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 do we automatically detect 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 (Application Graph)**.** An *application graph* is a directed multigraph $G = (V, E)$, where each vertex $v \in V$ has a state v_σ mapping variables to a domain of values $\mathcal{A}$, and every edge $e = (u, v)$ with tail u and head v is labeled with a formula e_φ in some logical theory $\mathcal{F}$.

The reason we define the application graph representation as a multigraph is because program representation is hard.

Here, every vertex corresponds to some display, and we remark that multiple vertices may exist that represents the same display. For each vertex v, the state v_σ is meant to map the variables to a logical formula over their possible values as defined by their types. This allows us to differentiate between the same display when, perhaps the underlying state variable mappings differ.

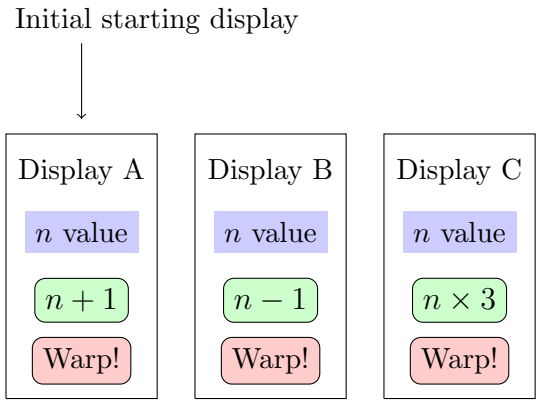
Every edge represents an action, and for each edge e the associated formula e_φ is intended to capture the logical conditions that must be satisfied when the action is performed in order for the transition to occur. We note that the conditions for edges that denote the same UI action need not be mutually unsatisfiable. This may be weird, and we attempt to address this later.

So what does this all mean?

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.

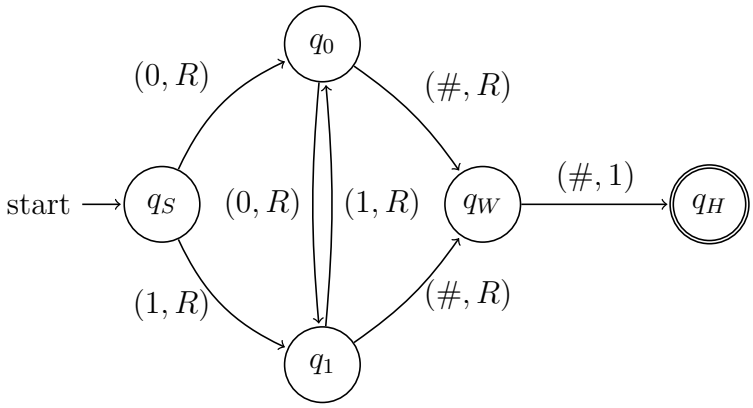
I think the green color actually hurts my eyes to read.




```
// 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 state graph is then defined as follows:

picture copied over from somewhere else; will fill in



0.4 The Dangerous Game

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.