The Science of Software and System Design*

Stavros Tripakis
Aalto University and University of California, Berkeley

May 11, 2017

1 Introduction

This paper has been prepared for the workshop *Foundations of Cyber-Physical Systems*, taking place on June 2nd, 2017, at KTH, Sweden. The goal of this paper is to start a discussion around the question *what is the science of system design?* In our opinion, such a science exists and is common to Cyber-Physical Systems (CPS) as to many other complex systems, in the sense described below. We will therefore speak generally of *systems* here, without focusing specifically on CPS.

Following [21, 22], we use the term *system* to denote dynamical systems in general, that is, objects that have a notion of state, a notion of dynamics (i.e., evolution of state over time), and a notion of *composition*, which allows to build larger systems from smaller and simpler components or subsystems. System composition is fundamental but often neglected in traditional system theories. On the other hand, compositionality has been extensively studied in the field of *formal methods* such as *model checking* [3, 6] and formal software development methodologies [10, 11, 8, 25, 2]. We use the term *design* not in the artistic sense, but in the engineering sense, to include the entire process from initial concept to final product.

2 System design approaches

There are basically two approaches to system design: (1) design by *trial and error*, and (2) *model-based design*. In trial-and-error, design is typically done by building prototypes, testing them (often in the field), finding errors, fixing them, and repeating the process. This approach may be feasible for relatively simple systems, but does not scale well and becomes infeasible for complex systems. By *complex*, we mean systems that are difficult to understand, to reason about, and to argue for their correct behavior. There are several factors contributing to the complexity of modern systems, from sheer size (e.g., millions of lines of code) to subtle cyber-physical interactions. A major complexity factor is the fact that modern systems fundamentally rely on *software*. Software is inherently complex: even small programs can exhibit behavior that defies the understanding of today's mathematics [22]. The larger the software, the smaller the chance to "get it right" simply by trial-and-error. This not only makes design by trial-and-error expensive and error prone. It also makes it unsafe, as most CPSs are safety-critical. Despite this, trial-and-error is prevalent today (c.f., the development of self-driving cars, and related accidents).

An alternative approach is model-based design (MBD) which proposes to use *models* rather than prototypes, at least during the design phase, but more and more also during other phases and indeed during the entire lifetime of the product [13]. Models are often cheaper to produce and test. Bugs found are typically found earlier when it is easier and cheaper to fix. Perhaps most importantly, a system model can often be subjected to *formal verification*, which provides stronger guarantees than simulation or testing. Such guarantees may include proofs of correctness with respect to given formal specifications.

*This work was partially supported by the National Science Foundation (awards #1329759 and #1139138), and by the Academy of Finland. Some of the ideas presented here have appeared previously in [21] and [22].

3 Is there a science of software and system design?

The position of this paper is that software and system development need not be just a craft or an art [12]. It can be a science. Understandably, *the science of system design* sounds so broad that one may be suspicious. After all, there are entire disciplines devoted specifically to the study of different types of systems, from mechanical, to electrical, to software engineering, and many more. Can the science of system design seriously claim to encompass all these disciplines? Our goal here is not to make grandiose claims about a "theory of everything". Nevertheless, we still believe that an overarching discipline of system design is both feasible and necessary. Such a discipline studies abstractions and common principles that are fundamental in all kinds of systems (see below). Such a discipline will also create rigorous interfaces with specific disciplines such as the ones mentioned above.

We believe that formal methods provide many of the foundations of the science of system design. Notions such as specification vs. implementation, correctness, termination, transition system, reachability, hierarchy, compositionality, interfaces, abstraction, refinement, and many more, are fundamental to most systems. Yet most engineers ignore these notions or, even if they have an intuitive understanding of them, have not been educated formally in them. Education is a crucial part of our vision. Courses in formal methods, verification, even basic logic, are typically given only at the advanced postgraduate level. What is worse, students often lack knowledge of basic mathematics (sets, functions, relations). Such courses need to be moved to the undergraduate curriculum and offered beyond the standard EECS majors.

But the task of developing (or consolidating from its fragments) the science of system design is Herculean, and will rely, in addition to education, on significant investments in research. In addition to classic and difficult topics such as computational logic, optimization, formal verification, *program synthesis* (e.g., see [1]), formal software engineering, programming languages, *hybrid systems* [18], and many more, we need to invest more effort in building bridges and interfaces, to close the gaps between the various disciplines. Efforts towards that direction include but are certainly not limited to: formal aspects of *multi-view modeling* [17, 15, 14]; compositional simulation [4, 20]; theories of contracts and interfaces [23, 7, 16, 9]; and semantics-preserving techniques to bridge high-level models and their low-level implementations [5, 19, 24].

References

[1] Rajeev Alur and Stavros Tripakis. Automatic synthesis of distributed protocols. *SIGACT News*, 48(1):55–90, 2017.

[2] R-J. Back and J. Wright. *Refinement Calculus*. Springer, 1998.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Stavros Tripakis, Michael Wetter, and Michael Masin. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the 13th ACM & IEEE International Conference on Embedded Software (EMSOFT'13)*, pages 2:1–2:12. IEEE, 2013.

[5] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-Preserving Multitask Implementation of Synchronous Programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–40, February 2008.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Cyber-physical system design contracts. In *ACM/IEEE 4th International Conference on Cyber-Physical Systems – ICCPS 2013*, pages 109–118. IEEE, April 2013.

[8] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured programming*, pages 1–82. Academic Press, London, UK, 1972.

[9] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *23rd International SPIN Symposium on Model Checking of Software (SPIN 2016)*, volume 9641 of *LNCS*, pages 38–56. Springer, April 2016.

[10] R.W. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[12] Donald E. Knuth. *The Art of Computer Programming.*

[13] Magnus Persson, Martin Törngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe, and Joachim Denil. A Characterization of Integrated Multi-View Modeling for Embedded Systems. In *Proceedings of the 13th ACM & IEEE International Conference on Embedded Software (EMSOFT'13)*, 2013.

[14] Maria Pittou and Stavros Tripakis. Checking Multi-View Consistency of Discrete Systems with respect to Periodic Sampling Abstractions. In *Formal Aspects of Component Software - The 13th International Conference (FACS 2016)*, 2016.

[15] Maria Pittou and Stavros Tripakis. Multi-View Consistency for Infinitary Regular Languages. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation – SAMOS XVI*, 2016.

[16] Viorel Preoteasa and Stavros Tripakis. Refinement Calculus of Reactive Systems. In *Proceedings of the 14th ACM & IEEE International Conference on Embedded Software (EMSOFT'14)*, pages 2:1–2:10. ACM, October 2014.

[17] Jan Reineke and Stavros Tripakis. Basic Problems in Multi-View Modeling. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2014*, volume 8413 of *LNCS*, pages 217–232. Springer, 2014.

[18] P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach.* Springer, 2009.

[19] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, October 2008.

[20] Stavros Tripakis. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation – SAMOS XV*, 2015.

[21] Stavros Tripakis. Compositional Model-Based System Design and Other Foundations for Mastering Change. *Transactions on Foundations for Mastering Change*, 1:113–129, 2016.

[22] Stavros Tripakis. Compositionality in the Science of System Design. *Proceedings of the IEEE*, 104(5):960–972, May 2016.

[23] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A Theory of Synchronous Relational Interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4), July 2011.

[24] Stavros Tripakis, Rhishikesh Limaye, Kaushik Ravindran, Guoqiang Wang, Hugo Andrade, and Arkadeb Ghosal. Tokens vs. signals: On conformance between formal models of dataflow and hardware. *Journal of Signal Processing Systems*, 85(1):23–43, October 2016.

[25] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4):221–227, 1971.