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of PLDI*, June 2002.

[9] D. L. Detlefs. An overview of the extended static checking system. In *Proc. of Workshop on Formal Methods in Software Practice*, pages 1–9, 1996.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of OSDI*, Oct. 2000.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. of PLDI*, June 2002.

[12] L. Holley and B. Rosen. Qualified dataflow problems. In *Proc. of POPL*, Jan. 1980.

[13] G. Kiczales, E. Hilsdale, J. Hungunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of ECOOP*, June 2001.

[14] J. Knoop, O. Ruthing, and B. Steffen. Expansion-based removal of semantic partial redundancies. In *Proc. of CC*, page 91, 1999.

[15] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proc. of ICS*, pages 204–211, July 1998.

[16] M. Sagiv, T. Reps, and R. Wilhelm. Parameteric shape analysis via 3-valued logic. In *Proc. of POPL*, pages 303–321, Jan. 1999.

[17] B. Steffen. Property-oriented expansion. In *LNCS 1145, Symposium on Static Analysis*, page 22, 1996.

[18] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of ICS*, June 1995.