The Dangerous Graph

Anton Xue

0.1 Real-World Applications?

A friend working at **Big Tech Company** told to me about a problem they were working on: how can we have automated detection of crashing errors in Android applications?

Users interacts with Android applications by touching the phone's user interface (UI). Touching the UI in specific ways alters the internal state of the application. The UI is specified by XML files that (each?) represent a different display layout. When specific touches are performed, different displays may be shown. These display transitions are conditional on the internal state of the application and the current active display.

We then ask: **what sequence of actions will cause the application to crash?**

0.2 Defining the Problem

From a high-level perspective, the problem is not well-defined in part because we have not discussed the level of access we have to the application under testing. Are we given the source code? Can we monitor internal application state and variables during execution? Or are we given just the phone and told to play with it?

If we have access to the source code, we may apply techniques such as backwards symbolic execution [1] or slicing [4].

If we may monitor application state during execution through some debugger or trace dumper, then short of decompilation, we may try techniques such as randomized testing or somehow approximating behavior.

If we are given just a phone, then we throw it against a wall.

The first condition presented, with complete source code access, is of course, the most favorable. It is also the biggest headache because all the techniques listed are a pain to implement.

We opt for the second.

0.3 Modeling the Problem

Definition 1 (Tail and Head Set)**.** For a directed multigraph $G = (V, E)$, where each edge has form $e = (u, v) \in E$ with tail u and head v, we define the *tail set* T_u and *head set* H_v:

$$T_u = \{(u, v) : \exists v \text{ s.t. } (u, v) \in E\}$$
$$H_v = \{(u, v) : \exists u \text{ s.t. } (u, v) \in E\}$$

□

This allows us to abbreviate some notation when reasoning about multigraphs.

Definition 2 (Application Graph)**.** An *application graph* is a directed multigraph $G = (V, E)$, where each edge is labeled with a formula e_φ in some logical theory $\mathcal{F}$ such that:

(1) The formula associated with each edge in a tail set is disjoint:

$$\forall u \in V, \forall e_i, e_j \in E_u \colon e_{i,\varphi} \wedge e_{j,\varphi} \implies \bot$$

(2) The disjunction of each edge in a tail set is a tautology:

$$\forall u \in V \colon \left(\bigvee_{e \in E_u} e_\varphi \right) \implies \top$$

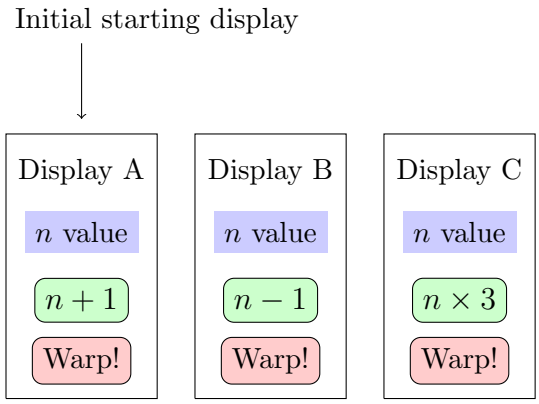
□

Here, every vertex corresponds to some display. Every edge e is intended to represent an action, and the associated formula e_φ attached to each edge is intended ot capture the logical conditions that must be satisfied when the action is performed so that the transition may take place. We require that the formula associated for each action be disjoint logical formulae, meaning that their conjunction is not satisfiable. This ensures uniqueness of action.

Example 1 (Simple UI)**.** Testing the limits of our LaTeX powers, we bring you this example. Here, the application consists of three displays, each with tree elements:

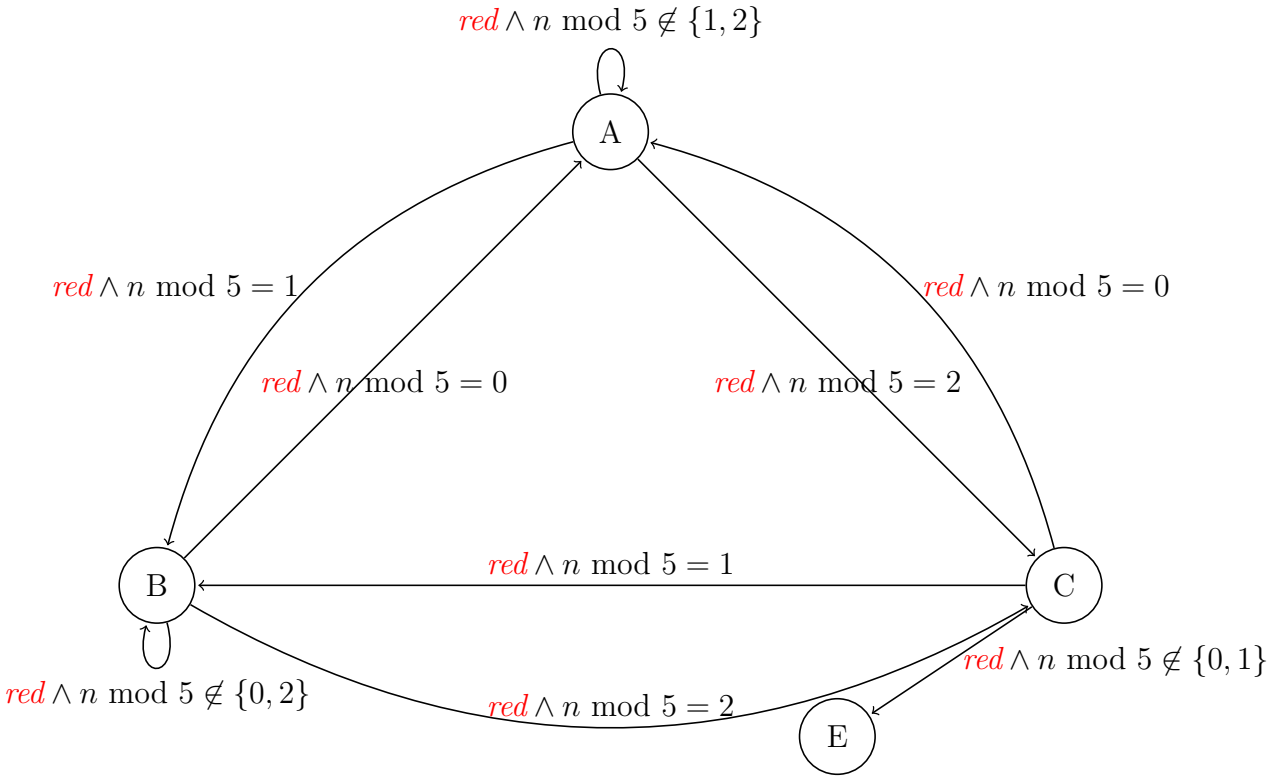
- A blue n-value label that shows the global value of n.
- A green action buton that modifies n according to its description.
- A red warp button that moves between displays. Sometimes crashes.

I think the green color hurts my eyes to read. Oof owie ouch!




```
// Display A:
green: set n = n + 1
red:   if n mod 5 == 1 then go to B
       if n mod 5 == 2 then go to C
       otherwise do nothing
// Display B:
green: set n = n - 1
red:   if n mod 5 == 0 then go to A
       if n mod 5 == 2 then go to C
       otherwise do nothing
// Display C:
green: set n = n * 3
red:   if n mod 5 == 0 then go to A
       if n mod 5 == 1 then go to B
       otherwise CRASH
```

The application graph should then look something like:



Here *red* denotes the action of the red button getting pressed. This is a mess, but I hope the idea is clear: the objective of the application grpah is to captur ethe notion of the conditions on which transitions are made when actions are performed. All actions as button presses, for the sake of simplicity. We treat the error crashing as a separate display.

We chose not to draw the green button presses *green* because otherwise the graph would be super cluttered. Actions that do not bring you to a new display are treated as self-loops. Hence, all *green* would just be self-loops. □

0.4 The Dangerous Game

Here is how we play The Dangerous Game: **starting from an empty graph of just vertices, how can we construct all the edges with the correct formula attached to each edge?**

To play the dangerous game to begin with, we have to keep track of some notion of current state of variable values:

Definition 3 (Exploration State)**.** An exploration state is a pairing (G, σ), where G is an application graph, and σ is a mapping of all relevant program variables that appear in the graph to values in some domain $\mathcal{A}$.

What are the values in $\mathcal{A}$? For now, we imagine them as logical formulae over the range of values that each variable may have with respect to their typings. Another reason for strong, static typing, yah?

There are some difficulties: it may be impossible to, without knowing the source code, to guess all the conditions that the programmer decided to encode for making the buttons transition. Nevertheless, we can try some basic, basic exploration techniques.

Algorithm 1 Naive exploration algorithm

```
1: procedure NAIVE EXPLORATION(G = (V, E))
2:     Let v_0 be the starting vertex in G
3:     v ↤ v_0
4:     while timer or ticker of some sort has not run out do
5:         Click the next button in display v according to some deterministic selection.
6:         if this transitions us to a new display v' then
7:
8:             v ↤ v'
9:         end if
10:    end while
11: end procedure
```

0.5 Conclusion

References

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *CoRR* abs/1610.00502 (2016). arXiv: 1610.00502. URL: http://arxiv.org/abs/1610.00502.

[2] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". In: *Automated Deduction – CADE-24*. Ed. by Maria Paola Bonacina. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–214. ISBN: 978-3-642-38574-2.

[3] Viktor Kuncak et al. "Complete Functional Synthesis". In: *SIGPLAN Not.* 45.6 (June 2010), pp. 316–329. ISSN: 0362-1340. DOI: 10.1145/1809028.1806632. URL: http://doi.acm.org/10.1145/1809028.1806632.

[4] Mark Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802557.