# Automatically Validating Temporal Safety Properties of Interfaces

Thomas Ball and Sriram K. Rajamani

Software Productivity Tools
Microsoft Research
http://www.research.microsoft.com/slam/

**Abstract.** We present a process for validating temporal safety properties of software that uses a well-defined interface. The process requires only that the user state the property of interest. It then automatically creates abstractions of C code using iterative refinement, based on the given property. The process is realized in the SLAM toolkit, which consists of a model checker, predicate abstraction tool and predicate discovery tool. We have applied the SLAM toolkit to a number of Windows NT device drivers to validate critical safety properties such as correct locking behavior. We have found that the process converges on a set of predicates powerful enough to validate properties in just a few iterations.

## 1 Introduction

Large-scale software has many components built by many programmers. Integration testing of these components is impossible or ineffective at best. Property checking of interface usage provides a way to partially validate such software. In this approach, an interface is augmented with a set of properties that all clients of the interface should respect. An automatic analysis of the client code then validates that it meets the properties, or provides examples of execution paths that violate the properties. The benefit of such an analysis is that errors can be caught early in the coding process.

We are interested in checking that a program respects a set of *temporal safety* properties of the interfaces it uses. Safety properties are the class of properties that state that "something bad does not happen". An example is requiring that a lock is never released without first being acquired (see [24] for a formal definition). Given a program and a safety property, we wish to either validate that the code respects the property, or find an execution path that shows how the code violates the property.

In this paper, we show that safety properties of system software can be validated/invalidated using model checking, without the need for user-supplied annotations (invariants) or user-supplied abstractions. The user only needs to state the safety properties of interest (in our specification language SLIC, described later). As no annotations are required, we use model checking to compute fixpoints automatically over an abstraction of the C code. We construct an appropriate abstraction by (1) obtaining an initial abstraction from the property that needs to be checked, and (2) refining this abstraction using an automatic refinement algorithm.

We model abstractions of C programs using *boolean programs* [3]. Boolean programs are C programs in which all variables have boolean type. Each boolean variable in a boolean program has an interpretation as a predicate over the infinite state space of the C program. Our experience shows that our refinement algorithm finds boolean program abstractions that are precise enough to validate properties. Furthermore, if the property is violated, the process of searching for a suitable boolean program abstraction leads to a manifestation of the violation.

We present the SLAM toolkit for checking safety properties of system software, and report on our experience in using the toolkit to check properties of Windows NT device drivers. Given a safety property to check on a C program, the SLAM process has the following phases: (1) abstraction, (2) model checking, and (3) predicate discovery. We have developed tools to support each of these phases:

– C2BP, a tool that transforms a C program $P$ into a boolean program $\mathcal{BP}(P, E)$ with respect to a set of predicates $E$ over the state space of $P$ [1, 2];
– BEBOP, a tool for model checking boolean programs [3], and
– NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program.

The SLAM toolkit provides a fully automatic way of checking temporal safety properties of system software. Violations are reported by the SLAM toolkit as paths over the program $P$. It never reports spurious error paths. Instead, it detects spurious error paths and uses them to automatically refine the abstraction (to eliminate these paths from consideration). Since property checking is undecidable, the SLAM refinement algorithm may not converge. However, in our experience, it usually converges in a few iterations. Furthermore, whenever it converges, it gives a definite "yes" or "no" answer.

The worst-case run-time complexity of the SLAM tools BEBOP and C2BP is linear in the size of the program's control flow graph, and exponential in the number of predicates used in the abstraction. We have implemented several optimizations to make BEBOP and C2BP scale gracefully in practice, even with a large number of predicates. The NEWTON tool scales linearly with path length and number of predicates.

We applied the SLAM toolkit to check the use of the Windows NT I/O manager interface by device driver clients. There are on the order of a hundred rules that the clients of the I/O manager interface should satisfy. We have automatically checked properties on device drivers taken from the Microsoft Driver Development Kit[1]. While checking for correct use of locks, we found that the SLAM process converges in one or two iterations to a boolean program that is

[1] The code of the device drivers we analyzed is freely available from `http://www.microsoft.com/ddk/W2kDDK.htm`

sufficiently precise to validate/invalidate the property. We also checked a data-dependent property, which requires keeping track of value-flow and aliasing, using four iterations of the SLAM tools.

The remainder of this paper is organized as follows. Section 2 gives an overview of the SLAM approach by applying the tools to verify part of an NT device driver. Sections 3, 4 and 5 give brief descriptions of the three tools that compose the SLAM toolkit and explain how they work in the context of the running example. Section 6 describes our experience applying the tools to various NT device drivers. Section 7 discusses related work and Section 8 concludes the paper.

## 2 Overview

This section introduces the SLAM refinement algorithm and then applies this algorithm to a small code example, extracted from a PCI device driver. The SLAM toolkit handles a significant subset of the C language, including pointers, structures, and procedures (with recursion and mutual recursion). A limitation of our tools is that they assume a *logical* model of memory when analyzing C programs. Under this model, the expression $p + i$, where $p$ is a pointer and $i$ is an integer, yields a pointer value that points to the same object pointed to by $p$. That is, we treat pointers as references rather than as memory addresses. Note that this is the same basic assumption underlying most points-to analysis, including the one that our tools use [11].

### 2.1 Property Specification

We have created a low-level specification language called SLIC (Specification Language for Interface Checking) in which the user states safety properties. A SLIC specification describes a state machine and has two components: (1) a static set of *state variables*, described as a C structure, and (2) a set of *events* and state transitions on the events. The state variables can be of any C type, including integers and pointers.

Figure 1(a) shows a SLIC specification that specifies proper usage of spin locks. There is one state variable `locked` that is initialized to `0`. There are two events on which state transitions happen —returns of calls to the functions `KeAcquireSpinLock` and `KeReleaseSpinLock`. Erroneous sequences of calls to these functions results in the execution of the `abort` statement.

We wish to check if a temporal safety property $\varphi$ specified using SLIC is satisfied by a program $P$. We have built a SLIC instrumentation tool that automatically instruments the given program $P$ with property $\varphi$ to result in a program $P'$ such that $P$ satisfies $\varphi$ iff the label `SLIC_ERROR` is not reachable in $P'$. In particular, the tool first creates C code from the SLIC specification, as shown in Figure 1(b). The label `SLIC_ERROR` in the procedure `slic_abort` reflects the finite state machine executing an `abort` statement and moving into a reject state. The tool then inserts calls to the appropriate SLIC C functions

```
state {
   enum { Unlocked=0, Locked=1 }
     state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

(a)

```
enum { Unlocked=0, Locked=1 }
  state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ;
}

void KeAcquireSpinLock_return() {
  if (state == Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked;
}
```

(b)

**Fig. 1.** (a) A SLIC specification for proper usage of spin locks, and (b) its compilation into C code.

in the program $P$ to result in the instrumented program $P'$. This is known in the model checking community as a "product automaton construction" and is a fairly standard way to encode safety properties. Due to want of space, the formal syntax and semantics of SLIC, and details of the automatic instrumentation tool will be the topic of a future paper.

### 2.2 Refinement Algorithm

We wish to check if the instrumented program $P'$ can ever reach the label `SLIC_ERROR`. Let $i$ be a metavariable that records the SLAM iteration count. In the first iteration ($i = 0$), we start with a set of predicates $E_0$ that are present in the conditionals of the SLIC specification. Let $E_i$ be some set of predicates over the state of $P'$. Then iteration $i$ of SLAM is carried out using the following steps:

1. Apply C2BP to construct the boolean program $\mathcal{BP}(P', E_i)$.
2. Apply BEBOP to check if there is a path $p_i$ in $\mathcal{BP}(P', E_i)$ that reaches the `SLIC_ERROR` label. If BEBOP determines that `SLIC_ERROR` is not reachable, then the property $\varphi$ is valid in $P$, and the algorithm terminates.
3. If there is such a path $p$, then we use NEWTON to check if $p$ is feasible in $P$. There are two outcomes:

```
void example() {
 do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;
  req = devExt->WLHV;
  if(req && req->status){
    devExt->WLHV = req->Next;
    KeReleaseSpinLock();

    irp = req->irp;
    if(req->status > 0){
      irp->IoS.Status = SUCCESS;
      irp->IoS.Info = req->Status;
    } else {
      irp->IoS.Status = FAIL;
      irp->IoS.Info = req->Status;
    }
    SmartDevFreeBlock(req);
    IoCompleteRequest(irp);
    nPackets++;
  }
 } while(nPackets!=nPacketsOld);
 KeReleaseSpinLock();

}
```

(a) Program $P$

```
void example() {
   do {
    KeAcquireSpinLock();
A:  KeAcquireSpinLock_return();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if(req && req->status){
      devExt->WLHV = req->Next;
      KeReleaseSpinLock();
B:    KeReleaseSpinLock_return();
      irp = req->irp;
      if(req->status > 0){
        irp->IoS.Status = SUCCESS;
        irp->IoS.Info = req->Status;
      } else {
        irp->IoS.Status = FAIL;
        irp->IoS.Info = req->Status;
      }
      SmartDevFreeBlock(req);
      IoCompleteRequest(irp);
      nPackets++;
    }
   } while(nPackets!=nPacketsOld);
   KeReleaseSpinLock();
C: KeReleaseSpinLock_return();
 }
```

(b) Instrumented Program $P'$

**Fig. 2.** (a) A snippet of device driver code $P$, and (b) instrumented code $P'$ that checks proper use of spin locks.

  - "yes": the property $\varphi$ has been invalidated in $P$, and the algorithm terminates with an error path $p_i$ (a witness to the violation of $\varphi$).
  - "no": NEWTON finds a set of predicates $F_i$ that explain the infeasibility of path $p_i$ in $P$.
4. Let $E_{i+1} := E_i \cup F_i$, and $i := i + 1$, and proceed to the next iteration.

As stated before, this algorithm is potentially non-terminating. However, when it does terminate, it provides a definitive answer.

### 2.3 Example

Figure 2(a) presents a snippet of (simplified) C code from a PCI device driver that processes interrupt request packets (IRPs). Of interest here are the calls the code makes to acquire and release spin locks (`KeAcquireSpinLock` and `KeReleaseSpinLock`). Figure 2(b) shows the program automatically instrumented

```
decl {state==Locked}, {state==Unlocked};

void slic_abort() begin SLIC_ERROR: skip; end

void KeAcquireSpinLock_return()
begin
  if ({state==Locked})
    slic_abort();
  else
    {state==Locked},{state==Unlocked} := T,F;
end

void KeReleaseSpinLock_return()
begin
  if ({state == Unlocked})
    slic_abort();
  else
    {state==Locked},{state==Unlocked} := F,T;
end
```

**Fig. 3.** The C code of the SLIC specification from Figure 1(b) compiled by C2BP into a boolean program.

by the SLIC tool with respect to the property specification in Figure 1(a). Note that calls to the appropriate SLIC C functions (Figure 1(b)) are introduced (at labels `A`, `B`, and `C`).

The question we wish to answer is: is the label `SLIC_ERROR` reachable in the program $P'$ comprised of the code from Figure 1(b) and Figure 2(b)? The following sections apply the algorithm given above to show that `SLIC_ERROR` is unreachable in this program.

### 2.4 Initial Boolean Program

The first step of the algorithm is to generate a boolean program from the C program and the set of predicates $E_0$ that define the states of the finite state machine. We represent our abstractions as *boolean programs*. The syntax and semantics of boolean programs was defined in [3]. Boolean programs are essentially C programs in which the only allowed types are `bool`, with values T (true) and F (false), and `void`. Boolean programs also allow control non-determinism, through the conditional expression "$*$", as shown later on.

For our example, the set $E_0$ consists of two *global* predicates ($state = Locked$) and ($state = Unlocked$) that appear in the conditionals of the SLIC specification. These two predicates and the program $P'$ are input to the C2BP (C to Boolean Program) tool. The translation of the SLIC C code from Figure 1(b) to the boolean program is shown in Figure 3. The translation of the `example` procedure

```
    void example()
    begin
     do
      skip;
A:    KeAcquireSpinLock_return();
      skip;
      skip;
      if (*) then
        skip;
        skip;
B:      KeReleaseSpinLock_return();
        skip;
        if (*) then
          skip;
          skip;
        else
          skip;
          skip;
        fi
        skip;
        skip;
        skip;
      fi
     while (*);
     skip;
C:   KeReleaseSpinLock_return();
    end
```

(a) Boolean program $\mathcal{BP}(P', E_0)$

```
    void example()
    begin
     do
      skip;
A:    KeAcquireSpinLock_return();
      b := T;
      skip;
      if (*) then
        skip;
        skip;
B:      KeReleaseSpinLock_return();
        skip;
        if (*) then
          skip;
          skip;
        else
          skip;
          skip;
        fi
        skip;
        skip;
        b := choose(F,b);
      fi
     while (!b);
     skip;
C:   KeReleaseSpinLock_return();
    end
```

(b) Boolean program $\mathcal{BP}(P', E_1)$

**Fig. 4.** The two boolean programs created while checking the code from Figure 2(b). See text for the definition of the `choose` function.

is shown in Figure 4(a). Together, these two pieces of code comprise the boolean program $\mathcal{BP}(P', E_0)$ output by C2BP.

As shown in Figure 3, the translation of the SLIC C code results in the global variables, {`state==Locked`} and {`state==Unlocked`}.[2] For every statement $s$ in the C program and predicate $e \in E_0$, the C2BP tool determines the effect of statement $s$ on predicate $e$. For example, consider the assignment statement "`state = Locked;` " in Figure 1(b). This statement makes the predicate $(state = Locked)$ true and the predicate $(state = Unlocked)$ false. This is reflected in the boolean program by the parallel assignment statement

[2] Boolean programs permit a variable identifier to be an arbitrary string enclosed between "{" and "}". This is helpful for giving boolean variables names to directly represent the predicates in the C program that they represent.

```
{state==Locked}, {state==Unlocked} := T,F;
```

in Figure 3. The translation of the boolean expressions in the conditional statements of the C program results in the obvious corresponding boolean expressions in the boolean program. Control non-determinism is used to conservatively model the conditions in the C program that cannot be abstracted precisely using the predicates in $E_0$, as shown in Figure 4(a).

Many of the assignment statements in the `example` procedure are abstracted to the **skip** statement (no-op) in the boolean program. The C2BP tool uses Das's points-to analysis [11] to determine whether or not an assignment statement through a pointer dereference can affect a predicate $e$. In our example, the points-to analysis shows that no variable in the C program can alias the address of the global `state` variable.[3]

We say that the boolean program $\mathcal{BP}(P', E_0)$ *abstracts* the program $P'$, since every feasible execution path $p$ of the program $P'$ also is a feasible execution path of $\mathcal{BP}(P', E_0)$.

### 2.5 Model Checking The Boolean Program

The second step of our process is to determine whether or not the label `SLIC_ERROR` is reachable in the boolean program $\mathcal{BP}(P', E_0)$. We use the BEBOP model checker to determine the answer to this query. In this case, the answer is "yes". Like most model checkers, the BEBOP tool produces a (shortest) path leading to the error state. In this case, the shortest path to the error state is the path that goes around the loop twice, acquiring the lock twice without an intermediate release, as given by the error path $p_0$ of labels [`A, A, SLIC_ERROR`].

### 2.6 Predicate Discovery over Error Path

Because the C program and the boolean program abstractions have identical control-flow graphs, the error path $p_0$ in $\mathcal{BP}(P', E_0)$ produced by BEBOP is also a path of program $P$. The question then is: does $p_0$ represent a feasible execution path of $P$? That is, is there some execution of program $P$ that follows the path $p_0$? If so, we have found a real error in $P$. If not, path $p_0$ is a spurious error path.

The NEWTON tool takes a C program and a (potential) error path as an input. It then uses verification condition generation (VCGen) to determine if the path is feasible. The answer may be "yes" or "no". [4]

[3] We had to write stubs for the procedures `SmartDevFreeBlock`, and kernel procedures `IoCompleteRequest`, `KeAcquireSpinLock`, and `KeReleaseSpinLock`. The analysis determines that these procedures cannot affect state variables so the calls to them are removed.

[4] Since underlying decision procedures in the theorem prover and our axiomatization of C are incomplete, "don't know" is also a possible answer. In practice, the theorem provers we use [27, 13, 4] have been able to give a "yes" or "no" answer in every example we have seen so far.

If the answer is "yes", then an error path has been found, and we report it to the user. If the answer is "no" then NEWTON uses a new algorithm to identify a small set of predicates that "explain" why the path is infeasible.

In the running example, NEWTON detects that the path $p$ is infeasible, and returns a single predicate ($nPackets = npacketsOld$) as the explanation for the infeasibility. This is because the predicate ($nPackets = nPacketsOld$) is required to be both true and false by path $p$. The assignment of `nPacketsOld` to `nPackets` makes the predicate true, and the loop test requires it to be false at the end of the **do-while** loop for the loop to iterate, as specified by the path $p$.

### 2.7 The Second Boolean Program

In the second iteration of the process, the predicate ($nPackets = nPacketsOld$) is added to the set of predicates $E_0$ to result in a new set of predicates $E_1$. Figure 4(b) shows the boolean program $\mathcal{BP}(P', E_1)$ that C2BP produces. This program has one additional boolean variable (`b`) that represents the predicate ($nPackets = nPacketsOld$). The assignment statement `nPackets = nPacketsOld;` makes this condition true, so in the boolean program the assignment `b := T;` represents this assignment. Using a theorem prover, C2BP determines that if the predicate is true before the statement `nPackets++`, then it is false afterwards. This is captured by the assignment statement in the boolean program `b := choose(F,b);`. The `choose` function is defined as follows:

```
bool choose(pos, neg)
begin
     if (pos) then return T; elsif (neg) then return F;
  elsif (*)   then return T; else             return F; fi
end
```

The `pos` parameter represents positive information about a predicate while the `neg` parameter represents negative information about a predicate. The `choose` function is never called with both parameters evaluating to true. If both parameters are false then there is not enough information to determine whether the predicate is definitely true or definitely false, so F or T is returned, non-deterministically.

Applying BEBOP to the new boolean program shows that the label `SLIC_ERROR` is not reachable. In performing its fixpoint computation, BEBOP discovers that the following loop invariant holds at the end of the **do-while** loop:

$$\begin{aligned}&(state = Locked \wedge nPackets = nPacketsOld)\\ \vee\ &(state = Unlocked \wedge nPackets \neq nPacketsOld)\end{aligned}$$

That is, either the lock is held and the loop will terminate (and thus the lock needs to be released after the loop), or the lock is free and the loop will iterate. The combination of the predicate abstraction of C2BP and the fixpoint computation of BEBOP has found this loop-invariant over the predicates in $E_1$.

This loop-invariant is strong enough to show that the label `SLIC_ERROR` is not reachable.

## 3 C2BP: A Predicate Abstractor For C

C2BP takes a C program $P$ and a set $E = \{e_1, e_2, \ldots, e_n\}$ of predicates on the variables of $P$, and automatically constructs a boolean program $\mathcal{BP}(P, E)$[1, 2]. The set of predicates $E$ are pure C boolean expressions with no function calls. The boolean program $\mathcal{BP}(P, E)$ contains $n$ boolean variables $V = \{b_1, b_2, \ldots, b_n\}$, where each boolean variable $b_i$ represents a predicate $e_i$. Each variable in $V$ has a *three-valued* domain: **false**, **true**, and $*$.[5] The program $\mathcal{BP}(P, E)$ is a *sound* abstraction of $P$ because every possible execution trace $t$ of $P$ has a corresponding execution trace $t'$ in $B$. Furthermore, $\mathcal{BP}(P, E_0)$ is a precise abstraction of $P$ with respect to the set of predicates $E_0$, in a sense stated and shown elsewhere [2]. Since $\mathcal{BP}(P, E)$ is an abstraction of $P$, it is guaranteed that an invariant $I$ discovered (by BEBOP) in $\mathcal{BP}(P, E)$, as boolean combinations of the $b_i$ variables, is also an invariant in the C code, where each $b_i$ is replaced by its corresponding predicate $e_i$.

C2BP determines, for every statement $s$ in $P$ and every predicate $e_i \in E$, how the execution of $s$ can affect the truth value of $e_i$. This is captured in the boolean program by a statement $s_B$ that conservatively updates each $b_i$ to reflect the change. C2BP computes $s_B$ by (1) first computing the weakest precondition of $e_i$, and its negation with respect to $s$, and (2) strengthening the weakest precondition in terms of predicates from $E$, using a theorem prover.

We highlight the technical issues in building a tool like C2BP:

- **Pointers**: We use an alias analysis of the C program to determine whether or not an update through a pointer dereference can potentially affect an expression. This greatly increases the precision of the C2BP tool. Without alias analysis, we would have to make very conservative assumptions about aliasing, which would lead to invalidating many predicates.
- **Procedure calls**: Since boolean programs support procedure calls, we are able to abstract procedures modularly. The abstraction process for procedure calls is challenging, particularly in the presence of pointers. After a call, the caller must conservatively update local state that may have been modified by the callee. We provide a sound and precise approach to abstracting procedure calls that takes such side-effects into account.
- **Precision-efficiency tradeoff**. C2BP uses a theorem prover to strengthen weakest pre-conditions in terms of the given predicate set $E$. Doing this strengthening precisely requires $O(2^{|E|})$ calls to the theorem prover. We have explored a number of optimization techniques to reduce the number of calls made to the theorem prover. Some of these techniques result in an equivalent boolean program, while others trade off precision for computation

[5] The use of the third value $*$, is encoded using control-nondeterminism as shown in the `choose` function of Section 2. That is, "$*$" is equivalent to "`choose(F,F)`".

speed. We currently use two automatic theorem provers Simplify [27, 13] and Vampyre [4]. We are also investigating using other decision procedures, such as those embodied in the Omega test [30] and PVS [28].

**Complexity.** The runtime of C2BP is dominated by calls to the theorem prover. In the worst-case, the number of calls made to the theorem prover for computing $\mathcal{BP}(P, E)$ is linear in the size of $P$ and exponential in the size of $E$. We can compute sound but imprecise abstractions by considering only $k$-tuples of predicates in the strengthening step. In all examples we have seen so far we find that we lose no precision for $k = 3$. Thus, in practice the complexity is cubic in the size of $E$.

## 4 BEBOP: A Model Checker for Boolean Programs

The Bebop tool [3] computes the set of reachable states for each statement of a boolean program using an interprocedural dataflow analysis algorithm in the spirit of Sharir/Pnueli and Reps/Horwitz/Sagiv [34, 31]. A state of a boolean program at a statement $s$ is simply a valuation to the boolean variables that are in scope at statement $s$ (in other words, a bit vector, with one bit for each variable in scope). The set of reachable states (or invariant) of a boolean program at $s$ is thus a set of bit vectors (equivalently, a boolean function over the set of variables in scope at $s$).

Bebop differs from typical implementations of dataflow algorithms in two crucial ways. First, it computes over sets of bit vectors at each statement rather than single bit vectors. This is necessary to capture correlations between variables. Second, it uses binary decision diagrams [5] (BDDs) to implicitly represent the set of reachable states of a program, as well as the transfer functions for each statement in a boolean program. Bebop also differs from previous model checking algorithms for finite state machines, in that it does not inline procedure calls, and exploits locality of variable scopes for better scaling. Unlike most model checkers for finite state machines, Bebop handles recursive and mutually recursive procedures. Bebop uses an explicit control-flow graph representation, as in a compiler, rather than encoding the control-flow with BDDs, as done in most symbolic model checkers. It computes a fixpoint by iterating over the set of facts associated with each statement, which are represented with BDDs.

**Complexity.** The worst-case complexity of Bebop is linear in the size of the program control-flow graph, and exponential in the maximum number of boolean variables in scope at any program point. We have implemented a number of optimizations to reduce the number of variables needed in support of BDDs. For example, we use live variable analysis to find program points where a variable becomes dead and then eliminate the variable from the BDD representation. We also use a global MOD/REF analysis of the boolean program in order to perform similar variable eliminations.

```
Input: A sequence of statements p = s_1, s_2, ..., s_m.
store := null map;
history := null set;
conditions := null set;
/* start of Phase 1 */
for i = 1 to m do {
  switch( s_i ) {
     "e_1 := e_2" :
        let lval = LEval (store, e_1) and
        let rval = REval(store, e_2) in {
              if(store[lval] is defined)
                 history := history ∪ {(lval, store[lval])}
              store[lval] := rval
        }
     "assume(e)" :
        let rval = REval(store, e) in {
           conditions := conditions ∪ {rval}
           if(conditions is inconsistent){
              /*Phase 2 */
              Minimize size of conditions while maintaining inconsistency
              /*Phase 3 */
              predicates := all dependencies of conditions using store and history
              Say "Path p is infeasible"
              return(predicates)
           }
        }
  }
}
Say "Path p is feasible"
return
```

**Fig. 5.** The high-level algorithm used by NEWTON

## 5 NEWTON: A Predicate Discoverer

NEWTON takes a C program $P$ and an error path $p$ as inputs. For the purposes of describing NEWTON, we can identify the path $p$ as a sequence of assignments and assume statements (every conditional is translated into an assume statement). The `assume` statement is the dual of `assert`: `assume`($e$) never fails. Executions on which $e$ does not hold at the point of the `assume` are simply ignored [15].

The internal state of NEWTON has three components: (1) *store*, which is a mapping from locations to symbolic expressions, (2) *conditions*, which is a set of predicates, and (3) a *history* which is a set of past associations between locations and symbolic expressions. The high-level description of NEWTON is given in Figure 5. The functions LEval and REval evaluate the l-value and r-value of a given expression respectively. NEWTON maintains the dependencies of each symbolic expression on the elements in *store*, to be used in Phase 3. It

```
s1:   nPacketsOld = nPackets;
s2:   req = devExt->WLHV;
s3:   assume(!req);
s4:   assume(nPackets != nPacketsOld);
```

| loc. | value | deps. | conds. | deps. |
|---|---|---|---|---|
| 1. *nPackets*: | $\alpha$ | () | | |
| 2. *nPacketsOld*: | $\alpha$ | (1) | | |

after `s1`

| loc. | value | deps. | conds. | deps. |
|---|---|---|---|---|
| 1. *nPackets*: | $\alpha$ | () | | |
| 2. *nPacketsOld*: | $\alpha$ | (1) | | |
| 3. *devExt*: | $\beta$ | () | | |
| 4. $\beta \rightarrow$ *WLHV*: | $\gamma$ | (3) | | |
| 5. *req*: | $\gamma$ | (3,4) | | |

after `s2`

| loc. | value | deps. | conds. | deps. |
|---|---|---|---|---|
| 1. *nPackets*: | $\alpha$ | () | $!(\gamma)$ | (5) |
| 2. *nPacketsOld*: | $\alpha$ | (1) | | |
| 3. *devExt*: | $\beta$ | () | | |
| 4. $\beta \rightarrow$ *WLHV*: | $\gamma$ | (3) | | |
| 5. *req*: | $\gamma$ | (3,4) | | |

after `s3`

| loc. | value | deps. | conds | deps. |
|---|---|---|---|---|
| 1. *nPackets*: | $\alpha$ | () | $!(\gamma)$ | (5) |
| 2. *nPacketsOld*: | $\alpha$ | (1) | $(\alpha != \alpha)$ | (1,2) |
| 3. *devExt*: | $\beta$ | () | | |
| 4. $\beta \rightarrow$ *WLHV*: | $\gamma$ | (3) | | |
| 5. *req*: | $\gamma$ | (3,4) | | |

after `s4`

**Fig. 6.** A path of four statements and four tables showing the state of Newton after simulating each of the four statements.

also uses symbolic constants for unknown values. We illustrate these using an example. Consider a path with the following four statements:

```
s1:   nPacketsOld = nPackets;
s2:   req = devExt->WLHV;
s3:   assume(!req);
s4:   assume(nPackets != nPacketsOld);
```

This path is a projection of the error path from Bebop in Section 2.

Figure 6 shows four states of Newton, one after processing each statement in the path. The assignment `nPacketsOld = nPackets` is processed by first introducing a symbolic constant $\alpha$ for the variable `nPackets`, and then assigning it to `nPacketsOld`. The assignment `req = devExt->WLHV` is processed by first introducing a symbolic constant $\beta$ for the value of `devExt`, then introducing a second symbolic constant $\gamma$ for the value of $\beta$`->WLHV`, and finally assigning $\gamma$ to `req`. The conditional `assume(!req)` is processed by adding the predicate `!(`$\gamma$`)` to the condition-set. The dependency list for this predicate is `(5)` since its evaluation depended on entry `5` in the store. Finally, the conditional `assume(nPackets != nPacketsOld)` is processed by adding the (inconsistent) predicate ($\alpha$ `!=` $\alpha$) to the condition-set, with a dependency list `(1,2)`. At this point, the theorem prover determines that the condition-set is inconsistent, and Newton proceeds to Phase 2.

```
VOID
SerialDebugLogEntry(IN ULONG Mask, IN ULONG Sig,
  IN ULONG_PTR Info1, IN ULONG_PTR Info2, IN ULONG_PTR Info3)
{
    KIRQL irql;
    irql = KeGetCurrentIrql();
    if (irql < DISPATCH_LEVEL) {
        KeAcquireSpinLock(&LogSpinLock, &irql);
    } else {
        KeAcquireSpinLockAtDpcLevel(&LogSpinLock);
    }
    // other code (deleted)
    if (irql < DISPATCH_LEVEL) {
        KeReleaseSpinLock(&LogSpinLock, irql);
    } else {
        KeReleaseSpinLockFromDpcLevel(&LogSpinLock);
    }
    return;
}
```

**Fig. 7.** Code snippet from serial-port driver.

Phase 2 removes the predicate `!(`$\gamma$`)` from the condition store, since the remaining predicate ($\alpha$`!=` $\alpha$) is inconsistent by itself. Phase 3 traverses store entries 1 and 2 from the dependency list. A post processing step then determines that the symbolic constant $\alpha$ can be unified with the variable `nPackets`, and NEWTON produces two predicates: ($nPacketsOld = nPackets$) and ($nPacketsOld \neq nPackets$). Since one is a negation of the other, only one of the two predicates needs to be added in order for the path to be ruled out in the boolean program. Though no symbolic constants are present in the final set of predicates in our example, there are other examples where the final list of predicates have symbolic constants. C2BP is able to handle predicates with symbolic constants. We do not discuss these details here due to want of space. The *history* is used when a location is overwritten with a new value. Since no location was written more than once in our example, we did not see the use of *history*. NEWTON also handles error paths where each element of the path is also provided with values to the boolean variables from BEBOP, and checks for their consistency with the concrete states along the path.

**Complexity.** The number of theorem-prover calls made by NEWTON on a path $p$ is $O(|\, p \,|)$, where $|\, p \,|$ is the length of the path.

## 6 NT Device Drivers: Case Study

This section describes our experience applying the SLAM toolkit to check properties of Windows NT device drivers. We checked two kinds of properties: (1)

Locking-unlocking sequences for locks should conform to allowable sequences (2) Dispatch functions should either complete a request, or make a request pending for later processing. In either case, a particular sequence of Windows NT specific actions should be taken.

The two properties have different characteristics from a property-checking perspective.

- The first property depends mainly on the program's control flow. We checked this property for a particular lock (called the "Cancel" spin lock) on three kernel mode drivers in the Windows NT device driver tool kit. We found two situations where spurious error paths led our process to iterate. With its inter-procedural analysis and detection of variable correlations, the SLAM tools were able to eliminate all the spurious error paths with at most one iteration of the process. In all the drivers, we started with 2 predicates from the property specification and added at most one predicate to rule out spurious error paths.
- The second property is data-dependent, requiring the tracking of value flow and alias relationships. We checked this property on a serial port device driver. It took 4 iterations through the SLAM tools and a total of 33 predicates to validate the property.

The drivers we analyzed were on the order of a thousand lines of C code each. In each of the drivers we checked for the first property, the SLAM tools ran in under a minute on an 800MHz Pentium PC with 512MB RAM. For the second property on the serial driver, the total run time for all the SLAM tools was about three minutes to complete all the four iterations.

We should note that we did not expect to find errors in these device drivers, as they are supposed to be examplars for others to use. Thus, the fact that the SLAM tools did not find errors in these program is not too surprising. We will report on the defect detection capabilities of the tools in a future paper.

### 6.1 Property 1

We checked for correct lock acquisition/release sequences on three kernel mode drivers: MCA-bus, serial-port and parallel-port. The SLAM tools validated the property on MCA-bus and parallel-port drivers without iteration. However, interprocedural analysis was required for checking the property, as calls to the acquire and release routines were spread across multiple procedures in the drivers. Furthermore, in the serial-port driver, the SLAM tools found one false error path in the first iteration, which resulted in the addition of a single predicate. Then the property was validated in the second iteration. The code-snippet that required the addition of the predicate is shown in Figure 7. The snippet shows that the code has a dependence on the interrupt request level variable (`irql`) that must be tracked in order to eliminate the false error paths. At most three predicates were required to check this property.

### 6.2 Property 2

A dispatch routine to a Windows NT device driver is a routine that the I/O manager calls when it wants the driver to perform a specific operation (e.g, read, write etc.) The dispatch routine is "registered" by the driver when it is initialized. A dispatch routine has the following prototype:

```
NTSTATUS DispatchX(IN PDEVICE_OBJECT DeviceObject, IN PIRP  irp)
```

The first parameter is a pointer to a so called "device object" that represents the device, and the second parameter is a pointer to a so called "I/O request packet", or "IRP" that contains information about the current request.

All dispatch routines must either process the IRP immediately (call this *option A*), or queue the IRP for processing later (call this *option B*). Every IRP must be processed under one of these two options. If the driver chooses option A, then it has to do the following actions in sequence:

1. Set the `irp->IoS.status` to some return code other than `STATUS_PENDING` (such as `STATUS_SUCCESS`, `STATUS_CANCELLED` etc.)
2. Call `IoCompleteRequest(irp)`
3. Return the same status code as in step 1.

If the driver chooses option B, then it has do the following actions in sequence:

1. Set `irp->IoS.status` to `STATUS_PENDING`
2. Call the kernel function `IoMarkIrpPending(irp)`
3. Queue the IRP into the driver's internal queue using the kernel function `IoStartPacket(irp)`
4. Return `STATUS_PENDING`

Note that this is a partial specification for a dispatch routine —just one of several properties that the dispatch routine must obey. Figure 8 shows a Slic specification for this property. The variable `$1` is used by Slic to denote the first parameter of the function, and the variable `$return` is used to denote the return value. Note that we first store the IRP at the entry of the dispatch routine in a state variable `gIrp` and then later check if the calls to `IoCompleteRequest` and `IoMarkIrpPending` refer to the same IRP.

**Checking the instrumented driver**. We started the first iteration of SLAM with 7 predicates from the Slic specification. It took 4 iterations of the SLAM tools and a total of 33 predicates to discover the right abstraction to validate this property. The discovered predicates kept track of the value of the flow of the `irp` pointer and the status value through several levels of function calls. We found one bug in the fourth iteration, which turned out to be due to an error in the Slic specification. After fixing it, the property passed.

## 7 Related Work

SLAM seeks a sweet spot between tools based on verification condition generation(VCGen) [14, 25, 26, 6] that operate directly on the concrete semantics, and

```
state {
  enum {Init, Complete, Pending}
   s = Init;
  PIRP gIrp = 0;
}

Dispatch.entry {
  s, gIrp = Init, $2;
}

IoCompleteRequest.call{
  if(gIrp == $1) {
    if( s != Init) abort;
    else s = Complete;
  }
}
```

```
IoMarkIrpPending.call{
  if(gIrp == $1) {
    if( s != Init) abort;
    else s = Pending;
  }
}

Dispatch.exit{
  if (s == Complete) {
    if( $return == STATUS_PENDING)
      abort;
  } else if (s == Pending) {
    if( $return != STATUS_PENDING)
      abort;
  }
}
```

**Fig. 8.** SLIC specification for completing an IRP or marking it as pending.

model checking or data flow-analysis based tools [8, 21, 18, 16] that operate on abstractions of the program. We use VCGen-based approach on finite (potentially interprocedural) paths of the program, and use the knowledge gained to construct abstract models of the program. NEWTON uses VCGen on the concrete program, but as it operates on a single finite interprocedural path at a time, it does not require loop-invariants, or pre-conditions and post-conditions for procedures. C2BP also reasons about the statements of the C program using decision procedures, but does so only locally, one statement at a time. Global analysis is done only on the boolean program abstractions, using the model checker BEBOP. Thus, our hope is to scale without losing precision, as long as the property of interest allows us to do so, by inherently requiring a small abstraction for its validation or invalidation.

SLAM generalizes Engler et al.'s approach [18] in three ways: (1) it is sound (modulo the assumptions about memory safety); (2) it permits interprocedural analysis; (3) it avoids spurious examples through iterative refinement (in some of the code Engler et al. report on, their techniques generated three times as many spurious error paths as true error paths, a miss rate of 75%.[6]) In fact, with a suitable definition of abstraction, and choice of initial predicates, the first iteration of the SLAM process is equivalent to performing Engler et al.'s approach interprocedurally.

[6] Jon Pincus, who led the development of an industrial-strength error detection tool for C called PREfix [6], observes that users of PREfix will tolerate a false alarm rate in the range 25-50% depending on the application [29].

Constructing abstract models of programs has been studied in several contexts. Abstractions constructed by [18] and [22] are based on specifying transitions in the abstract system using a pattern language, or as a table of rules. Automatic abstraction support has been built into the Bandera tool set [17]. They require the user to provides finite domain abstractions of data types. Predicate abstraction, as implemented in C2BP is more general, and can capture relationships between variables. The predicate abstraction in SLAM was inspired by the work of Graf and Saidi [20] in the model checking community. Efforts have been made to integrate predicate abstraction with theorem proving and model checking [32]. Though our use of predicate abstraction is related to these efforts, our goal is to analyze software written in common programming languages.

The SLAM tools C2BP and BEBOP can be used in combination to find loop-invariants expressible as boolean functions over a given set of predicates. The loop-invariant is computed by the model checker BEBOP using a fixpoint computation on the abstraction computed by C2BP. Prior work for generating loop invariants has used symbolic execution on the concrete semantics, augmented with widening heuristics [35, 36]. The Houdini tool guesses a candidate set of annotations (invariants,preconditions, postconditions) and uses the ESC/Java checker to refute inconsistent annotations until convergence [19].

Boolean programs can be viewed as abstract interpretations of the underlying program [9]. The connections between model checking, dataflow analysis and abstract interpretation have been explored before [33] [10]. The model checker BEBOP is based on earlier work on interprocedural dataflow analysis [34, 31]. Automatic iterative refinement based on error paths first appeared in [23], and more recently in [7]. Both efforts deal with finite state systems.

An alternative approach to static validation of safety properties, is to provide a rich type system that allows users to encode both safety properties and program annotations as types, and reduce property validation to type checking [12].

## 8 Conclusions

We have presented a fully automated methodology to validate/invalidate temporal safety properties of software interfaces. Our process does not require user supplied annotations, or user supplied abstractions. When our process converges, it always give a definitive "yes" or "no" answer.

The ideas behind the SLAM tools are novel. The use of boolean programs to represent program abstractions is new. To the best of our knowledge, C2BP is the first automatic predicate abstraction tool to handle a full-scale programming language, and perform a sound and precise abstraction. BEBOP is the first model checker to handle procedure calls using an interprocedural dataflow analysis algorithm, augmented with representation tricks from the symbolic model checking community. NEWTON uses a path simulation algorithm in a novel way, to generate predicates for refinement.

We have demonstrated that the SLAM tools converge in a few iterations on device drivers from the Microsoft DDK.

The SLAM toolkit has a number of limitations that we plan to address. The logical model of memory is a limitation, since it is not the actual model used by C programs. We plan to investigate using a physical model of memory. We also are exploring what theoretical guarantees we can give about the termination of our iterative refinement. Finally, we plan to evolve the SLAM tools by applying them to more code bases, both inside and outside Microsoft.

**Acknowledgements.** We thank Rupak Majumdar and Todd Millstein for their hard work in making the C2BP tool come to life. Thanks to Andreas Podelski for helping us describe the C2BP tool in terms of abstract interpretation. Thanks also to the members of the Software Productivity Tools research group at Microsoft Research for many enlightening discussions on program analysis, programming languages and device drivers, as well as their numerous contributions to the SLAM toolkit.

## References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation(to appear)*. ACM, 2001.
2. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems(to appear)*. Springer-Verlag, 2001.
3. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
4. D. Blei and et al. Vampyre: A proof generating theorem prover — http://www.eecs.berkeley.edu/ rupak/vampyre.
5. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
8. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000: International Conference on Software Engineering*, pages 439–448. ACM, 2000.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
10. P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL 00: Principles of Programming Languages*, pages 12–25. ACM, 2000.
11. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.
12. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation(to appear)*. ACM, 2001.
13. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover – http://research.compaq.com/src/esc/simplify.html.

14. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.
15. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
16. M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE 94: Foundations of Software Engineering*, pages 62–75. ACM, 1994.
17. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: Software Engineering (to appear)*, 2001.
18. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation.* Usenix Association, 2000.
19. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters (to appear)*, 2001.
20. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
21. G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
22. G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
23. R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.
24. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.
25. K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In *CC 98: Compiler Construction*, LNCS 1383, pages 302–305. Springer-Verlag, 1998.
26. G. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.
27. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
28. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV 96: Computer-Aided Verification*, LNCS 1102, pages 411–414. Springer-Verlag, 1996.
29. J. Pincus. personal communication, October 2000.
30. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
31. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
32. H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.
33. D. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*, pages 38–48. ACM, 1998.
34. M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
35. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *POPL 77: Principles of Programming Languages*, pages 132–143. ACM, 1977.
36. Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *PLDI 00: Programming Language Design and Implementation*, pages 70–82. ACM, 2000.