Tripakis, Stavros

The Science of Software and System Design

Published in:
IFAC-PapersOnLine

DOI:

Published: 01/01/2018

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

Please cite the original version:
The Science of Software and System Design

Stavros Tripakis

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.

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.

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/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.

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

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

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...
Formal methods are the science of so-called model-based societal-scale inference-and-decision-making systems. They haven't happened yet: “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 \( \phi \). Verification then consists in checking that \( S \) satisfies \( \phi \). Synthesis is the problem of automatically generating a system \( S \) from a specification \( \phi \), so that \( S \) satisfies \( \phi \) 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 \( \phi \), 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. 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 \( \phi \). In other words, any generalization which satisfies \( \phi \) 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 datadriven methods such as learning – see Tripakis (2018). An example in this direction is the synthesis of distributed protocols from scenarios and requirements.

REFERENCES


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).


