The SLAM Toolkit

Thomas Ball and Sriram K. Rajamani
Microsoft Research
http://www.research.microsoft.com/slam/

1 Introduction
The SLAM toolkit checks safety properties of software without the need for user-supplied annotations or abstractions. Given a safety property to check on a C program $P$, the SLAM process [4] iteratively refines a boolean program abstraction of $P$ using three tools:

- C2BP, a predicate abstraction tool that abstracts $P$ into a boolean program $BP(P, E)$ with respect to a set of predicates $E$ over $P$ [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.

Property violations are reported by the SLAM toolkit as paths over the program $P$. Since property checking is undecidable, the SLAM refinement algorithm may not converge. We have applied the SLAM toolkit to automatically check properties of device drivers taken from the Microsoft Driver Development Kit. While checking for various properties, we found that the SLAM process converges to a boolean program that is sufficiently precise to validate/invalidate the property [4].

Several ideas behind the SLAM tools are novel. C2BP is the first automatic predicate abstraction tool to handle a full-scale programming language with procedure calls and pointers, 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.

2 Overview and Example
We introduce the SLAM refinement algorithm and apply it to a small code example. We have created a low-level specification language called SLIC (Specification Language for Interface Checking) for stating safety properties. Figure 1(a) shows a SLIC specification that states that it is an error to acquire (or release) a spin lock twice in a row. There are two events on which state transitions happen — returns of calls to the functions KeAcquireSpinLock and KeReleaseSpinLock.

We wish to check if a temporal safety property $\varphi$ specified using SLIC is satisfied by a program $P$. We have built a tool that automatically instruments the 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


```c
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;
}

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;
}
```

(a) | (b)
---|---

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

creates C code from the SLIC specification, as shown in Figure 1(b). The tool then inserts calls to the appropriate SLIC C functions in the program \( P \) to result in the instrumented program \( P' \).

Now, we wish to check if the label **SLIC_ERROR** is reachable in the instrumented program \( P' \). Let \( i \) be a metavariable that records the SLAM iteration count. The first iteration \( (i = 0) \) starts with the 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 \( BP(P', E_i) \). Program \( BP(P', E_i) \) is guaranteed to **abstract** the program \( P \), as every feasible execution path \( p \) of the program \( P' \) also is a feasible execution path of \( BP(P', E_i) \).
2. Apply BEBOP to check if there is a path \( p_i \) in \( BP(P', E_i) \) 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 \( p_i \), then we use NEWTON to check if \( p_i \) is feasible in \( P' \). There are two outcomes: “yes”, the property \( \varphi \) is violated by \( P \) and the algorithm terminates with an error path \( p_i \); “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.
Figure 2(a) presents a snippet of (simplified) C code from a PCI device driver. Figure 2(b) shows the instrumented program (with respect to the SLIC specification in Figure 1(a)). Calls to the appropriate SLIC C functions (see Figure 1(b)) have been 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 first step of the algorithm is to generate the initial boolean program. A boolean program [3] is a C program in which the only type is boolean.

For our example, the initial set of predicates $E_0$ consists of two global predicates $(\text{state} = \text{Locked})$ and $(\text{state} = \text{Unlocked})$ 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 is shown in Figure 4(a). Together, these two pieces of code comprise the boolean program $BP(P', E_0)$ output by C2BP.
| decl {state==Locked}, |
| {state==Unlocked} := F,T; |
| void slic_abort() begin |
| SLIC_ERROR: skip; |
| end |
| void KeAcquireSpinLock_return() begin |
| if ({state == Unlocked}) |
| 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.

As shown in Figure 3, the translation of the SLIC C code results in the global boolean variables, `{state==Locked}` and `{state==Unlocked}`. 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 $e$ and codes that effect in the boolean program. 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 an alias analysis to determine whether or not an assignment statement through a pointer dereference can affect a predicate $e$.

The second step of our process is to determine whether or not the label `SLIC_ERROR` is reachable in the boolean program $BP(P', E_0)$. We use the BeBoF model checker to determine the answer to this query. In this case, the answer is “yes” and BeBoF produces a (shortest) path $p_0$ leading to `SLIC_ERROR` (specified by the sequence of labels $[A, A, SLIC\_ERROR]$).

Does $p_0$ represent a feasible execution path of $P'$? The NEWTON tool takes a C program and a (potential) error path as an input. It uses verification condition generation to determine if the path is feasible. If the path is feasible, we have found a real error in $P$. 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 $p_0$ is infeasible, and returns a single predicate ($n\text{Packets} = n\text{packetsOld}$) as the explanation for the infeasibility.

---

1 Boolean programs permit a variable identifier to be an arbitrary string enclosed between “[” and “]”.
 Figure 4. The two boolean programs created while checking the code from Figure 2(b).

<table>
<thead>
<tr>
<th>(a) Boolean program $BP(P', E_0)$</th>
<th>(b) Boolean program $BP(P', E_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>void example() begin do skip; A: KeAcquireSpinLock_return(); skip; if (<em>) then skip; B: KeReleaseSpinLock_return(); skip; if (</em>) then else skip; fi skip; fi while (*); skip; C: KeReleaseSpinLock_return(); end</td>
<td></td>
</tr>
<tr>
<td>void example() begin do skip; A: KeAcquireSpinLock_return(); b := T; if (<em>) then skip; B: KeReleaseSpinLock_return(); skip; if (</em>) then else skip; fi skip; fi while (t); skip; C: KeReleaseSpinLock_return(); end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(a) Boolean program $BP(P', E_0)$</th>
<th>(b) Boolean program $BP(P', E_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>void example() begin do skip; A: KeAcquireSpinLock_return(); skip; if (<em>) then skip; B: KeReleaseSpinLock_return(); skip; if (</em>) then else skip; fi skip; fi while (*); skip; C: KeReleaseSpinLock_return(); end</td>
<td></td>
</tr>
<tr>
<td>void example() begin do skip; A: KeAcquireSpinLock_return(); b := T; if (<em>) then skip; B: KeReleaseSpinLock_return(); skip; if (</em>) then else skip; fi skip; fi while (t); skip; C: KeReleaseSpinLock_return(); end</td>
<td></td>
</tr>
</tbody>
</table>

References