Composition of Small Languages for Resilient Embedded Systems: Rethinking an “Operating System” for the Edge

Research Proposal • DC Resilient Embedded Systems

Albert Zak

This research project intends to identify novel expressive composition primitives that dissolve the divide between middleware and operating system in pervasive information infrastructures, with a focus on resilience – especially security, usability, and adaptive evolvability – and expressiveness through the stacking of dynamic domain-specific languages.

Context. With advances in hardware enabling increasingly sophisticated software, devices considered *embedded* now commonly run UNIX-like operating systems. Current software engineering methods reasonably allow for design, implementation, and operation of dependable systems, albeit at high cost when sufficient guarantees are required.

Problem. Software today is bloated. Our stack is cheap, fast, and full of features. It is also full of cruft, vulnerabilities, and we barely understand the innards anymore. With increasing need to operate at massive horizontal scale over unreliable and hostile networks, while satisfying ever higher user requirements and cost pressure to deliver more features in less time, the resulting emergent nature of such systems causes a significant drop in dependability and security [1].

Flaws in underlying abstractions and tools are papered over with more of the same. Gradually, the production and nurturing of the entire stack turns into an end itself instead a of a means toward an end [2]. Such complexity begets *permanent faults*, as they were embedded at design time. Because of the gradual change in responsibilities and shifting context of layers, the complexity incurred by performance optimizations which had been necessary on previous hardware now stands in the way. Common systems programming languages are focused on machine performance while lacking in safety and expressive power needed to describe the behavior of pervasive systems: prevalent mutable state together with its distribution over unreliable networks, weak language support for explicit consistency vs. availability decisions, and little ability to accommodate changes especially at runtime hinder prevention and tolerance of faults.

Relevance. We need novel specification and design methods which focus on dependability and resilience aspects of large scale distributed systems, especially to prevent human-made fault classes that occur during development, and to securely deal with malicious faults at runtime. Such systems need to be late-bound, evolvable, and adaptive to enable maintenance and repairs at runtime, and they cannot separate the concept of dependability from security [3].

Alan Kay's research group led one exceptional project with the goal of creating a minimal and understandable personal computing environment in less than 20,000 lines of code: STEPS [4][5] bootstraps, from bare metal, an operating system with support for vector graphics, sound, networking, a full user space including universal document editing capabilities and a hypertext browser. Contrast their achievement to the tens of millions of lines of code [6] that comprise just the kernel of common operating systems today.

The proposed endeavor is a rendition of similar ideas, albeit much simpler [7] and with a different focus as the system's intended placement on embedded devices does not require graphics or document authoring capabilities. The entire stack may benefit from removing accumulated complexity throughout the layers.

Goals. The grand vision is to rethink the basic concepts of computing and programming for a world of distributed and embedded systems. Expressiveness, dynamism, and security are favored over performance or machine optimizations.

One intent is to find abstractions that reduce and avoid state wherever possible [8]. Plain data structures replace Kay's objects [9] in the small. Objects as actors passing messages and keeping state – when coupled with a mechanism for fault tolerance as seen in Erlang/OTP [10] – appear to naturally belong towards the edge of the system, i.e. as gatekeepers for peripherals [11] and to communicate with other nodes over the network. This tenet stems from the observation that objects generally do not compose as trivially as data or pure functions do [12]. Instead of coupling state with behavior while attempting to hide all of that inside an "object", the system will decidedly be built on the primacy of plain data structures, preferably immutable [13][14], with pure functions operating on them in the small. Plain but rich data structures [15] such as maps, sets, vectors, tagged literals, and keywords as a higher-level "lowest common denominator" instead of byte streams are the system's attempt to remove barriers of cooperation between worlds of processes, nodes and languages.

The pervasive embrace of dynamic languages seems to be at odds with the state of the art in language security research, as their focus lies mostly on static typing [16], yet Laprie noticed a dissociation between resilience and stability and writes: "a system can be very resilient and still fluctuate greatly, i.e., have low stability" and that "low stability seems to introduce high resilience" [17].

The proposed ideas include a willingness to trade away performance optimizations in return for almost *everything else* – simplicity, security, conciseness, aesthetics, introspectability, late binding, developer ergonomics, etc. There will be no compilation [18] and the resulting system will likely not run well on today's embedded hardware.

Questions. Where do faults and vulnerabilities occur? What are better metrics [19] to assess security in software? How and where to slice the stack? Where to demarcate between operating system and middleware [20]? Do we even need a distinction [21]? How much of what can be removed at each layer, from the hardware up [22]? What does it take to execute an understandable specification? Can we separate meaning from optimization? How do data, state, process, and effects [23] interplay? Are there simpler primitives [24] that could collectively provide a run time system? What abstractions compose cleanly and reliably even in higher layers [25], and are abstractions even the path towards understandability [2]? In what ways can we fit an object capability security model with delegation of authority [26] within the *functionality vs. safety* tradeoff in order to achieve safe composition and reuse of untrusted functions? How can we keep the benefits of maximum visibility into the lower layers and the ability to quickly reconfigure late-bound parts of the underlying operating system?

Originality. The proposed research follows the spirit of discovery which led to ambitious working systems such as STEPS [5], Mu [2] and Lisp Machines [27] while borrowing strategies from a vast body of past work.

Some of the ideas to be combined include Kay's *"extreme late-binding of all things"* [28] that allows gradual refinement and experimentation on a running system. *Metalinguistic abstraction* [29] creates a tower of interpreters [30][31] whereby each domain-specific language [32] sits on the language below it, all of which are expressed in *homoiconic* forms [33]. Ultimately, the whole tower should strive to read as an *executable specification*, where *meaning* is separate from optimization [8][34]. From Smalltalk [35] the project takes its focus on *messaging* [36] and combines it with mechanisms for fault tolerance [10] from Erlang/OTP [37].

Other ideas include the primacy of plain data and algebraic effects expressed as data as seen in idiomatic Clojure [15], the parsimony and composability of Forth [38], and the programmatic availability of abstractions [21], meaning there is little distinction between process and operating system [39]. The project takes its ideas on security from various work on object capabilities [40][41], the E language [42][43], and security models of operating systems such as Multics [44], Minix [45], and KeyKOS [46].

Method. The proposed research will be based on iterative cycles of experimentation and construction of a minimal and understandable working system. Starting a *conversation with the machine* [47] and experimenting with its building blocks is supposed to lead towards the goal of identifying simpler building blocks which compose to form a useful runtime system. Having a small and understandable working model together with the ability to adaptively experiment with and evolve at runtime seems like a useful artifact to study.

References

[1] Jean-Claude Laprie, “Resilience for the scalability of dependability,” in *Fourth IEEE international symposium on network computing and applications*, 2005, pp. 5–6.

[2] Kartik Agaram, “Bicycles for the mind have to be see-through,” in *International conference on the art, science, and engineering of programming*, 2020.

[3] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[4] Alan C Kay, Daniel HH Ingalls, Yoshiki Ohshima, Ian Piumarta, and Andreas Raab, “Steps toward the reinvention of programming,” *Viewpoints Research Institute*, 2006.

[5] Ted Kaehler, Bert Freudenberg, Aran Lunzer, Alan C Kay, Ian Piumarta, Takashi Yamamiya, and Alan Borning *et al.*, “STEPS toward the reinvention of programming, final report submitted to the national science foundation,” *Viewpoints Research Institute*, 2012.

[6] Edward E Ogheneovo and others, “On the relationship between software complexity and maintenance costs,” *Journal of Computer and Communications*, vol. 2, no. 14, p. 1, 2014.

[7] Butler W Lampson, “Hints for computer system design,” in *Proceedings of the ninth ACM Symposium on Operating systems principles*, 1983, pp. 33–48.

[8] Ben Moseley and Peter Marks, “Out of the tar pit,” *Software practice advancement*, 2006.

[9] Daniel HH Ingalls, “The Smalltalk-76 programming system design and implementation,” in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on principles of programming languages*, 1978, pp. 9–16.

[10] Francesco Cesarini and Steve Vinoski, *Designing for scalability with Erlang/OTP: Implement robust, fault-tolerant systems.* O’Reilly Media, Inc., 2016.

[11] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller, “Devil: An IDL for hardware programming,” in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - volume 4*, 2000.

[12] Mehmet Aksit and Bedir Tekinerdogan, “Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters,” in *OOPSLA AOP’98 workshop position paper*, 1998.

[13] Chris Okasaki, *Purely functional data structures.* Cambridge University Press, 1999.

[14] Rich Hickey, “Persistent data structures and managed references.” 2009.

[15] Rich Hickey, “The Clojure programming language,” in *Proceedings of the 2008 symposium on dynamic languages*, 2008, pp. 1–1.

[16] Mark Tarver, *The Book of Shen.* 2013.

[17] Jean-Claude Laprie, “From dependability to resilience,” in *38th IEEE/IFIP Intenational Conference on Dependable Systems and Networks*, 2008, pp. G8–G9.

[18] Xavier Saint-Mleux, Marc Feeley, and Jean-Pierre David, “SHard: A Scheme to hardware compiler,” in *Workshop on Scheme and functional programming*, 2006.

[19] Michael D Brown and Santosh Pande, “Is less really more? Towards better metrics for measuring security improvements realized through software debloating,” in *12th USENIX workshop on cyber security experimentation and test (CSET 19)*, 2019.

[20] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein, “SeL4 enforces integrity,” in *International conference on interactive theorem proving*, 2011, pp. 325–340.

[21] Stephen Kell, “The operating system: Should there be one?” in *Proceedings of the seventh workshop on programming languages and operating systems*, 2013, pp. 1–7.

[22] Galen C Hunt and James R Larus, “Singularity: Rethinking the software stack,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.

[23] Wouter Swierstra, “A functional specification of effects,” PhD thesis, University of Nottingham, 2009.

[24] Pierre-Evariste Dagand, “Language support for reliable operating systems,” *Master’s thesis, ENS Cachan-Bretagne*, 2009.

[25] Butler W Lampson, “On reliable and extendable operating systems, techniques in software engineering,” in *NATO science commmittee workshop material*, 1969, vol. 2.

[26] Butler W Lampson, “Computer security in the real world,” *Computer*, vol. 37, no. 6, pp. 37–46, 2004.

[27] Richard D Greenblatt, Thomas F Knight, John T Holloway, and David A Moon, “A lisp machine,” in *Proceedings of the fifth workshop on computer architecture for non-numeric processing*, 1980, pp. 137–138.

[28] Stefan L Ram, “Dr. Alan Kay on the meaning of object-oriented programming,” 2003.

[29] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and interpretation of computer programs.* MIT Press, 1996.

[30] Martin P Ward, “Language-oriented programming,” *Software-Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[31] Matthew Flatt, “Creating languages in Racket,” *Communications of the ACM*, vol. 55, no. 1, pp. 48–56, 2012.

[32] Mike Shapiro, “Purpose-built languages,” *Communications of the ACM*, vol. 52, no. 4, pp. 36–41, 2009.

[33] John McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[34] David HD Warren, "Logic programming and compiler writing," *Software: Practice and Experience*, vol. 10, no. 2, pp. 97–125, 1980.

[35] Adele Goldberg, "Smalltalk–80: The interactive programming environment." Addison-Wesley Longman Publishing Co., Inc., 1984.

[36] Alan C Kay, "The early history of Smalltalk," in *History of programming languages—II*, 1996, pp. 511–598.

[37] Joe Armstrong, "A history of Erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 6–1.

[38] Charles H Moore, "Forth, a new way to program a mini computer," *Astronomy and Astrophysics Supplement Series*, vol. 15, p. 497, 1974.

[39] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen, "Programming languages as operating systems (or revenge of the son of the Lisp machine)," *ACM SIGPLAN Notices*, vol. 34, no. 9, pp. 138–147, 1999.

[40] Jack B Dennis and Earl C Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[41] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, nos. 5-6, pp. 1189–1205, 2013.

[42] Mark S Miller and Jonathan S Shapiro, "Paradigm regained: Abstraction mechanisms for access control," in *Annual asian computing science conference*, 2003, pp. 224–242.

[43] Mark S Miller, *Robust composition: Towards a unified approach to access control and concurrency control.* Johns Hopkins University, 2006.

[44] Fernando J Corbató and Victor A Vyssotsky, "Introduction and overview of the Multics system," in *Proceedings of the november 30–december 1, 1965, fall joint computer conference, part I*, 1965, pp. 185–196.

[45] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum, "MINIX 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[46] SA Rajunas, Norman Hardy, Allen C Bomberger, William S Frantz, and Charles R Landau, "Security in KeyKOS," in *1986 IEEE Symposium on Security and Privacy*, 1986, pp. 78–78.

[47] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, and Robert Hirschfeld, "Living in your programming environment: Towards an environment for exploratory adaptations of productivity tools," in *Proceedings of the 3rd ACM SIGPLAN international workshop on programming experience*, 2017, pp. 17–27.