Data-Driven and Model-Based Design

Stavros Tripakis
Aalto University and University of California, Berkeley

***Abstract*—This paper explores novel research directions arising from the revolutions in artificial intelligence and the related fields of machine learning, data science, etc. We identify opportunities for system design to leverage the advances in these disciplines, as well as to identify and study new problems. Specifically, we propose *Data-driven and Model-based Design* (DMD) as a new system design paradigm, which combines model-based design with classic and novel techniques to learn models from data.**

***Index Terms*—System design, formal methods, machine learning, verification, synthesis**

I. Introduction

A. Artificial Intelligence

Computers and an abundance of data from all sorts of sources are revolutionizing many fields of science and technology, from biology and medicine, to astronomy and the social sciences. The so-called (new) *artificial intelligence* (AI) revolution is also changing our industry, society, and everyday lives. We use the term AI in a broad sense, to include many related fields and buzzwords, such as *machine learning*, *big data*, *data science*, and so on. In fact, it is reasonable to argue that after a long "winter", AI is once again blooming thanks to the recent advances in machine learning and data science.

B. System Design

The field of *system design* is equally broad, and includes classic (continuous) and modern (discrete, hybrid, and cyber-physical) system theory, modeling, simulation, testing, formal verification, scheduling, and other topics [1], [10], [33], [34], [40], [43], [49], [53], [57], [68], [69], [73].

C. Two Interesting Questions

The ongoing AI revolution is raising the following questions, in terms of the two fields discussed above:

- *Can AI benefit from system design, and how?*
- *Can system design benefit from AI, and how?*

We believe that the answer to both questions is yes, and outline some ideas supporting our claim in Sections II and III, respectively. Section III also identifies DMD as a fundamentally new approach to system design. Key elements of DMD are presented in Section IV. Section V concludes the paper.

II. System Design for AI

As AI systems become parts of our daily routine (e.g., in self-driving cars) we need to trust those systems with our lives. AI systems of today rely on so-called *learning-enabled components* (LECs) [48]. For example, a self-driving car may rely on a LEC performing visual recognition. In order to trust the AI system, we must also trust its LECs. However, today's machine learning systems are highly unpredictable [66]. Prototyping such systems may seem quick and easy, but it incurs a high cost, called *technical debt* [58]. These problems are not theoretical. Numerous traffic violations, accidents, and even human casualties have resulted as the fault of LECs. It is therefore correctly and widely recognized that *AI systems are not dependable* [23].

How to achieve dependability of LECs? "Manual" reasoning about these systems is prohibitive. Systems like neural networks become incomprehensible as their size grows [59]. Moreover, using a *trial-and-error* approach (testing) does not scale [38].

We argue that we need rigorous, *formal* modeling and verification techniques, in order to

1) *model* AI systems, and in particular their LECs;
2) *specify* what properties these systems have, or should have;
3) reason about such properties and ultimately *verify* that they are satisfied.

Indeed, these research areas are currently emerging, see for instance [36], [39], [54], [59]. Although the problems outlined above are far from being solved, it is our belief that they will become key problems in the near future, and will attract significant interest from researchers, as well as major investments from funding agencies. We also believe that the field of formal methods has a lot to contribute in this line of research, although it is likely that existing methods may fall short in being able to cope with LECs in the near term. This will create opportunities for the discovery of novel methods. We do not discuss these issues further here, as our focus in this paper is on the second question posed above, namely, how can system design benefit from AI. This is discussed next.

III. AI for System Design

Can we leverage the advances in AI in order to improve, and perhaps even revolutionize the way we design systems? How can we use the methods of AI, machine learning, data science, etc., in order to advance the methodology and science of system design?

A. Traditional Approaches to System Design: Trial-and-error and Model-based Design

Traditionally, there have been essentially two approaches to system design [73]: (1) design by *trial-and-error*, and (2) *model-based design* (MBD).

Design by trial-and-error essentially proceeds as follows: build a prototype system; test it; find bugs; fix them; repeat.

The hope is that progressively fewer and fewer bugs are found. The process stops when no bugs are found, or the deadline to release the project is reached, or some other criterion, technical- or business-driven. In the age of software, design by trial-and-error does not scale: as software becomes more and more complex, the number of tests needed to achieve even rudimentary coverage becomes astronomical. Design by trial-and-error is also unsafe. In the age of cyber-physical systems, which closely interact with humans in safety-critical ways, trial-and-error is not a good option.

Model-based design [49], [67] improves trial-and-error by building *models* instead of system *prototypes*. Models are sometimes called *virtual systems* or *virtual prototypes*. Using models instead of prototypes has several advantages:

- Models are safer than prototypes: a self-driving car model can cause no casualties.
- Models are cheaper and faster to build, cheaper and faster to modify/repair, cheaper and faster to maintain, etc.
- Models are cheaper and (sometimes) faster to simulate. Simulating a model is analogous to testing a prototype. Using parallelism and other techniques, it is possible to run many more simulations in a model than tests in a real prototype.
- Beyond simulation, models are amenable to more rigorous, formal, and exhaustive types of analysis, such as static analysis and formal verification. In addition to finding bugs, these analyses, if they succeed, can prove the *absence* of bugs, which can never be done by testing a prototype [19].

An obvious challenge in the MBD paradigm is how to make the *proof of correctness* promise of formal methods a reality in practice. Steady advances in the field keep bringing us closer to this goal (see [73] and references therein for a more extended discussion on this topic). But in addition to the formal verification challenge, using models raises several other serious concerns:

1) Where do the models come from?
2) How does one go from a model to a real system?

In fact, the second question has been easier to address. The success that MBD enjoys in the industry is very much due to the ability to address this question. Indeed, this has been achieved by building various types of *code generators* which take as input models and generate automatically implementations in various forms (e.g., C software code, HDL or FPGA hardware code, etc.). Automatically generating implementations from models is by no means an easy problem. It is analogous to the problem solved by *compilers*, which generate machine code (assembly) from a high-level programming language such as C or Java. Moreover, in the case of real-time, embedded, and safety-critical systems, code generators face new challenges, such as:

- How to meet real-time, memory, energy, and other resource constraints?
- How to ensure that safety and other properties established at the model level carry over to the real system?

Many years of research have been devoted to addressing these questions, in the related fields of (real-time) scheduling [10], [11], [31], [65], semantics-preserving code generation [12], [13], [15], [30], [32], [63], [72], and system design in general [43], [49], so that the research problem can be today considered more or less solved.

B. MBD as System Programming

The question *where do the models come from*, on the other hand, has not been adequately addressed. Here, it is useful to make a distinction between two types of models: *system models*, and *environment models*. The former are models of the system to be built, denoted S. The latter are models of the *environment* which S is supposed to operate in (typically in a closed-loop configuration). Thinking of MBD as *system programming*, and of verification and code generation tools as parts of *system compilers*, it is natural to also think of system models as *system programs*. Just like standard programmers need to write programs in C or Java, it is natural to expect system programmers to write the system models. The difference, however, is that standard programmers do not typically need to write extra code for the environment in which their programs operate: typically, this environment is a set of other programs which are interfaced with the program under development. In the case of embedded and cyber-physical systems, the situation is different. Here, the environment is the physical world, including humans (pilots, drivers, pedestrians, patients, nurses, etc.). It is too complicated and difficult to capture the environment in a detailed and accurate manner. Thus, building good environment models, that strike a good balance between tradeoffs such as precision, size, and complexity, is an art. Most companies spend considerable resources to build their environment models (e.g., engine models in the automotive industry) and cherish them as their most important intellectual property (of higher importance than their controllers).

In the era of machine learning, an obvious idea is: instead of building environment (and perhaps other kinds of) models "by hand," can we *learn* such models automatically? This is one of several questions that motivate the novel system design paradigm that we propose in this paper, discussed next.

C. Data-driven and Model-based Design (DMD)

This paper advocates a third and fundamentally new approach to system design, which we call *Data-driven and Model-based Design* (DMD). DMD can be seen as a *hybrid* approach which seeks

- to combine the best of both worlds, namely trial-and-error and model-based design;
- to leverage the advances of AI, in particular, in machine learning and data science; and
- to complement existing AI methods with novel machine learning and synthesis techniques, developed specifically for system design.

In the sequel, we elaborate on the above points, and present a list of the elements of DMD. This list is by no means claimed to be exhaustive, and is necessarily partial.

IV. Elements of DMD

A. *Formal Modeling and Verification*

DMD extends MBD. Therefore, all elements of MBD are also elements of DMD. In particular, formal modeling and verification [7], [16] are the most important elements of MBD, as argued in [73]. We will not repeat this discussion here. Suffice it to say that we consider formal modeling and verification, and in general formal methods, i.e., mathematically rigorous design methods, a key element also of DMD. The goal is to *enhance* those methods with data-driven techniques, *without* sacrificing mathematical rigor, so that we get the benefits of both worlds as mentioned above.

B. *Machine Learning, Model Learning, System Identification*

The crucial new element that DMD adds to MBD is *learning models from data*. As of today, learning models from data is not a single, cohesive, discipline. It is a broad set of techniques, fragmented and spread over many communities and disciplines, from machine learning and data mining to theoretical computer science and even to more traditional fields like control theory.

There are many variants of the *learning models from data* problem, because there are many practical instances where solving such problems can be extremely useful. The different variants arise from variations along two primary dimensions: (1) what kind of data is provided as input, and (2) what type of model is expected as output. (A third but less important dimension is what kind of method is used to do the learning.)

In control theory, the area of *system identification* is concerned with "building mathematical models of dynamical systems based on observed data from the systems" [44]. In this context, "models of dynamical systems" is taken to mean the kinds of models usually studied in signals, systems and control theory, such as differential or difference equations [50].

But dynamical systems can also be modeled differently. In computer science, which for the most part studies discrete systems, dynamical systems are typically modeled using automata, state machines and transition systems [7], [16], [41]. Recently, Vaandrager used the term *model learning* to describe the problem of building different types of "state diagram models of software and hardware systems by providing inputs and observing outputs" [75]. Model learning has a long history that goes back to early works on *passive* learning (only from examples) and *active* learning (with the aid of a *teacher*) of finite automata and other models of formal languages [6], [18], [27]. These techniques have been extended over the years to models such as input/output state machines (Moore and Mealy) [26], [61], as well as symbolic or extended machines with variables and other types of data [9], [14], [20], [35], [37]. The field of model learning is closely related to the fields of testing [42] and of *synthesis* (discussed further below), as well as to optimization and games [77].

Machine learning is a vast field with the ambitious ultimate goal of understanding "how to program computers to learn – to improve automatically with experience" [47]. As such, machine learning has close ties with artificial intelligence, and may also help us understand the mechanisms of human learning. As stated in [47], machine learning is inherently multidisciplinary and draws results from many disciplines, from statistics to neuroscience.

Machine learning studies several instances of the generic problem of *learning models from data*. Examples include: learning classifiers or decision trees, training (i.e., learning the weights of) a neural network, deriving confidence intervals, learning probabilities (Bayesian inference), and so on. It is interesting to note that much of the machine learning corpus of knowledge focuses on learning *stateless* models such as classifiers or decision trees. This is contrary to model learning which focuses on learning models with state such as automata and state machines. However, there exist subareas of machine learning which focus on stateful models, such as reinforcement learning and recurrent neural networks.

All this multidisciplinary panoply of somewhat heterogeneous learning techniques forms an important part of what we call DMD. In a system design context, several of these learning methods may be useful, and even necessary, in a single project. For example, system identification may be necessary to identify the parameters of a continuous-time plant model, in an embedded-control system. Model learning techniques may be used to build state machine models of the behavior of an (human) operator. If the embedded control system is part of, say, a robot, reinforcement learning may be used to learn the strategy to accomplish certain missions. Statistical methods may be used to predict probabilities of failure, maintenance events, etc. Even specifications of what the system is supposed to be doing can be "mined" from data [5], [45].

In addition to providing possible answers to questions such as *where do models (and specifications) come from?*, learning techniques can be useful in improving the MBD processes that follow even after models become available. For example, learning techniques can be used in verification, to automatically synthesize *invariants* or other conditions from examples or counter-examples [25], [62].

C. *Synthesis*

As mentioned above, learning is related to the area of *synthesis*, which is generally concerned with automatically generating, given a formal specification ϕ, a program that satisfies ϕ *by construction* [22], [28]. (We use the term *synthesis* broadly, to encompass the somewhat disparate fields of deductive program synthesis [46], reactive synthesis [22], [51], controller synthesis [21], [55], and modern program synthesis [28].) As with learning, there are many variants of the synthesis problem, depending on the type of specifications used as input, and the type of programs expected as output.

Synthesis is a key element of MBD, and also of DMD. However, synthesis has scalability limitations which have so far made it difficult to apply in practice. These limitations are both methodological in nature (it is difficult to write complete formal specifications of real-world systems), and algorithmic (many synthesis problems have a prohibitive computational complexity or are even undecidable [52], [70], [71]). In what follows, we propose a way of remedying these concerns by mixing synthesis and learning into a powerful combination.

D. Synthesis from Examples and Requirements

Before presenting the combined method, let us summarize the synthesis and learning problems, in semi-formal notation. This will reveal their similarities but also some important differences, and lead to a new proposal that we present below.

Problem 1 (Synthesis): Given specification ϕ, synthesize system S such that S satisfies ϕ, denoted $S \models \phi$.

We will not define formally what ϕ and S are, neither the satisfaction relation $\models$. As we said above, there are many variants of the problem. We refer the reader to the literature.

Problem 2 (Learning): Given set of examples E, synthesize system S such that S is *consistent with* E and *generalizes well*.

Let us explain the terms *consistent with* and *generalizes well* via an example. Consider an image classifier S which has been trained to distinguish between, say, bicyclists and pedestrians. The set of examples E contains the training images and their correct classification. After the classifier is trained, we expect it to work correctly, at the very least, on the examples it has been trained with. That is, given an image in E, S must classify it correctly. This is what we mean by S being *consistent* with E.

But in reality, we expect S to do much more than that. Not only should it work correctly on the images it has been trained with, it should also work well on images it has never encountered before. That is, S should *generalize well*. It is important to note that what exactly *well* means depends on the application, and is often difficult if not impossible to formalize. For instance, *well* could mean *as well as a typical adult human*, or *as well as an experienced driver*, or *at least as well as the previous version of the self-driving car*. One of the difficulties in formalizing learning problems comes from the difficulty of formalizing generalization. In some cases, principles such as *Occam's razor* are used. For example, in automata learning, we might require not just any automaton which is consistent with the examples, but also a *minimal* automaton (in terms of number of states). Problem 3 proposed next offers, among other benefits, an alternative definition of generalization, with several benefits.

Problem 3 (Synthesis from Examples and Specification): Given set of examples E and specification ϕ, synthesize system S such that: (1) S is *consistent with* E, and (2) $S \models \phi$.

We call Problem 3 the *synthesis from examples and specification* paradigm (SES). SES is powerful in many respects:

(1) The synthesizer is provided with more information than in both the synthesis and learning problems. It is provided with both a set of examples and a specification. Having both pieces of information opens up many possibilities for better and more efficienly computed solutions.

(2) In particular, the examples can be sometimes used to bootstrap the synthesizer with an easily produced albeit *incomplete* solution S_0. S_0 may be easy to construct to be consistent with the examples, although it might not satisfy the entire specification. Then, the problem becomes one of *completing* S_0 while taking care to satisfy the specification. This *completion problem* is often easier to tackle than the synthesis problem [4].

(3) The specification ϕ can be seen as a *generalization boundary*. The possible completions of the initial solution S_0 can be seen as all possible generalizations of systems consistent with the examples E. Among all these generalizations, which ones are *good* generalizations? SES offers a natural, and also formal, answer to this question: a good generalization is a completion that satisfies ϕ.

(4) Having a formal specification ϕ also paves the way to making AI systems more dependable. For instance, ϕ might be used to encode certain safety properties. Doing so, we require the synthesizer to give us a solution which is not only consistent with the examples and generalizes well, but is also safe (since it must satisfy ϕ).

Our previous work on synthesizing distributed protocols [2]–[4] can be seen as a concrete instance of the SES paradigm, which inspired the generic formulation of Problem 3. The distributed protocol synthesis problem from specifications only is generally undecidable [52], [70], [71]. SES turns this undecidable synthesis problem into a decidable one by providing to the synthesizer with a set of example *scenarios* E (in addition to ϕ). In our case, E was represented as a set of *message sequence charts*, but the precise choice of representation is not that important. Our experiments showed that the distributed protocol SES is not only decidable in theory, but also tractable in practice for several interesting protocols [2]–[4].

In addition to being algorithmically tractable, SES also alleviates the methodological problems of synthesis mentioned earlier. In SES, the specification ϕ does not have to be a complete specification of the system. It simply has to constrain the set of possible generalizations. For example, ϕ might state that any generalization is acceptable as long as it is deadlock-free. This is much easier to write, compared to a full specification required for synthesis without examples.

Other instances and variations of the SES problem have been studied in several recent works, including for instance in the contexts of FSM identification [74], and path planning [76].

The SES formulation draws inspiration from our own work as explained above. But it is also related to other recent ideas that emerged in the field of synthesis. One is *counter-example guided inductive synthesis* (CEGIS) [64]. CEGIS is based on reducing the synthesis problem to a combination of search and verification. A search is done over possible candidate solutions, and each candidate is checked for correctness using a verifier (e.g., a model-checker [7], [16]). If the candidate is correct, a solution is found. If not, the verifier returns a *counterexample*, which can be used to prune the search space. SES is also related to *sciduction* [60] which proposes to provide to the synthesizer additional information about the system structure. In our case, we assume that such information is "hardwired" into the problem (the space of systems S is fixed by definition of the problem).

Similar ideas have appeared also in the field of machine learning. [47] distinguishes *inductive* learning methods (from examples) and *analytical* learning methods, where the learner is provided, in addition to examples, with a *domain theory* B representing "background knowledge". For example, in developing a program that learns to play chess, B may encode the rules of chess. B can be viewed as the specification ϕ in our formulation of SES, although the completion and CEGIS based algorithms are different.

V. Conclusion

Science is knowledge that can help us make predictions. The stronger the science, the stronger the predictions it allows to make. In [73] we argued for a science of system design based on MBD and formal verification, which allows to make stronger predictions than trial-and-error methods based on simulation and testing. In this paper, we argue for a further development towards DMD, which extends MBD with techniques and tools for learning models from data. DMD attempts to exploit the *big data* revolution and use information contained in data from all types of sources, including data from legacy systems, already deployed systems, prototypes, or even models and simulations, as well as software repositories [29], [56]. DMD also seeks to exploit the recent advances in data science and machine learning, but also other types of learning, such as model learning. DMD seeks to do all this while at the same time preserving the rigour and guarantees of MBD, by relying on solid mathematical foundations.

It is not the ambition of this paper to give a complete roadmap of DMD. We expect DMD to form into a dynamically evolving research area. At the same time, we identified a set of elements which we believe are crucial for DMD, in particular, the combination of synthesis and learning into the SES approach, which already appears innovative and promising.

DMD provides only one possible answer to the question *how can system design benefit from the AI/machine learning revolution?* There are many other ways to leverage learning techniques for system design. For instance, [17] uses machine learning techniques to improve nonlinear constraint solving.

The question *how can AI benefit from system design?* is also of great interest. As we discussed, today's AI systems are unsafe, insecure, and unreliable in general. Remedying this requires advances in formal modeling and verification of learning-enabled systems, a next challenge for the field of formal methods.

Acknowledgments

I would like to thank my collaborators in the iCyPhy project (2012-2015) [24] and the participants of the Dagstuhl seminar *Formal Synthesis of Cyber-Physical Systems* [8] who helped shape the thoughts expressed in this paper.

References

[1] R. Alur, *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[2] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing Finite-state Protocols from Scenarios and Requirements," in *Haifa Verification Conference*, ser. LNCS, vol. 8855. Springer, 2014.

[3] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Automatic Completion of Distributed Protocols with Symmetry," in *27th International Conference on Computer Aided Verification (CAV)*, ser. LNCS, vol. 9207. Springer, 2015, pp. 395–412.

[4] R. Alur and S. Tripakis, "Automatic synthesis of distributed protocols," *SIGACT News*, vol. 48, no. 1, pp. 55–90, 2017.

[5] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'02*. ACM, 2002, pp. 4–16.

[6] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

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

[8] C. A. Belta, R. Majumdar, M. Zamani, and M. Rungger, "Formal Synthesis of Cyber-Physical Systems (Dagstuhl Seminar 17201)," *Dagstuhl Reports*, vol. 7, no. 5, pp. 84–96, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/8281

[9] M. Botincan and D. Babic, "Sigma*: symbolic learning of input-output specifications," in *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13*, 2013, pp. 443–456.

[10] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, 2001.

[11] G. Buttazzo, *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications," in *Proceedings of the 2003 ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, Jun. 2003, pp. 153–162.

[13] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-Preserving Multitask Implementation of Synchronous Programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, pp. 1–40, Feb. 2008.

[14] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Learning Extended Finite State Machines," in *SEFM*, 2014, pp. 250–264.

[15] S. Chattopadhyay, A. Roychoudhury, J. Rosen, P. Eles, and Z. Peng, "Time-Predictable Embedded Software on Multi-Core Platforms: Analysis and Optimization," *Foundations and Trends in Electronic Design Automation*, vol. 8, no. 3-4, pp. 199–356, 2014.

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

[17] S. Dathathri, N. Arechiga, S. Gao, and R. M. Murray, "Learning-based abstractions for nonlinear constraint solving," in *IJCAI-17*, 2017, pp. 592–599.

[18] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. CUP, 2010.

[19] E. W. Dijkstra, "The Humble Programmer," ACM Turing Lecture 1972. [Online]. Available: https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

[20] S. Drews and L. D'Antoni, "Learning symbolic automata," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, ser. LNCS, vol. 10205, 2017, pp. 173–189.

[21] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi, "Supervisory Control and Reactive Synthesis: A Comparative Introduction," *Discrete Event Dynamic Systems*, vol. 27, no. 2, pp. 209–260, 2017.

[22] B. Finkbeiner, "Synthesis of reactive systems," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, J. Esparza, O. Grumberg, and S. Sickert, Eds., vol. 45. IOS Press, 2016, pp. 72–98.

[23] Research Challenge II: Dependability, Finnish Center for Artificial Intelligence. [Online]. Available: http://fcai.fi/research/

[24] A. Fisher, C. A. Jacobson, E. A. Lee, R. M. Murray, A. Sangiovanni-Vincentelli, and E. Scholte, "Industrial cyber-physical systems – icyphy," in *Complex Systems Design & Management*, M. Aiguier, F. Boulanger, D. Krob, and C. Marchal, Eds., 2014, pp. 21–37.

[25] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A Robust Framework for Learning Invariants," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Springer, 2014, pp. 69–87.

[26] G. Giantamidis and S. Tripakis, "Learning Moore Machines from Input-Output Traces," in *21st International Symposium on Formal Methods (FM 2016)*, ser. Lecture Notes in Computer Science, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, Eds., vol. 9995, 2016, pp. 291–309.

[27] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[28] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[29] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *2015 ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA. ACM, 2015, pp. 416–432.

[30] G. Han, M. D. Natale, H. Zeng, X. Liu, and W. Dou, "Optimizing the implementation of real-time Simulink models onto distributed automotive architectures," *Journal of Systems Architecture - Embedded Systems Design*, vol. 59, no. 10-D, pp. 1115–1127, 2013.

[31] M. Harbour, M. Klein, R. Obenza, B. Pollak, and T. Ralya, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer, 1993.

[32] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree, "From control models to real-time code using Giotto," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 50–64, 2003.

[33] T. Henzinger and J. Sifakis, "The discipline of embedded systems design," *IEEE Computer*, vol. 40, no. 10, pp. 32–40, 2007.

[34] T. A. Henzinger, "Two challenges in embedded systems design: predictability and robustness," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 366, no. 1881, pp. 3727–3736, 2008.

[35] F. Howar, B. Steffen, B. Jonsson, and S. Cassel, "Inferring Canonical Register Automata," in *VMCAI*, 2012, pp. 251–266.

[36] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," *CoRR*, vol. abs/1610.06940, 2016. [Online]. Available: http://arxiv.org/abs/1610.06940

[37] B. Jonsson, "Learning of Automata Models Extended with Data," in *SFM 2011, Advanced Lectures*, 2011, pp. 327–349.

[38] N. Kalra and S. M. Paddock, "Driving to safety – how many miles of driving would it take to demonstrate autonomous vehicle reliability?" RAND Research Report RR-1478-RC, Tech. Rep., 2016. [Online]. Available: https://www.rand.org/pubs/research_reports/RR1478.html

[39] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," *CoRR*, vol. abs/1702.01135, 2017. [Online]. Available: http://arxiv.org/abs/1702.01135

[40] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transact ions on*, vol. 19, no. 12, pp. 1523–1543, Dec 2000.

[41] Z. Kohavi, *Switching and finite automata theory*, 2nd ed. McGraw-Hill, 1978.

[42] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - A survey," *Proceedings of the IEEE*, vol. 84, pp. 1090–1126, 1996.

[43] I. Lee, J. Leung, and S. Son, Eds., *Handbook of Real-Time and Embedded Systems*. Chapman & Hall, 2007.

[44] L. Ljung, *System Identification: Theory for the User*, 2nd ed. Prentice Hall, 1999.

[45] D. Lo, S.-C. Khoo, J. Han, and C. Liu, *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011.

[46] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90–121, Jan. 1980.

[47] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[48] S. Neema, "Assured autonomy," DARPA Research Program. [Online]. Available: https://www.darpa.mil/program/assured-autonomy

[49] G. Nicolescu and P. J. Mosterman, Eds., *Model-Based Design for Embedded Systems*. CRC Press, 2010.

[50] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals &Amp; Systems*, 2nd ed. Prentice-Hall, 1996.

[51] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *ACM Symp. POPL*, 1989.

[52] ——, "Distributed reactive systems are hard to synthesize," in *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, 1990, pp. 746–757.

[53] R. Poovendran, K. Sampigethaya, S. K. S. Gupta, I. Lee, K. V. Prasad, D. Corman, and J. Paunicka, "Special Issue on Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 6–12, Jan. 2012.

[54] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *Computer Aided Verification: CAV 2010*, T. Touili, B. Cook, and P. Jackson, Eds., 2010, pp. 243–257.

[55] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, Jan. 1989.

[56] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'15*. ACM, 2015, pp. 111–124.

[57] A. Sangiovanni-Vincentelli, "Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[58] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. MIT Press, 2015, pp. 2503–2511.

[59] D. Selsam, P. Liang, and D. L. Dill, "Developing bug-free machine learning systems with formal mathematics," in *34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 2017, pp. 3047–3056.

[60] S. A. Seshia, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *Proceedings of the Design Automation Conference (DAC)*, June 2012, pp. 356–365.

[61] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *FM 2009*, 2009, pp. 207–222.

[62] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as learning geometric concepts," in *Static Analysis*, F. Logozzo and M. Fähndrich, Eds. Springer, 2013, pp. 388–411.

[63] J. Sifakis, S. Tripakis, and S. Yovine, "Building Models of Real-Time Systems from Application Software," *Proceedings of the IEEE, Special issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 100–111, Jan. 2003.

[64] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California at Berkeley, 2008.

[65] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[66] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *CoRR*, vol. abs/1312.6199, 2013. [Online]. Available: http://arxiv.org/abs/1312.6199

[67] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, no. 4, pp. 110 –111, apr 1997.

[68] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang, "Toward a science of cyber-physical system integration," *IEEE Proc.*, vol. 100, no. 1, pp. 29–44, Jan. 2012.

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

[70] J. Thistle, "Undecidability in decentralized supervision." *Systems & Control Letters*, vol. 54, no. 5, pp. 503–509, 2005.

[71] S. Tripakis, "Undecidable Problems of Decentralized Observation and Control on Regular Languages," *Information Processing Letters*, vol. 90, no. 1, pp. 21–28, Apr. 2004.

[72] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale, "Implementing Synchronous Models on Loosely Time-Triggered Architectures," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008.

[73] S. Tripakis, "Compositionality in the Science of System Design," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 960–972, May 2016.

[74] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *STTT*, vol. 20, no. 1, pp. 35–55, 2018.

[75] F. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017.

[76] M. Wen, I. Papusha, and U. Topcu, "Learning from demonstrations with high-level side information," in *IJCAI*, 2017, pp. 3055–3061.

[77] M. Yannakakis, "Testing, optimization, and games," Invited talk at ICALP-LICS'04.