History and Context for *Defining Liveness*[1] Winner 2018 Edsger W. Dijkstra Prize

Fred B. Schneider[2]
July 2018

It's great to be back at PODC. I attended this conference religiously through 1992. It's interesting to see what has changed but also what has not changed in 25 years. And I can't think of a happier excuse to be back. All of us know how gratifying it is to see that our work is having impact, and that's what this award signifies.

To win an award that carries Dijkstra's name is especially meaningful for me. As a graduate student, I read and reread Dijkstra's work; it changed the way I looked at systems research and the research enterprise. (Bowen will offer some remarks that expand on this theme.) I then had a chance to meet Dijkstra when I served as teaching assistant for a 1-week short course he taught in Santa Cruz. That was the summer following my first year as an assistant professor at Cornell; my colleague David Gries had gotten me the assistantship. I guess I did OK, because Dijkstra and his wife Ria subsequently invited me to visit their home in Nuenen (The Netherlands) for a week. What a thrill. I subsequently saw Dijkstra at technical meetings and socially at least once a year, until he passed away. So, yes, I have many Dijkstra stories to tell. Catch me after desert for those.

You can read our paper "Defining Liveness" (it's only 5 pages!) if you are interested in the technical details. I thought that instead I would talk about how we developed those ideas and where things stand today. I'm always fascinated by the history behind discoveries, and I suspect that I'm not alone in enjoying this "academic gossip".

The idea of proving programs was discussed by Turing in 1949 (where he proved that a subroutine for factorial computed the desired result).[3] Starting in the early 1960's through the late 1970's, the CS research community embraced the idea of writing proofs for programs. In those days, you would "prove the program correct". And there was a debate about whether to

---

[1] Bowen Alpern and Fred B Schneider. Defining Liveness. *Information Processing Letters* 21, 4 (October 1985), 181–185.

[2] Department of Computer Science, Gates Hall, Cornell University, Ithaca, New York, 14853. Email: fbs@cs.cornell.edu.

[3] Turing, A.M. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematics Laboratory, Cambridge, 67-69. For a discussion of the proof, see Morris, F.L., and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing* 6, 2 (April 1984), 139- 143.

prove "partial correctness" versus "total correctness". Floyd's 1967 paper[4] had showed how to attach assertions to the edges of a flowchart; Hoare's 1969 paper[5] implemented this basic approach as a logic involving pre- and postconditions.

Lamport in a 1977 paper[6] showed how to "prove correctness" of multiprocess programs "to solve synchronization problems". He extended Floyd's work, introducing the terms "safety property" and "liveness property" to abstractly characterize different kinds of things you might want to prove. Synchronization protocols didn't produce answers, and termination was generally considered a failing, so proving partial or total correctness was not a useful goal. Petri nets were all the rage back then, as a specification language and as a way to simulate concurrent systems. The theory of Petri nets introduced the terms "liveness" and "boundedness" for describing how the assignment of a Petri net's "tokens" to its "places" could evolve; Petri net "safety" was a specific form of boundedness. Lamport borrowed those names for use in his verification work, but he gave the terms different meanings.

Fast forward to the mid 1980's. CS now had a formal methods research community that was devoted to program verification, and those researchers understood that the real problem was not verifying partial or total correctness but the more general problem of proving that a program satisfied a given specification. Some researchers were exploring temporal logics for this; others were exploring automata, because you could perform automated analysis with automata.

Bowen Alpern, who was my Ph.D. student at Cornell, was engaged in thesis research that had a foot in both camps. He understood that you could specify rich sets of program executions by using so-called Buchi-automata (which were known to be models for temporal logic formulas). A Buchi automaton was a finite-state automaton that accepted infinite sequences. Specifically, it rejected sequences of input symbols that forced the automaton to make an undefined transition or that did not infinitely-often enter the automaton's accepting states. Not all input symbols had transitions defined in every automaton state, and not every automaton state would be an accepting state; one would formulate a Buchi automaton to accept exactly those program executions (modeled as infinite sequences of program states) that satisfied the property of interest.

But Bowen didn't pursue automated analysis for Buchi automata. Instead, he showed how to perform program verification by creating a correspondence between (i) program states and automaton states, and (ii) program transitions and automaton transitions. This correspondence was validated by discharging proof obligations that resembled the verification conditions you would have with Floyd's method or Hoare's logic.

---

[4] Floyd, R.W. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics* 19, 1967, 19-31.

[5] Hoare, C.A.R. An axiomatic basis for computer programming. *CACM* 12, 10 (Oct. 1969), 576-580.

[6] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering* SE-3, 2 (March 1977), 125-143.

First, you would show that the automaton wouldn't make an undefined transition when reading the state sequence corresponding to a program execution. The obligations here involved constructing invariants that related program states and automaton states. Second, you would show that the automaton could not remain forever in non-accepting states. These obligations involved exhibiting variant or well-founded functions, which decreased in value with each program step and evaluated to a minimal element when the next program step was guaranteed to cause transition into an accepting state.

As it happens, Leslie Lamport and I were both speakers at a 2-week NATO Advanced Course on Distributed systems at Technical University of Munich in April 1984. Lamport had been using a temporal logic for reasoning about concurrent programs, and he planned to talk about that work in Munich. (Ironically, my lectures in Munich concerned Lamport's state machine approach, and those presentations were the genesis for my well known survey paper on the subject.) It was natural in motivating the use of his temporal logic for Lamport to give a formal definition of safety properties, since that enabled him to argue that his temporal logic had sufficient expressive power. He shared with me this formal definition of safety properties. He also wanted to give an expressiveness argument for liveness properties, but he had been unable to devise a formal definition of liveness. He shared that challenge with me, too.

Safety properties assert that "bad things" don't happen. That can be formalized as eschewing irremediable finite prefixes of executions; the "bad thing" is thus formalized as a set of finite prefixes. When I told Bowen about this definition, he quickly saw the connection to his work on Buchi automata: The "bad prefixes" of safety properties were those prefixes that caused the Buchi automaton to make an undefined transition.

By returning to the full acceptance criteria for Buchi automata, a formal definition of liveness now became obvious. A liveness property had to be something that stipulated every prefix would have a continuation that caused acceptance by the Buchi automaton—that is, a continuation that infinitely-often would cause the automaton to enter accepting states. So the essence of liveness was: no matter what the finite prefix, an extension could be accepted by the Buchi automaton.

I told Lamport about this proposed formal definition for liveness. He, in turn, told Gordon Plotkin, who had been experimenting with using topology to reason about classes of properties for concurrent programs. Plotkin hated our liveness definition, and he mailed us a letter, giving a topological formulation (which he attributed to a paper by Mike Smyth[7]) to present his objections. That letter was 5½ handwritten pages, and I still have a copy (albeit scanned).

Bowen and I were not convinced that the concerns Plotkin raised were reasons to abandon our proposed liveness definition. Moreover, Plotkin's letter explained that Lamport's formal definition of safety properties corresponded to the "closed sets" in a natural topology, and our liveness definition corresponded to the "dense sets". To somebody well versed in topology (which is not me), the fact that any set is the intersection of a closed set with a dense set follows

[7] Michael B. Smyth. Power Domains and Predicate Transformers: A Topological View. *ICALP* 1983, 662-675.

trivially from the definitions of closed and dense sets. This meant that we not only had a liveness definition but we also had a proof that every property was the intersection (conjunction, when formalized in a logic) of a safety property and a liveness property! Such a decomposition result seemed to us to be a truly compelling rationale in support of our liveness definition.

So let's take stock of where we were.

Bowen and I now had formal definitions along with a proof that safety and liveness were an orthogonal basis for all "properties" (though I will return to the notion of properties shortly). Bowen also had shown that safety properties required invariance proofs whereas liveness properties required variant functions.[8] Thus, there was a benefit to performing that decomposition—it told you what proof technique to use for verifying each of the different pieces of an arbitrary property. Finally, we had also established that temporal logics were not required for proving arbitrary properties of concurrent programs; the kinds of proof obligations used in Floyd's original paper were sufficient. (Recall, showing that temporal logics were not needed was the original goal of Bowen's Ph.D. thesis. Those results can be found in a TOPLAS paper[9] that has mostly been ignored.).

Now, let's skip ahead 30 years. Safety and liveness have decidedly entered into the vernacular—which is to say, they are used without citation, alas. But, after 30 years, it has also become clear that the definition we had been using for "property" was simplistic. The defining characteristic of a "property" (today, often called a "trace property") is a predicate that says whether each single execution in isolation is in that property. Yet many important aspects of system execution cannot be formalized in terms of such predicates: confidentiality, integrity, and service-level agreements, are examples. These are sets of executions that cannot be formalized as checks on an individual execution in isolation—they involve checks on pairs of executions or larger subsets. For example, confidentiality involves pairs of executions because it is a statement about correlation between the value of some variable and the value of some secret; checking for correlation requires looking at pairs of executions.

There's now some good news and some bad news. The good news: the safety/liveness orthogonal basis still works. Michael Clarkson and I introduced[10] the idea of "hyper-properties" as *sets of sets of* execution sequences, and we showed that all hyper-properties could be decomposed into hyper-safety (finite set of finite sequences) and hyper-liveness (set of infinite sequences can extend a set of prefixes). The bad news: "refinement" which was roughly "subset" for trace properties, becomes much more complicated for hyper-properties. Researchers are starting to develop proof methods for classes of hyper-properties, but nobody has identified a small set of building-block proof obligations like invariance and variant

---

[8] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed Computing* 2, 3 (1987), 117–126.
[9] Bowen Alpern and Fred B Schneider. Verifying temporal properties without temporal logic. *TOPLAS* 11, 1 (January 1989), 147–167.
[10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security* 18, 6 (September 2010), 1157–1210.

functions. So there is much work still to be done. And it is important work, because proving system security grows ever more crucial as we come to depend more and more on networked information systems.

With that, you are up to date on safety/liveness. And now you also know why and how our Defining Liveness paper came into being. Again, thank you for this honor.