Tripakis, Stavros

The Science of Software and System Design

Published in:
IFAC-PapersOnLine

DOI:
10.1016/j.ifacol.2018.09.002

Published: 01/01/2018

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Tripakis, S. (2018). The Science of Software and System Design. *IFAC-PapersOnLine*, *51*(7), 505-507. https://doi.org/10.1016/j.ifacol.2018.09.002

Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 51-7 (2018) 505–507

The Science of Software and System Design ⋆

Stavros Tripakis *

* *Aalto University*

Abstract: Software enables impressive cyber-physical system applications, but also dominates the costs and perils of designing such systems. Is there a science of software and system design? This paper reflects on this question, and identifies opportunities for interactions with the fields of artificial intelligence and machine learning.

1. INTRODUCTION

Our work has been motivated by trying to find solutions to the following problems:

(1) How can we design better systems?
(2) How can we better design systems?

The first problem deals with creating systems that are safer, more reliable, more secure, more efficient, easier to maintain, and so on. The second problem deals with improving the design process itself: making it faster, cheaper, etc.

2. SYSTEMS AND SOFTWARE

What kind of systems are we talking about? We are interested in fundamental ideas and methods that should apply to many kinds of systems (although it is natural to also have special techniques which are applicable to specific classes of systems). The ideas presented here have been influenced primarily by our experience with safety-critical, distributed, real-time, embedded, and cyber-physical systems (CPS). A good example of a CPS is an automated intersection. For a fun video, see "RUSH HOUR" by Fernando Livschitz at `https://vimeo.com/bsfilms`.[1] Today we are not there yet. Self-driving cars are making steady progress, but are still far from being sufficiently reliable.

Systems like the above would not exist without software. Software is what makes systems of such complexity possible, even imaginable. Indeed, we claim that software is the most complex artifact humans have ever built – c.f. Tripakis (2016).

In this paper the term *software* is used broadly. Software is not just code written in typical programming languages like C or Java. A Simulink model is also software. So is a Verilog program. In that sense, the distinction between software and systems becomes blurry and often disappears. A chip designer who writes Verilog code designs software, even though the final product may be hardware.[2]

3. WHAT IS SCIENCE?

The title of this paper makes use of the pompous-sounding term *science*. The goal is not to sound arrogant. We use the term with a specific meaning: science is knowledge that helps us make predictions. Some predictions can be made without much science. Toddlers know that if they drop an object it will fall to the ground. This is an easy prediction which can be made without knowledge of physics. However, in order to predict the next appearance of a comet, or that a lander will safely reach the Moon, a non-trivial knowledge of physics is required. The strongest a science is, the more difficult the predictions it can make.

With this definition at hand, we may ask ourselves: *what is the science of system design? is there one? what predictions can we make about the systems we design and build?*

4. THE SCIENCE OF SOFTWARE

What predictions can we make about the programs we write? Can we predict that our program will terminate? that it will not crash? that it will produce the right result? in all cases? in some cases? The knowledge that allows us to answer questions such as these forms what we call *the science of software*. Although testing is an element of the science of software, it has severe limitations and is by itself not capable of making strong predictions. The reason is that testing can only cover some, but not all possible behaviors of a system. The number of behaviors in real-world systems is astronomical and for cyber-physical systems it is infinite. Such systems are effectively untestable – see Kalra and Paddock (2016).

Yet testing is the predominant approach to designing software, and systems in general. In the case of CPS like self-driving cars, this approach can be termed *design by trial-and-error* – c.f. Tripakis (2016, 2018). Moreover, because the industry is still unregulated, the human drivers currently play the role of the testers. According to `https://arstechnica.com/cars/2015/11/`

⋆ Some of the ideas in this paper appear also in Tripakis (2016, 2018).

[1] The author would like to thank Christos Cassandras for pointing out this video.

[2] One might object that in the case of programs written in say C, the code is running *on the final system* (e.g., car) whereas Simulink and Verilog models are not really part of the final product. However, embedded code generators exist for Simulink that generate production code, and Verilog programs are transformed into circuits via logic synthesis. Also, even in the case of C, what is actually running on the car is not the C program itself, but the machine code generated from it by a compiler.

2405-8963 © 2018, IFAC (International Federation of Automatic Control) Hosting by Elsevier Ltd. All rights reserved.
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2018.09.002

tesla-will-restrict-self-driving-autopilot-mode-to-stop-people-doing-crazy-things/, the Tesla CEO stated in November 2015 in reference to errors of the *Autopilot* system in Tesla cars: *It was described as a beta release. The system will learn over time and get better and that's exactly what it's doing. It will start to feel quite refined within a couple of months.* The first human casualty due to a failure of the Tesla autopilot was reported a few months later, in May 2016. Other accidents, including fatal ones, have occurred since then (the latest at the time we started writing this paper, in March 2018). These accidents raise the question: *are the drivers supposed to debug the autopilot?*

Things don't have to be this way. A rich body of knowledge exists that allows to make stronger predictions about software than testing allows. This science is known under the names of *formal verification* or *formal methods* – see Hoare (1969); Clarke et al. (2000); Baier and Katoen (2008). The basic principle of formal verification is to try to *prove* the absence of errors under all possible scenarios, rather than testing the system in only a few scenarios. Formal verification is applicable not just to (traditional) software, but to systems in general. Formal verification is the main component in the science of software and system design – c.f. Tripakis (2016), and formal methods are becoming increasingly important in the industry – see Newcombe et al. (2015).

Formal methods are often difficult and expensive to use, for a number of reasons. First, they require skills that programmers or designers often lack (e.g., knowledge of logic). Second, formal verification is not a simple "push-button" process, especially for real-world systems. Considerable "manual" effort is required to make formal techniques scalable to large systems, e.g., by building suitable modeling abstractions – see Newcombe et al. (2015). Another way to cope with scalability problems is *compositionality* – see Tripakis (2016). In our research group, we have been developing the *Refinement Calculus of Reactive Systems*, a compositional formal modeling and reasoning framework for CPS – see Preoteasa et al. (2017); Dragomir et al. (2018).

5. INTERACTIONS BETWEEN THE SCIENCE OF SOFTWARE AND THE SCIENCE OF LEARNING

Formal methods are the science of so-called *model-based design* – c.f. Tripakis (2016). In order to verify a system, one needs a formal model of the system, and a formal specification (which specifies correctness or other desirable system properties). Model-based design has been constantly gaining ground in the industry, but still faces important problems. For instance, a key problem is how to construct reliable models, especially of the physical world. Techniques from the booming fields of artificial intelligence and machine learning can be leveraged to come up with new solutions to this and other problems, revolutionizing system design – c.f. Tripakis (2018).

Conversely, system design can, and must, aid artificial intelligence. To quote from https://medium.com/@mijordan3/artificial-intelligence-the-revolution-hasnt-happened-yet-5e1d5812e1e7 : "many of our early societal-scale inference-and-decision-making systems are already exposing serious conceptual flaws" and "we are missing an engineering discipline with its principles of analysis and design." The formal methods community can help, e.g., by developing verification methods for machine learning systems (c.f. the FoMLAS Workshop at ETAPS 2018).

6. SYNTHESIS OF DISTRIBUTED PROTOCOLS FROM SCENARIOS AND REQUIREMENTS

In this section we give an example of a fruitful interaction between learning and system design.

Verification requires a formal model of the system to be verified, say S, and a formal specification, say ϕ. Verification then consists in checking that S satisfies ϕ. *Synthesis* is the problem of automatically generating a system S from a specification ϕ, so that S satisfies ϕ by construction – see Pnueli and Rosner (1989); Ramadge and Wonham (1989). Synthesis is a hard problem, and when there are more than one systems (e.g., controllers) to be synthesized, the problem is generally undecidable – see Pnueli and Rosner (1990); Thistle (2005); Tripakis (2004).

One way to avoid undecidability and obtain a practical approach to synthesis of distributed protocols is to give as input to the synthesis tool, in addition to the formal specification ϕ, a set of example *scenarios*. We call this approach *synthesis from scenarios and requirements* – see Alur et al. (2014); Alur and Tripakis (2017). The example scenarios specify how the system should behave in *some* (but not all) cases. [3] Then, the problem becomes one of *generalizing*: finding out how the system should behave in *all* possible cases, not just those specified by the examples. Generalization is always present in learning, although it is typically difficult to define what is the right way to generalize. In our case, we are fortunate: the right way to generalize is well-defined: it is defined by the formal specification ϕ. In other words, any generalization which satisfies ϕ is a valid generalization. Moreover, in the case of finite-state systems, the search of all possible generalizations is finite, and can be searched effectively, turning an undecidable problem into a decidable one. See Alur et al. (2014); Alur and Tripakis (2017) for details.

7. CONCLUSION

System design has evolved from a methodology based on trial-and-error to more rigorous, model-based methodologies, where formal modeling and verification play key roles. The next step in this evolution is to incorporate data-driven methods such as learning – see Tripakis (2018). An example in this direction is the synthesis of distributed protocols from scenarios and requirements.

REFERENCES

Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., and Udupa, A. (2014). Synthesizing Finite-state Protocols from Scenarios and Requirements. In *Haifa Verification Conference*, volume 8855 of *LNCS*. Springer.

[3] Typically we want to keep the number of scenarios small, for ease of design. In the examples we tried, a very small number of scenarios (1-10) is sufficient – see Alur et al. (2014); Alur and Tripakis (2017).

Alur, R. and Tripakis, S. (2017). Automatic synthesis of distributed protocols. *SIGACT News*, 48(1), 55–90.

Baier, C. and Katoen, J.P. (2008). *Principles of Model Checking*. MIT Press.

Clarke, E., Grumberg, O., and Peled, D. (2000). *Model Checking*. MIT Press.

Dragomir, I., Preoteasa, V., and Tripakis, S. (2018). The Refinement Calculus of Reactive Systems Toolset. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*, volume 10806 of *LNCS*. Distinguished Artifact Award.

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

Kalra, N. and Paddock, S.M. (2016). Driving to safety – how many miles of driving would it take to demonstrate autonomous vehicle reliability? Technical report, RAND Research Report RR-1478-RC.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4), 66–73.

Pnueli, A. and Rosner, R. (1990). Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, 746–757.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *ACM Symp. POPL*.

Preoteasa, V., Dragomir, I., and Tripakis, S. (2017). The Refinement Calculus of Reactive Systems. *CoRR*, abs/1710.03979.

Ramadge, P. and Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE*.

Thistle, J.G. (2005). Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5), 503–509.

Tripakis, S. (2004). Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1), 21–28.

Tripakis, S. (2016). Compositionality in the Science of System Design. *Proceedings of the IEEE*, 104(5), 960–972.

Tripakis, S. (2018). Data-driven and model-based design. In *1st IEEE International Conference on Industrial Cyber-Physical Systems (ICPS 2018)*.