Error Localization And System Repair

Shyam Rajagopalan

EECS 219c

Motivation

- Close to 70% of project time for chip-design is spent on verification [1]
- Industrial verification centers around assertion-based debugging [1]
  - E.g. LTL Property: G (request -> F ack)
  - E.g. C-style assert: assert(x == y);
- If Model Checker verifies the property
  - Assertion is true for design
- If Model Checker fails to verify the property
  - Tool returns an error-trace
  - Then, what?

Two Approaches

- Error traces tend to be long and complex. Reducing the amount of information a designer/verifier has to process reduces the time spent on debugging (Localization)
  - Source: Groce et al: Error Explanation with Distance Metrics

- Automate the repairing process itself. Rather than displaying what went wrong with the program, display suggestions for how the program could be fixed
  - Source: Grismayer et al, Repair of Boolean Programs with an application to C

Outline of Error Explanation

- SSA form and Distance Metrics
- Notions of Causality
- Which parts of the error-trace were relevant to the error?
- Which parts of the error-trace should be presented to the user?

Guiding Example

```
1 Void MiniMax (int input1, int input2, int input3)
2 {
3         int least = input1;
4         int most = input1;
5         if (most < input2)
6               most = input2;
7         if (most < input3)
8               most =  input3;
9         if (least > input2)
10              most = input2;       (ERROR!)
11        if (least > input3)
12              least = input3;
13        assert (least <= most);
14 }
```

Static Single Assignment (SSA) Form

```
1 Void MiniMax (int input1, int input2, int input3)
2 {
3           int least = input1;
4           int most = input1;
5           if (most < input2)
6                   most = input2;
7           if (most < input3)
8                   most =  input3;
9           if (least > input2)
10                  most = input2;
11          if (least > input3)
12                  least = input3;
13          assert (least <= most);
14  }
```

```
{-14} least#0 == input1#0
{-13} most#0 == input1#0
{-12} \guard#1 == (most#0 < input2#0)
{-11} most#1 == input2#0
{-10} most#2 == (\guard#1 ? most#1 : most#0)
{-9} \guard#2 == (most#2 < input3#0)
{-8} most#3 == input3#0
{-7} most#4 == (\guard#2 ? most#4 : most#3)
{-6} \guard#3 == (least#0 > input2#0)
{-5} most#5 == input2#0
{-4} most#6 == (\guard#3 ? most#5 : most#4)
{-3} \guard#4 == (least#0 > input3#0)
{-2} least#1 == input3#0
{-1} least#2 == (\guard#4 ? least#1 : least#0)
-------------------------
{1} least#2 <= most#6
```

- CBMC uses loop unrolling (with known finite depths) and SSA form to convert every c-program into a series of single assignments
- CBMC plugs in CNF equivalent of clauses:

$$(\{-14\} \wedge \{-13\} \wedge \ldots \wedge \{-1\} \wedge \neg\{1\})$$

CBMC continued

{-14} least#0 == input1#0
{-13} most#0 == input1#0
{-12} \guard#1 == (most#0 < input2#0)
{-11} most#1 == input2#0
{-10} most#2 == (\guard#1 ? most#1 : most#0)
{-9} \guard#2 == (most#2 < input3#0)
{-8} most#3 == input3#0
{-7} most#4 == (\guard#2 ? most#4 : most#3)
{-6} \guard#3 == (least#0 > input2#0)
{-5} most#5 == input2#0
{-4} most#6 == (\guard#3 ? most#5 : most#4)
{-3} \guard#4 == (least#0 > input3#0)
{-2} least#1 == input3#0
{-1} least#2 == (\guard#4 ? least#1 : least#0)

{1} least#2 <= most#6

input1#0 = 1
input2#0 = 0
input3#0 = 1
least#0 = 1
most#0 = 0
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Distance Metrics

- How close/far away are two error traces?
- Apply the concept of a distance metric

1. *Nonnegative property:* $\forall a \;.\; \forall b \;.\; d(a,b) \geq 0$
2. *Zero property:* $\forall a \;.\; \forall b \;.\; d(a,b) = 0 \Leftrightarrow a = b$
3. *Symmetry:* $\forall a \;.\; \forall b \;.\; d(a,b) = d(b,a)$
4. *Triangle inequality:* $\forall a \;.\; \forall b \;.\; \forall c \;.\; d(a,b) + d(b,c) \geq d(a,c)$

Distance Metric in CMBC

- Represent executions of program P as a set of assignments using SSA form
- Execution a : {v0 = val_0; v1 = val_1 ..}
- Execution b : {v0 = val_0'; v1 = val_1'.. }
- Because of SSA form, executions a and b perform assignment to the same sequence of assignments
- $d(a,b) = \sum \Delta(i)$ where $\Delta(i) = (val\_i' == val\_i) ? 0 : 1$
- Distance Metric is the number of differing assignments in the execution path.

Sample Distance Metric Calculation

Execution trace a:
input1#0 = 1
input2#0 = 0
input3#0 = 1
least#0 = 1
most#0 = 0
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Execution trace b:
input1#0 = 1
input2#0 = 0
input3#0 = 0
least#0 = 1
most#0 = 0
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

=> d(a,b) = 1

Error Explanation Procedure

- Use SAT Solver to solve: Prog. AND (NOT Spec)
  - (generates counterexample)
- Use explain tool to generate closest valid execution of P
- Compute Δ's between valid and invalid executions
- Perform Slicing Step to reduce number of Δ's that must be presented to the user

Finding the valid closest execution

- First Method:
  - Solve SAT instance of (Program and Spec)
  - Encode required distance, i.e the sum of the Δ(i)'s into the SAT problem. For a fixed error trace a, encode d(a,b) = n directly into the SAT problem by requiring exactly n of the Δ's to be 1.
  - Then iteratively solve for various values of n
  - In practice this is not very efficient
    - Encoding that exactly n of the Δ's should be 1 results in large problems and state space explosion for long error traces.

Finding the closest execution

- Second Method:
  - Use a Pseudo-Boolean solver (PBS)
  - A PBS solver can accept a SAT problem in CNF and maximizes a pseudo-boolean expression objective function
  - A pseudo-boolean formula is of the form:

$$\sum_{i=1}^{n} c_i * d_i$$

where di is a boolean variable, and ci is a rational constant

  - Use ci = 1 and di as each of the Δi variables and minimize d(a,b)

Example of finding a close valid execution

Error trace a:
input1#0 = 1
input2#0 = 0
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Closest Successful Trace a':
input1#0 = 1
input2#0 = 1
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 1
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = FALSE
most#5 = 1
most#6 = 1
\guard#4 = FALSE
least#1 = 1
least#2 = 1

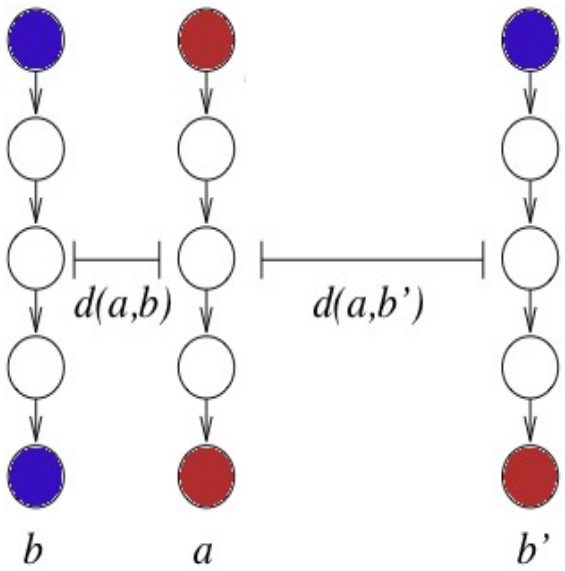
Definition of Causality

- A predicate e is **causally dependent** on a predicate c in an execution trace a iff:

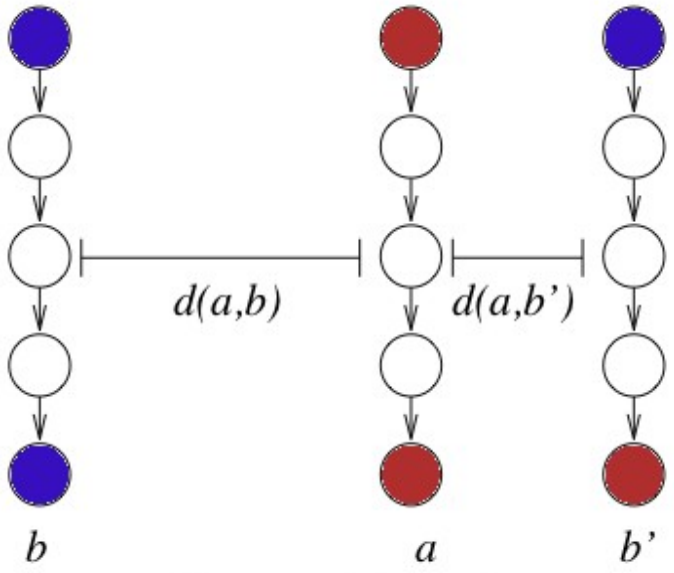
$$c(a) \wedge e(a)$$
$$\exists b \neg c(b) \wedge \neg e(b)$$
$$(\forall b', \neg c(b') \wedge e(b') \implies d(a,b) < d(a,b'))$$

- What does this mean?

Illustration



Causal dependence



No causal dependence

Inspiration for Algorithm

Theorem: let a be the counterexample trace and b be any closest successful execution to a. Let D be the set of Δs for which the values in a and b differ. If c is a predicate stating that an execution disagrees with b for at least one of these values, and e is the proposition that an error occurs, e is causally dependent on c in a.

Inspiration for algorithm

- David Lewis's theory [2] is that explanation is the analysis of causal relationships.

- Presenting the set of differences between the erroar trace and the closest successful trace satisfies the definition of explaining the error.

Example of finding a close valid execution

Error trace a:
input1#0 = 1
input2#0 = 0
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Closest Successful Trace a':
input1#0 = 1
input2#0 = 1
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 1
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = FALSE
most#5 = 1
most#6 = 1
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Presenting traces to a user

```
1 Void MiniMax (int input1, int input2, int input3)
2 {
3         int least = input1;
4         int most = input1;
5         if (most < input2)
6                 most = input2;
7         if (most < input3)
8                 most =  input3;
9         if (least > input2)
10                most = input2;       (ERROR!)
11        if (least > input3)
12                least = input3;
13        assert (least <= most);
14 }
```

∆-Slicing

- ∆'s might contain assignments to some variable z that is not relevant to failed assertion

Guiding Example: Let input1 = 1, input2 = 1; and then let input1 = 1, input2 = 0; line 7 would be part of ∆, but is irrelevant to failed assertion

```
1   Int main () {
2   int input1, input2;
3   int x = 1, y = 1, z = 1;
4   if (input1 > 0) {
5      x += 5;
6     y += 6;
7     z += 4;
8   }
9   if (input2 > 0) {
10    x += 6;
11    y += 5;
12    z += 4;
13  }
14  assert ((x < 10) || (y < 10));
15  }
```

∆-Slicing

- Attempts to answer the question: “What is the smallest subset of changes in values between these two executions that results in a change in the value of the predicate”
- Further reduce the number of lines that a designer has to examine

Δ-Slicing (2)

- Let a be the error trace and b be the closest successful trace
- Construct a new PBS problem:
  - For every variable Vi such that Δ(i) = 0, i.e Vi{a} == Vi{b}, construct a clause: (Vi = Vi{a})
- For every variable Vi such that Δ(i) = 1, i.e (Vi{a} != Vi{b}) introduce a new clause:

$$(Vi = Vi\{a\}) \vee ((Vi = Vi\{b\}) \wedge f(Vi))$$

- F(Vi) is an expression indicating that changing Vi from Vi{b} to Vi{a} at that point in the execution changes the value of the predicate (wether error occurs)

∆-Slicing (3)

- Minimizing over the same PBS formula, i.e. d(a,b), we remove all the ∆'s that were irrelevant to the change in value of the predicate
- If all the ∆'s were important to the change in success, we can't remove any slices
- However, variables that were simply changed, because of execution branch taken will be identified.
- However, the result is generally not a valid execution sequence of the program
  - This doesn't matter, since all we are interested in is localizing error

Example of finding a close valid execution

Error trace a:
input1#0 = 1
input2#0 = 0
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 0
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = TRUE
most#5 = 0
most#6 = 0
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Closest Successful Trace a':
input1#0 = 1
input2#0 = 1
input3#0 = 1
least#0 = 1
most#0 = 1
\guard#1 = FALSE
most#1 = 1
most#2 = 1
\guard#2 = FALSE
most#3 = 1
most#4 = 1
\guard#3 = FALSE
most#5 = 1
most#6 = 1
\guard#4 = FALSE
least#1 = 1
least#2 = 1

Presenting traces to a user

```
1 Void MiniMax (int input1, int input2, int input3)
2 {
3         int least = input1;
4         int most = input1;
5         if (most < input2)
6                 most = input2;
7         if (most < input3)
8                 most =  input3;
9         if (least > input2)
10                most = input2;       (ERROR!)
11        if (least > input3)
12                least = input3;
13        assert (least <= most);
14 }
```

The number of lines presented to the user can be reduced by one

Evaluating Fault Localization

- Renieris and Reiss[3] proposed the following algorithm:
  - Consider a graph, G, where nodes represent lines of code, and edges represent dependencies
  - A node in this graph is faulty if it is incorrect
  - An error report R presents a set of lines of code, i.e a set of nodes in this graph
  - Perform BFS starting from R, and let R* be the smallest layer that contains at least one faulty node
  - Error metric is: $1 - \frac{|R^*|}{|G|}$

Intuition behind benchmark

- Benchmark Measure: $1 - \frac{|R^*|}{|G|}$
- Lowest scores are achieved when |R*| is big:
  - Many nodes are presented to the user
  - Nodes are far away from faulty nodes
- Highest scores are achieved when |R*| is small:
  - Few nodes are presented to the user
  - Presented Nodes are closest to the user
- This benchmark has become accepted widely in the fault localization research community

Benchmark results

	explain			assume			JPF		R & R		CBMC	
Var.	exp	slice	time	assm	slice	time	JPF	time	n-c	n-s	CBMC	time
#1	0.51	0.00	4	0.90	**0.91**	4	0.87	1,521	0.00	0.58	0.41	1
#11	0.36	0.00	5	0.88	**0.93**	7	0.93	5,673	0.13	0.13	0.51	1
#31	0.76	0.00	4	0.89	**0.93**	7	FAIL	-	0.00	0.00	0.46	1
#40	0.75	**0.88**	6	-	-	-	0.87	30,482	0.83	0.77	0.35	1
#41	0.68	0.00	8	0.84	0.88	5	0.30	34	0.58	**0.92**	0.38	1
Average	0.61	0.18	5.4	0.88	**0.91**	5.8	0.59	7,542	0.31	0.48	0.42	1
μC/OS-II	**0.99**	**0.99**	62	-	-	-	N/A	N/A	N/A	N/A	0.97	44
μC/OS-II*	**0.81**	**0.81**	62	-	-	-	N/A	N/A	N/A	N/A	0.00	44

Summary of Error Localization

- Model Checker produces error trace
- Explain tool generates close counter example using a PBS
- Slicing removes the number of differences between the error trace and valid trace that are presented
- Slicing and Solving for the correct trace are both solved using PBS. Is there some way to combine them?

Repair of Errors

- Even with error localization techniques, the counterexample is simply a hint to the root cause of the error: some faulty piece of code
- To fix the bug, the counterexample must be analysed by a human who must identify the root cause
- It would be even more useful to automatically suggest repairs to the programmer

Repair of Errors

- Intuition: Model checker internally computes an abstraction of the c-program:
  - A boolean program
- Come up with a strategy to repair the boolean program
- Map repairs of boolean programs to repairs of c-programs to suggest a repair
- Source: Grismayer et al, Repair of Boolean Programs with an application to C

Boolean Programs

- Global Variables; Local Variables; Recursion, Assignments, Parallel assignments and Nondeterminism.
- Formalization:
  - (R, main, Vg)
  - R is a set of routines
  - Each R is (Sr, Vr)
  - Sr = (Sr,0...Sr,f) is a set of statements
  - Vr is set of local variables
  - Vr' = Vg U Vr set of visible variables
  - Let E be the subset of Vr' that is set (called Valuation)
    - Each E is in Xr = $2^{Vr'}$
  - Control flow is given by: next(E, s, s') if s' is a possible next statement of s under valuation E

Boolean Programs Continued

- The set of states of a routine is in Qr = Sr * Xr'
- For a call statement from src to dest, define a relation Us: Xsrc * Xdest
- For a return statement define Ps: Xsrc * Xdest -> Xsrc

Model-Checking Boolean Programs

- For each routine, associate an execution graph Er
- Compute Set of reachable states
- If the set ever contains an error state, i.e the set of visible variables that are on violate some assertion, then boolean program is faulty.

Requirements

- Repair should change program as little as possible
- Repairs have to depend only on local variables and global variables, i.e. only the visible variables
  - So strategy does not introduce new memory

Game Formulation

- System is protagonist
- Environment is antagonist
- Winning Strategy is one that ensures that specification is adhered to by fixing system decisions.
- If a winning strategy exists, we can fix the boolean program.

The Game

- Extend model checking algorithm
- On one iteration of the model checker, there is a transition from a good state to a bad state via a boolean expression
- This is the expression that needs to be repaired

Computing the strategy

- A possible expression is of the form Xr->Xr or is in 2^{Xr}
- Iterating over all possible expressions is computationally infeasible
- Use BDDs to share computation and examine all possible repairs simultaneously

Mapping repairs to C

- Boolean repair comes up with a list of predicates
- Each line of the boolean program corresponds to some line of the c program after abstraction
- Use meaning of these predicates to suggest repairs for c program.

Experimental Results

Driver	LoC	# Expr.	# Total	# in Driver	Time(s)	# vars	Results	Property
1394 diag	7223	273	57	8	1345	2/10	✓	MarkIrpPending
bulltlp3.1	4751	860	30	3	16482	13/15	X^1	IrpProcComplete
daytona	14364	305	2	0	379	2/0	X^1	StartIoRecursion
gameenum	4001	217	29	1	577	2/9	✓	MarkIrpPending
hidgame	3611	335	27	4	7132	9/17	X^2	LowerDriverReturn
mousefilter	1755	165	21	3	4035	7/33	✓	PendCompleteReq
parport	24379	1055	3	1	8334	2/0	✓	DoubleCompletion
pscr	4842	374	5	0	2797	6/7	X^1	IrqlReturn
sfloppy	2216	19	6	4	4	2/0	✓	AddDevice

Conclusion

- Using the concept of a distance metric, we can reduce the amount of information that a user has to look at to identify a system error
- We can also use the model checker to identify the transition on which the error occurs.
  - Using this, we can determine whether there is an automatic strategy to fix the expression so that the error state is not reached
- Using the concept of a distance metric, we can reduce the amount of information that a user has to look at to identify a system error

Outside References

- [1] Dave, Sailesh: “Assertion-Based Verification Shortens Project Design Time”, Chip Design Magazine, Issue 16, Article ID 437
- [2] Lewis, Davis: “Causation”, Journal of Philosophy 70:556-557
- [3] Reiter, R: “Fault localization with nearest neighbor queries”, Automated Software Engineer, pages 30-39