The Dangerous Graph

Anton Xue

0.1 Real World Applications?

A friend working at **Big Tech Company** talked to me about a problem: 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. What sequence of touches will cause the application to crash?

0.2 The Problem

At a high level overview standpoint, there are a number of ways to tackle this problem: randomized testing, backwards symbolic execution [1], program slicing [4], to name a few. The techniques that may be applied depends on whether we have white-box access to the program source code.

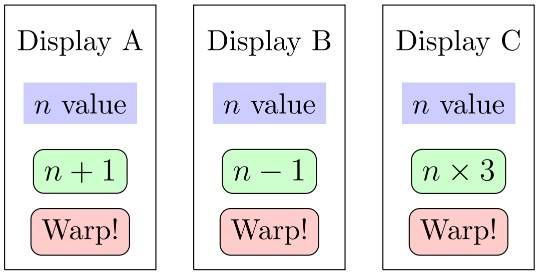
The most reliable techniques is also among the hardest to implement, and the sensible solution is perhaps one of in-depth code analysis. From personal experience, this is both hard and annoying, so we ignore it, and try something simpler.

Instead, we pretend that we do not have source code access — in fact, we make it even more confusing: we pretend that we have access to the internal program state, the UI representation of some sort, and of course a phone with the compiled application.

We then model this as follows:

Definition 1 (Application Graph)**.** An *application graph* is a directed graph $G = (V, E)$. Each $v \in V$ corresponds to a UI display, and has a state v_σ which maps variables to elements in a domain of values $\mathcal{A}$. Furthermore, every edge $e = (u, v)$ with tail u and head v has a corresponding label e_φ, where $\varphi \in \mathcal{F}$ is a formula in some logical theory $\mathcal{F}$.

Example 1 (Simple UI)**.** We push our LaTeX skills to bring you this.



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

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.