Program Analysis December 20, 2015

Lecture 9

Lecturer: Shmuel Sagiv *Scribe: Aviv Kuvent, Asya Frumkin*

1 SLAM

The motivation for this subject arisen in the operating systems field. Since the Windows device driver API is quite complicated and drivers are written in C, it is very easy to introduce bugs there. And indeed, Microsoft blames most Windows crashes on third party device drivers. Therefore, Microsoft developed a tool, called SLAM, to automatically check device drivers for certain errors.

The SLAM tool is given precise usage rules for the device, that are written in a specification language called SLIC. If the input source code doesn't obey those rules, it reports a bug.

2 Counter Example Guided Refinement (CEGAR)

So far, we've seen many kinds of abstract domains in the course, for example: signs, constant propagation, intervals etc. We can define a lattice of abstractions, where every element is an abstract domain, and $A \sqsubseteq A'$ if there exists a Galois Connection from A to A'.
We would like to choose the simplest abstract domain for a certain program. This choice incorporates a precision (less false alarms) vs. scalability trade-off. However, sometimes a more precise solution improves scalability by not considering infeasible program paths.

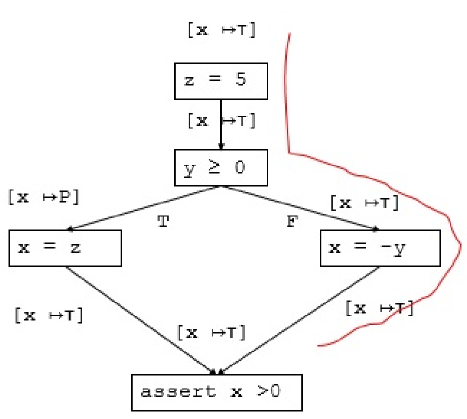
The idea is to specialize the abstraction for the desired property. The analysis starts with a simple abstract domain. If the property is verified we are done. Otherwise, if the analysis reports an error, we use a term prover in order to find out if the error is feasible by symbolic reasoning. If it is feasible, we can generate a concrete trace of the occurred error. If it is not feasible, we can refine the abstract domain and run the analysis again. Note, that this process may not terminate, as the domain is refined over and over.

Example 1 *Consider the following code segment:*

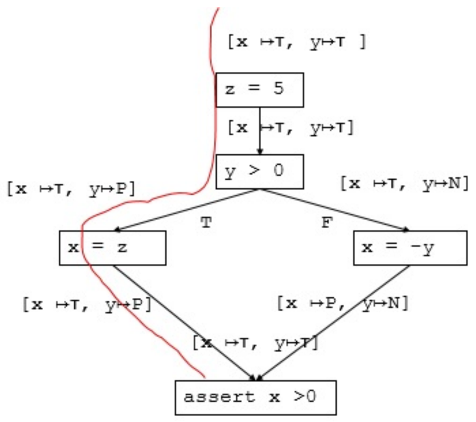
```
z = 5;
if (y >0)
   x = z;
else
```

```
  x = -y;
assert x >0
```

If we use the abstraction sign(x) and consider the path where $x = -y$ we might get a false alarm on the assert, as seen in figure 1a. Adding sign(y) to the abstraction and preforming



(a) False alarm on one path



(b) False alarm on a different path

another analysis is not enough as we observe a false alarm on a different path in figure 1b. Only after adding sign(z) to the abstraction we will get no false alarms and the analysis will terminate.

Note, that we can use different abstractions in different points of the program. For example,

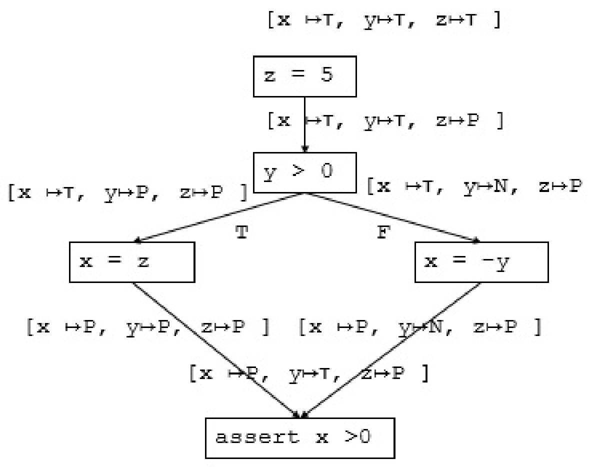


Figure 2: Refined final abstraction

we can save sign(x) in certain points in the program and sign(x), sign(y) in other points.

3 Predicate Abstraction

We are interested in proving some safety property of a program. As we've seen before, we can use predicates, like signs and aliases, in order to express program state. States that satisfy the same predicates are equivalent and are merged to a single abstract state. The number of possible equivalence classes for states is exponential.
Two questions arise when we create such an abstraction:

1. Which predicates to use?
2. How to write the transformers?

The predicate abstraction domain is composed from a fixed set of predicates $Pred$. Each predicate $p_i \in Pred$ is a closed first order formula f_i, and the lattice is $< P(P(Pred)), \emptyset, P(Pred), \cup, \cap >$.
The analysis is exponential in the number of predicates and hence there exists a scalability problem. However, for most programs it is enough to consider only a small number of predicates in order to prove interesting properties.

Example 2 *We are given the following program:*

```
int x, y;
x = 1;
y = 2;
while (*) do {
  x = x + y;
}
assert x > 0;
```

The used predicates are

$$p1 = x > 0$$

$$p2 = y \geq 0$$

*We translate the integer program to a new program that uses these predicates. The new program has only 3 possible values: true, false and *.*

```
bool p1, p2;
p1 = true;
p2 = true;
while (*) do {
  p1 = (p1&&p2 ? 1 : *)
}
assert p1;
```

We claim that if the assertion is true in the abstracted program, it is true in the original program. The opposite is not necessarily true.

In this way, we simplified the verification problem, and got a decidable problem - the problem of Boolean satisfiability.

A more complicated SLAM example:

Example 3 *The code below handles a spin lock. The functions KeAcquireSpinLock and KeReleaseSpinLock are API functions that have a specific semantics that involves locks:*

```
do {
   KeAcquireSpinLock();
   nPacketsOld = nPackets;
   if(request){
      request = request->Next;
      KeReleaseSpinLock();
      nPackets++;
   }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock();
```

In the first step SLAM handles only the code that calls the API and uses a single predicate for the lock. The tool finds a path where an error exists, and later all the code is passed to the term prover in order to determine if the found path is feasible or not.
In our case, the term prover finds that the path is infeasible. Hence, a new predicate:

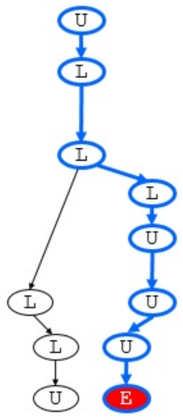


Figure 3: Infeasible path for locking code

$b = (nPacketsOld == nPackets)$ *is added to the Boolean program.*

Here, we see again the precision vs. scalability trade-off: more predicates give more precise but less scalable analysis. Less predicates give more scalability.

Example 4 *We can represent the following code, that has a non-deterministic loop by an automata with 3 states:*

```
while(*){
1:  if  (p1)  lock();
    if  (p1)  unlock();

2:  if  (p2)  lock();
    if  (p2)  unlock();

n:  if  (pn)  lock();
    if  (pn)  unlock();
}
```

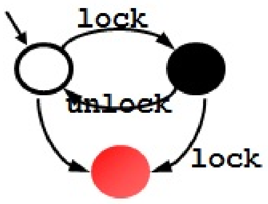


Figure 4: Automata representing the locking state

The property we would like to prove is that we don't preform two consecutive locks or unlocks. If we only track lock, our analysis is scalable. However, we get bogus counterexamples as we don't consider the correlations between branches.
If we track p_i's as well as lock, we get a state explosion as there are 2^n distinct states.
We note, that each p_i predicate is used in a specific section in the code and is not needed before or after. We can use this notion of locality in order to improve scalability. This way, the number of distinct states is reduced to $2n$.

4 Refinement

Suppose, that our analysis gave us an infeasible path to an error. Now we would like to understand how to preform the refinement in the algorithm in order to avoid this output. This is done using traces.

An **abstract trace** of a program represents an infinite number of concrete traces. Each concrete trace may have a different number of loop iterations and different concrete values. Thus, in order to simplify we consider only concrete traces with the same number of executions and same control locations.

We use formulas in order to represent sets of states. $\wedge$ is used for path intersection and $\vee$ for path merge. A trace is translated into a formula that is given to the theorem prover in order to determine if it's feasible or not.

Before obtaining a trace feasibility formula we translate a trace into an **SSA trace**. This is done by renaming variables such that variable names are unique. If the formula is satisfiable, the trace is feasible and a bug was found. Otherwise, the proof of unsatisfiability can be used to generate new predicates.

5 Proof of Unsatisfiability

We would like to extract predicates from the unsatisfiability proof of a trace formula. Such predicates can then be used to ensure we will not encounter this trace again during refinement of the abstraction.
Consider the following trace formula:

$$x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1 \wedge x_1 = i_0 - 1 \wedge y_1 \neq i_0$$

We can prove the unsatisfiability of this formula using resolution:
$x_1 = ctr_0$ and $x_1 = i_0 - 1$ imply $ctr_0 = i_0 - 1$;
$ctr_0 = i_0 - 1$ and $ctr_1 = ctr_0 + 1$ imply $ctr_1 = i_0$;
$ctr_1 = i_0$ and $y_1 = ctr_1$ imply $y_1 = i_0$;
$y_1 = i_0$ and $y_1 \neq i_0$ give us contradiction - therefore this formula is unsatisfiable.

This process proves that the formula is unsatisfiable. However, it is not localized - the proof requires the entire history of execution. In addition, it is not clear how we can extract from this proof the minimal predicate that causes this formula to be unsatisfiable.

We require a more localized process: at each point of the program in the given trace, we want to calculate the **present state**:

1. The state after execution of the trace prefix for this point.
2. The state that is only aware of the present values of variables at this point (vocabulary that already appeared in the trace).
3. The state which makes the trace suffix for this point unsatisfiable.

For each such **present state** at each point of the program in this trace, we want the **equivalent predicate** at this point in the trace formula:

1. The predicate that is implied by the prefix of the trace formula for this point.
2. The predicate that only contain variables which are common to the prefix and suffix of the trace formula for this point (common vocabulary).
3. The predicate for which its conjunction with the trace formula suffix for this point is unsatisfiable.

Craig's Interpolation theorem (1957) allows us to find the predicates (one for each point of the program in the trace formula) that have these 3 properties.

6 Craig's Interpolation Theorem

Theorem 1 *Given 2 formulas - Ψ^-, Ψ^+, s.t. $\Psi^- \wedge \Psi^+$ is unsatifiable, then there exists an interpolant Φ for Ψ^-, Ψ^+ s.t.*

1. $\Psi^- \implies \Phi$
2. *Φ has only symbols common to Ψ^- and Ψ^+*
3. *$\Phi \wedge \Psi^+$ is unsatisfiable*

Taking Ψ^- as the prefix of the trace formula and Ψ^+ as the suffix of the trace formula for each point of the program, the interpolant is the predicate we require for that point of the program.

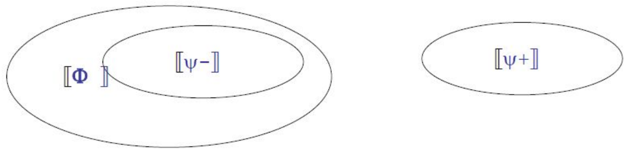


Figure 5: Diagram form of the theorem. Each circle represents the different assignments which satisfy the relevant formula

Similarly, we can write the same theorem using implication instead of conjunction

Theorem 2 *Given 2 formulas - Ψ^-, Ψ^+, s.t. $\Psi^- \implies \neg\Psi^+$, then there exists an interpolant Φ for Ψ^-, Ψ^+ s.t.*

1. $\Psi^- \implies \Phi \implies \neg\Psi^+$
2. *Φ has only symbols common to Ψ^- and $\neg\Psi^+$*

We will now look at some examples of Craig's Interpolation:

Example 5 $\Psi^- = b \wedge (\neg b \vee c)$
$\Psi^+ = \neg c$
We see that $\Psi^- \wedge \Psi^+$ is unsatisfiable, so we can calculate the interpolant.
The common vocabulary is marked in red.
The interpolant is: $\Phi = c$:

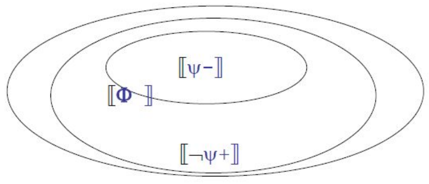


Figure 6: Diagram form of the theorem. Each circle represents the different assignments which satisfy the relevant formula

1. *$(b \wedge (\neg b \vee c)) \implies c$*
2. *it uses only the common vocabulary - c.*
3. *$c \wedge \neg c$ is unsatisfiable.*

Example 6 $\Psi^- = x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge\ y_1 = ctr_1$
$\Psi^+ = x_1 = i_0 - 1 \wedge\ y_1 \neq i_0$
We see that $\Psi^- \wedge \Psi^+$ is unsatisfiable, so we can calculate the interpolant.
Here as well, the common vocabulary is marked in red.
The interpolant is: $\Phi = y_1 = x_1 + 1$:

1. *$(x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1) \implies (y_1 = x_1 + 1)$*
2. *it uses only the common vocabulary - x_1 and y_1.*
3. *$y_1 = x_1 + 1 \wedge x_1 = i_0 - 1 \wedge y_1 \neq i_0$ is unsatisfiable.*

There are different tools which allow calculating the interpolant. This tools attempt to extract the interpolant from the unsatisfiability proof of $\Psi^- \wedge \Psi^+$ ([6], [5]).

Due to the SSA performed on the trace, the common vocabulary for prefix and suffix at a specific point in the program actually represent the current values of variables at this point (in the trace given above, the variable *ctr* is split during SSA phase to ctr_0 and ctr_1, each representing different values for *ctr* at different points of the program).

7 Predicate Map

Using Craig's interpolation, we can extract an interpolant for each point of the program in the trace and construct a predicate map.

Definition 3 *Splitting a trace into a prefix and suffix in order to extract an interpolant at a specific point is call a cut.*

Example 7 *We will now show an example of how to construct the predicate map for a specific trace:*

Trace	*Trace Formula*
$pc_1 : x = ctr$	$x_1 = ctr_0$
$pc_2 : ctr = ctr + 1$	$\wedge\ ctr_1 = ctr_0 + 1$
$pc_3 : y = ctr$	$\wedge\ y_1 = ctr_1$
$pc_4 :$ *assume(*$x = i - 1$*)*	$\wedge\ x_1 = i_0 - 1$
$pc_5 :$ *assume(*$y \neq i$*)*	$\wedge\ y_1 \neq i_0$

To construct the predicate map, we perform the following cuts:

1. *We take:*
   $\Psi^- = x_1 = ctr_0$
   $\Psi^+ = ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1 \wedge x_1 = i_0 - 1 \wedge y_1 \neq i_0$
   Calculating the interpolant, we get:
   $\Phi = x_1 = ctr_0$
   We then add this mapping to the predicate map:
   $pc_2 : x = ctr$

2. *We take:*
   $\Psi^- = x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1$
   $\Psi^+ = y_1 = ctr_1 \wedge x_1 = i_0 - 1 \wedge y_1 \neq i_0$
   Calculating the interpolant, we get:
   $\Phi = x_1 = ctr_1 - 1$
   We then add this mapping to the predicate map:
   $pc_3 : x = ctr - 1$

3. *We take:*
   $\Psi^- = x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1$
   $\Psi^+ = x_1 = i_0 - 1 \wedge y_1 \neq i_0$
   Calculating the interpolant, we get:
   $\Phi = y_1 = x_1 + 1$
   We then add this mapping to the predicate map:
   $pc_4 : y = x + 1$

4. *We take:*
   $\Psi^- = x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1 \wedge x_1 = i_0 - 1$
   $\Psi^+ = y_1 \neq i_0$
   Calculating the interpolant, we get:
   $\Phi = y_1 = i_0$
   We then add this mapping to the predicate map:
   $pc_5 : y = i$

The predicate map we built is:

$pc_2 :\quad x = ctr$
$pc_3 :\quad x = ctr - 1$
$pc_4 :\quad y = x + 1$
$pc_5 :\quad y = i$

Theorem 4 *The predicate map makes a trace abstractly infeasible*

Since we constructed the map by choosing predicates that are implied from the prefix of each cut and make the suffix unsatisifiable, the above theorem trivially follows (immediate from Craig's theorem).

Using these predicates, we can make the trace abstractly infeasible (the trace was originally concretely infeasible but abstractly feasible which is why we encountered during the analysis using the abstraction), and thus refine the abstraction.

8 Interprocedural Analysis

Using the method above to extract the relevant predicates becomes problematic when we have procedure calls in the trace.
The method above does not differentiate between the different scopes of the variables (the common vocabulary can contain variables of different scopes), so when calculating interpolant for cuts inside of a called procedure, we will get predicates which are not well-scoped. They might contain variables which are local to the caller, and not only variables which are visible at the point of the cut.

Instead, [2] suggests a variation of the method above to calculate suitable predicates in the case of procedure calls:
For simplicity, assume that there are no global variables and that the functions return a single integer.
We start with computing the effect of the function in terms of the arguments passed to it using polymorphic predicate abstractions ([4], [3]).
After performing the polymorphic predicate abstractions for the functions, we assume that the program now contains, for each function f and for each parameter x to this function:

1. A new symbolic variable f_x, local to f, which holds the value of the argument when the function f is called and is never written to.
2. The first statement of a function f is "assume $\wedge_i x_i = f_{x_i}$ for all formal parameters of this function.

[2] now generalizes the notion of cuts in the following manner:
For each location i of a trace, instead of partitioning to operations before i and after i, we

use a finer-grained partition:

1. Ψ_a^- — all operations which are before i and are before the scope of the current procedure in which i is located.
2. Ψ_b^- — all operations which are before i and are in the scope of the current procedure in which i is located.
3. Ψ_b^+ — all operations which are after i and are in the scope of the current procedure in which i is located.
4. Ψ_a^+ — all operations which are after i and are after the scope of the current procedure in which i is located.

We then calculate an interpolant where we take $\Psi^- = \Psi_b^-$ and $\Psi^+ = \Psi_a^- \wedge \Psi_b^+ \wedge \Psi_a^+$.
In this case, the common vocabulary will only refer to current values of the relevant variables for the scope in which i is located.

Example 8 *Consider the following trace (the function code appears inlined and the polymorphic predicate abstraction was performed):*

Program Counter	*Operation*	*Trace Formula*	*Partition*
pc_1 :	$a = 0$	$a_0 = 0$	Ψ_a^-
pc_2 :	$b = inc(a)$:	$f_{x_0} = 1$	Ψ_a^-
pc_2 :	f : $assume\ (x = f_x)$	$x_0 = f_{x_0}$	Ψ_b^-
pc_3 :	f : $x = x + 1$	$x_1 = x_0 + 1$	Ψ_b^-
pc_4 :	f : $r = x$	$r_0 = x_1$	Ψ_b^+
pc_5 :	$return\ r$	$b_0 = r_0$	Ψ_a^+
pc_6 :	$assume\ (a \neq b - 1)$	$a_0 \neq b_0 - 1$	Ψ_a^+

In this example, from the partition it can be seen that i is located between pc_3 and pc_4.

9 Results

[2] has implemented the interpolant calculation algorithms in BLAST, using the VAMPYRE proof generating theorem prover.
Below are the results for running BLAST on several Windows NT device drivers, checking a property related to the handling of I/O Request packets:

1. Program — the name of the program being analyzed.
2. LOC — number of lines of code.

3. Previous — results of running BLAST without Craig's interpolation.
4. Craig — results of running BLAST with Craig's interpolation, using it to discover predicates. It does not track different sets of predicates at different program locations.
5. Craig + Locality — results of running BLAST with Craig's interpolation, using it to discover predicates. It only tracks the relevant predicate at each program location.
6. Predicates (total) — the total number of predicates required for Craig+Locality.
7. Predicates (avg) — the average number of predicates tracked at a program location for Craig+Locality.

DNF means the tool did not finish in 6 hours.

Program	LOC	Previous	Craig	Craig + Locality	Predicates (total)	Predicates (avg)
kbfilter	12301	1m12s	0m52s	3m48s	72	6.5
floppy	17707	7m10s	7m56s	25m20s	240	7.7
diskperf	14286	5m36s	3m13s	13m32s	140	10
cdaudio	18209	20m18s	17m47s	23m51s	256	7.8
parport	61777	DNF	DNF	74m58s	753	8.1
parclass	138373	DNF	42m24s	77m40s	382	7.2

In some cases the previous BLAST and BLAST with only the Craig interpolation can be faster than Craig with locality when we start with an empty set of predicates, since Craig with locality might rediscover the same predicate at several different program locations. However, Craig with locality uses much less memory (tracking less predicates than without locality), and subsequent runs are faster, and the proof trees are smaller.

10 Limitations of CEGAR

There are several limitations to using CEGAR:

1. Only works for abstract domains which are disjunctive.
2. Interpolant computation can be complex.
3. Not clear how to handle interactions with widening during the analysis.
4. The abstract domain is very crude — the analysis might lose a lot of precision or refinement might not be successful.
5. Might perform unnecessary refinement steps.
6. Might have long refinement steps or infinite number of refinement steps.
7. Need to handle long traces.

10.1 Unsuccessful Refinement

Example 9 *Consider the following program:*

```
$x = malloc();$
$y = x;$
$while\ (...)$
    $t = malloc();$
    $t \to next = x;$
    $x = t;$
$...$
$while\ (x \neq y)\ do$
    $assert\ (x \neq null);$
    $x = x \to next;$
```

In this program, the assert $(x \neq null)$*; will never be reached. However, the CEGAR method will attempt to perform infinite number of steps, each time unrolling the loop another iteration, and will not be able to correctly recognize this property.*

10.2 Long Traces

Example 10 *Consider the following program flow:*

```
$pc_1$ :  $c = 0$
$pc_2$ :  $for\ (i = 0; i < 1000; i++)$
$pc_3$ :      $c = c + f(i)$
$pc_4$ :  $if\ (a > 0)$
$pc_5$ :      $if\ (x == 0)$
$pc_6$ :          $ERR$
```

In this example, we see that ERR is reachable (a and x are unconstrained). However, every feasible path prior to the error must unroll the loop 1000 times and find feasible paths through f*.*
Intuitively, the loop is irrelevant. We would like to use static analysis to report a statement is reachable without finding a feasible path.
For that, we can use path slicing, as defined by [7].

Definition 5 Path Slice *— A path slice of a program path* π *is a sub-sequence of the edges of* π *s.t. if the sequence of operations along the sub-sequence is:*

1. *infeasible — then* π *is infeasible.*
2. *feasible — then the last location of* π *is reachable (but not necessarily along* π*).*

We can use the path splicing to determine that the ERR in the example above is reachable without needing to unroll the loop 1000 times and finding a feasible path through f*.*

References

[1] Ball, Rajamani. The SLAM Project: Debugging System Software Via Static Analysis, POPL 2002.

[2] Henzinger, Jhala, Majumdar, McMillan. Abstractions From Proofs, POPL 2004.

[3] Reps, Horwitz, Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability, ACM 1995.

[4] Ball, Millstein, Rajamani. Polymorphic Predicate Abstraction, ACM 2005.

[5] McMillen. Interpolation and SAT-based Model Checking, Berkeley 2003

[6] Krajicek. Interpolation Theorems, Lower Bounds for Proof Systems and Independence Results for Bounded Arithmetic, Journal of Symbolic Logic 1997.

[7] Jhala, Majumdar. Path Slicing, PLDI 2005.