INVITED PAPER

Compositionality in the Science of System Design

The advent of CPSs has urged researchers to rethink systems and system design. This paper presents some challenges in the science of system design, expanding on the key principle of compositionality.

By Stavros Tripakis

ABSTRACT | Is there a science of system design? Just like any other design activity, system design is partly an art. However, mathematical theories and computer automation can help, and are even essential for designing complex systems reliably and economically. Until today, the plethora of different types of systems has resulted in a fragmented space of theories and tools. The advent of cyber-physical systems, which are by definition multidisciplinary, has urged researchers to rethink systems and system design, with model-based methods gaining acceptance. This paper describes some of the challenges in the domain, expanding on the key principle of compositionality.

KEYWORDS | Compositionality; computer-aided design; cyber-physical systems (CPSs); formal verification; interfaces

I. INTRODUCTION

The advent of cyber–physical systems (CPSs) has exposed the limitations of current system design theory and practice, and has revealed many new and exciting challenges. In a way, the challenges facing the design and construction of CPSs are not very different from those facing the design and construction of safety-critical systems, real-time systems, and embedded systems [1] and [2], since the boundaries between such systems and CPSs are blurry. On the other hand, CPSs can be seen as the next generation in the history of these systems, with increased size and complexity. This, and the fact that CPSs are becoming widespread in our modern societies, makes the need for overcoming the challenges pressing.

Broadly speaking, CPSs are systems which consist of cyber parts (computing and digital communications), physical parts (e.g., mechanical, chemical, bio, humans, etc.), and components interfacing the two (sensors and actuators). Application domains are very broad, and cover almost every part of society: transportation (e.g., automated cars), energy (e.g., the “smart” power grid), “smart” buildings, healthcare (e.g., assisted living), and so on. Several good introductory publications on CPSs are available: see [3]–[6], and the Proceedings of the IEEE January 2012 special issue on CPSs [7], to list a few.

The design and implementation of CPSs involves two *a priori* conflicting requirements: reliability and cost. On the one hand, CPSs need to be reliable (including safe, dependable, trustworthy, and secure), because they closely interact with humans (in fact, humans can be considered to be part of the CPSs). On the other hand, CPSs need to be relatively low cost, to be affordable.

Today, we can build systems with high degrees of reliability, such as airplanes and nuclear power plants. But these systems are very expensive to build. On the other hand, we also know how to build low-cost (albeit complex) systems such as consumer electronics. But these are not very reliable (Would we trust our smart phone to drive our car?). Unfortunately, we do not have a good way to design and implement systems which meet both requirements.

In addition to safety and monetary cost, timeliness is also of the essence in CPSs, as it is in products like consumer electronics. Such products evolve quickly and fast “time to market” (which depends on design, development, and testing delays) is key to a product’s success. Unfortunately, highly reliable systems such as airplanes and nuclear power plants are not only expensive, but

Manuscript received June 5, 2015; revised October 18, 2015; accepted December 15, 2015. Date of publication March 14, 2016; date of current version April 19, 2016. This work was supported in part by the National Science Foundation under Awards #1329759 and #1139138; by the Industrial Cyber-Physical Systems (iCyPhy) Research Center. (supported by IBM and United Technologies); and by the Academy of Finland.
The author is with the Department of Computer Science, Aalto University, Espoo 02150, Finland and also with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: stavros@eecs.berkeley.edu).

Digital Object Identifier: 10.1109/JPROC.2015.2510366

also take a long time to develop, so cannot serve as good recipes for CPSs in this aspect either.

How to design reliable CPSs in a timely fashion and at low cost? As CPSs probably represent the most interesting class of (engineered) systems today, we are tempted to generalize the question, and ask: What is the best way to design systems, in general? Or, is there a science of system design?

Before proceeding, we should clarify what we mean by the terms "system" and "design." Our focus is on dynamical systems, discussed more in Section II. As for design, we use it in this paper broadly, to include not just the high-level, conceptual design process, but also lower level development and implementation phases, all the way to building an actual system. In fact, the term "design" can be broadened even further to encompass all phases of a system's lifecycle, including the ones that follow its implementation and deployment, such as system maintenance, repair, modification, and so on.

Of course, such bold generalizations run the risk of making the scope of the paper too broad. We are aware of this risk. Clearly, there are many different types of systems, and entire disciplines have developed to study them, exploiting the particularities of each domain. Nevertheless, we believe that there are certain principles which are common to all systems. Many of these common principles, e.g., stability, are studied in classical system theory disciplines which focus primarily on continuous and control systems (e.g., [8] and [9]). Others, such as compositionality, are not emphasized enough. While it is beyond the scope of this paper to provide a survey of compositionality, we provide a general discussion of the topic drawing primarily from our own research experience and the computer science literature, specifically the field of formal methods (e.g., [10] and [11]).

II. SYSTEMS

The term "system" is overloaded with many meanings. In this paper, we understand a system to be any kind of dynamical system. We view a dynamical system broadly, as anything that has a notion of state and a notion of dynamics. The state can be seen as a "snapshot" of the system at some point in time. The dynamics is a set of rules that define how the state evolves in time. We can write

$$\begin{aligned} \text{system} &= \text{state} + \text{dynamics} \\ \text{dynamics} &= \text{rules defining how state evolves in time.} \end{aligned}$$

A system may also contain inputs, which influence the dynamics, and outputs, which correspond to what is observable about the system. Note, however, that inputs and outputs need not be always present in a system. One can imagine completely uncontrollable systems (with no inputs), or completely unobservable systems (with no outputs), or even both (complete black boxes). Also note that our discussion here focuses primarily on system behavior, and somewhat less on system structure, which is clearly also essential. We will return to the notion of structure in Section V.

The above "definitions" are intentionally abstract, in order to cover many (all, if possible) classes of dynamical systems, including discrete systems, continuous systems, and combinations of the two, called hybrid systems [12]. In discrete systems such as automata or state machines [13], the set of states is discrete: finite, or countably infinite. The dynamics are typically given as a discrete set of discrete transitions, where a transition describes the current state (before the transition) and the next state (after the transition). In continuous systems [8], the set of states is continuous, described typically as a vector of state variables ranging over the real numbers. These state variables are in fact functions of time, which is also continuous (typically a nonnegative real number). The dynamics are often described by differential equations, for instance, of the form $\dot{x} = f(x)$, where x is the state variable, $\dot{x}$ is the time derivative, and f is a function on reals. This equation can be written more precisely as $(dx(t)/dt) = f(x(t))$, making explicit the fact that x is a function of time t.

Hybrid systems combine discrete and continuous systems. A state in a hybrid system is typically a pair (q, x) where q is the discrete part and x is the continuous part. Both q and x evolve in time, so both can be seen as functions of t (except in the case of nondeterministic systems; more on this below). x evolves continuously with time (for instance, governed by a differential equation), whereas q evolves in discrete "steps" or "jumps." For example, think of a heating system where q models the state of the on/off switch. Switching from on to off or *vice versa* corresponds to a discrete transition, typically considered instantaneous, which changes the value of q.

Observe that state and time are inextricably linked in a (dynamical) system, since state can be defined as something that evolves in time, and time could itself be defined as the evolution of state. Again, the definitions are abstract in order to capture different types of systems. For instance, in continuous systems, time is typically modeled as the nonnegative reals. In discrete systems, time can be modeled explicitly as the nonnegative integers, but it can also be modeled implicitly, simply by the order of events, or state changes in the system. In the latter case, time is purely qualitative or logical, as opposed to being quantitative as in a continuous system. For instance, in a discrete system execution modeled as a sequence of states $s_0, s_1, s_2, \ldots$, all we know is that the system was first at state s_0, then at state s_1, then at s_2, and so on. We do not know at what time the system was at s_1, nor how much time elapsed until the state changed to s_2, etc. These notions only have a meaning in a quantitative time model. A simple model which allows to add this information to discrete systems without adding the full power of differential equations is the model of timed automata [14].

What is the basic mathematical model of a systems science? We believe it is the model of a transition system, as it can be used to capture systems of different kinds, finite or infinite, discrete or continuous, untimed, timed, or hybrid. A transition system is essentially a (finite or infinite) graph where the nodes correspond to states and the edges correspond to transitions. A path in the graph models one possible execution of the system.[1] The states or the transitions can be labeled with additional information which helps define properties of system behaviors. For instance, the states can be labeled with a set of atomic propositions capturing the facts that are true when the system is in that state. A transition may be labeled with an action that captures the cause of the state change. The model is able to capture not just discrete transitions, but also timed transitions modeling the passage of time, and "continuous transitions" modeling continuous system trajectories [12].

This operational view of systems may seem at odds with the denotational view of systems as signal transformers, i.e., as functions which take as inputs signals (functions of time) and produce new signals as output. The latter view is typically adopted in signals and systems theory [9]. Both the operational and denotational views are useful, each having its pros and cons. The denotational view is elegant and "lighter," and can often greatly simplify reasoning and proofs (especially regarding system composition; more on this below). The operational view has the advantage (and the disadvantage at the same time) of being "lower level." This allows it to capture more accurately the behavior of systems. Indeed, there are cases where a too abstract denotational view loses critical information about a system's behavior; cf., the so-called Brock–Ackerman anomaly [15]. Another advantage of the operational view is that it lends itself to automation even when analytical methods are not available. For instance, the state space of a finite transition system can be explored exhaustively using sophisticated model-checking algorithms [16], [17].

Several generic behavioral properties of systems (beyond structural properties such as the number of inputs and outputs) can be defined and understood at the level of a transition system. For instance, a system with no inputs is deterministic if it has a unique behavior, that is, if every state in its transition system has at most one outgoing transition. A system with inputs is deterministic if it has a unique behavior for each input behavior. Denotationally, nondeterministic systems can be modeled as relations instead of functions. However, care must be taken as relations can sometimes lose information [15]. Another generic system property is input receptiveness (sometimes also called input enabledness or input completeness), which states that the system is able to accept any possible input at any time. Yet a third generic system notion is the notion of deadlocks. Intuitively, a deadlock occurs when the system is "blocked." This is easy to define formally on transition systems: a deadlock is simply a state with no outgoing transitions. These are only some of the properties which can be defined on (transition) systems in a generic fashion, without being concerned about what type of system (finite state or infinite state, discrete or continuous) the model represents. In Section IV-B, we discuss other generic system properties.

III. COMPLEX SYSTEMS

Today's CPSs are highly complex, which motivates rethinking our system design methods. Before discussing these methods, it is useful to discuss what we mean by system complexity, and where it comes from.[2]

Complexity in CPSs stems from a number of factors. Important factors are heterogeneity and multidisciplinarity. A CPS is not one system but a system of systems interacting in complex and sometimes unexpected ways. Even a relatively simple embedded control system such as adaptive cruise control involves mechanical parts, the engine, sensors, actuators, computers, computer networks, and software. Such a system is not designed by a single person,[3] or not even by a single team, but by many teams, involving stakeholders coming from different engineering disciplines (mechanical, electrical, and software engineering, to name a few) and each having a different view [18]–[20].

Complexity in modern systems also stems from the increased use of software. Software is inherently complex, perhaps the most complex artifact that humans can build today. Even small pieces of software can be intrinsically difficult to reason about. For example, consider the following simple program (written in pseudocode):

```
int n := read_input()
while(n > 1)
  if(n modulo 2 = 0) then
    n := $\frac{n}{2}$
  else
    n := $3 * n + 1$.
```

[1]Often we are interested in infinite executions, modeling situations where the system never stops. For instance, embedded controllers are supposed to interact with the system under control repeatedly, and for a long duration of time. Just how long is typically unknown, thus it is convenient to assume it is infinite. Such nonterminating systems are sometimes also called reactive systems in the computer science literature [8], to distinguish them from classical models of computation such as Turing machines, where termination is important.

[2]Our focus being engineered systems, we do not discuss physical theories of complex systems, such as chaos theory and fractals.

[3]As a colleague recently joked, such a person would need to be Leonardo da Vinci in order to cope with the complexity.

The program reads an integer number from the console and then performs a series of arithmetic transformations on the number, depending on whether it is even or odd. We may ask: Does this program terminate for any input? This seemingly simple question represents an unsolved problem in mathematics, the famous Collatz conjecture.

In addition to its intrinsic complexity, the sheer amount of software in modern systems makes reasoning about these systems a daunting task. How many lines of code does it take to run a premium car? According to some sources, on the order of 100 million [21], [22]. This software performs hundreds or thousands of functions.[4] Even a small system such as a pacemaker can have up to a hundred thousand lines of code. In the case of a car, the software does not run on a single computer, but on a distributed, networked execution platform consisting of 70–100 electronic control units (ECUs) [21], [22]. In addition to ECUs, the system is connected to sensors and actuators (according to http://www.automotivesensors2015.com the number of sensors in a car ranges in the hundreds). This is just a single vehicle. When designing a large-scale CPS such as an automated intersection, one needs to consider a large and varying number of vehicles, interacting with many other systems, including humans. Clearly, size is another key factor of system complexity.

IV. SYSTEM DESIGN METHODS

Perhaps the most common system design method is design by trial and error. It consists in building a system prototype, testing it, finding bugs, and repeating the process until the system is deemed satisfactory for release. In addition, in most cases, bug fixing does not stop after system deployment but continues during the lifecycle of the system. This is especially true for software, where regular software updates (including bug fixes) are accepted software engineering practice. But the phenomenon is encountered in other industries as well, for instance, in the automotive industry, where car recalls are becoming commonplace [23]. Recalls are common in the medical device industry as well: see http://www.fda.gov/MedicalDevices/Safety/.

Design by trial and error is both costly and unsafe. It is costly mainly because finding errors after a system prototype is built is too late in the development process, and fixing those bugs is difficult and takes precious time. Finding bugs after the system is deployed is even worse. Design by trial and error is unsafe because unreliable CPSs put human lives at risk.

A more systematic and rigorous system design approach is so-called model-based design (MBD). The main idea is to build system models instead of system prototypes. Such models can be anything from simple spreadsheets to detailed, executable models of system behavior (or aspects thereof) used, for instance, for simulation (this is sometimes referred to as virtual prototyping). There are several advantages to using models rather than prototypes. First, it is safer to test a model than a real system. Second, a model is often cheaper to develop. Third, it is often faster to test a model (by running a simulation, or many simulations in parallel). As a result, more tests can be run during the available time budget. Moreover, it is sometimes possible to exhaustively test a model, that is, to formally verify that all its behaviors are correct (more on verification below). Last, when bugs are found, they can be fixed more easily (often simply by modifying the model itself) and earlier in the development process.

The main disadvantage of models, of course, is that they are not the "real thing." They are abstractions of reality, and as such contain inherent approximations and inaccuracies. Care must be taken so that these approximations are managed in a systematic and (mathematically) rigorous manner, so that the results of the analysis are not meaningless. This is one of the key scientific challenges in system design.

In summary, (model-based) system design can be seen as the process of answering three main questions.

1) Modeling: What is the system we want to design?
2) Analysis: Is this indeed the system that we want?
3) Implementation: How can this system be built?

Model-based design is widely accepted in today's industrial practice, especially in the domain of embedded systems. For an in-depth study of the multiple facets and applications of MBD we refer the reader to the scientific literature (e.g., [24] and [25]) as well as commercial websites (e.g., http://www.mathworks.com/solutions/model-based-design/). Concrete industrial case studies are often difficult to find, both due to intellectual property restrictions, as well as due to the size and complexity of systems involved. Nevertheless, a positive recent development is the set of benchmarks collected within the Applied Verification for Continuous and Hybrid Systems workshop (ARCH); see http://cps-vo.org/group/ARCH. Several of these benchmarks are accompanied by publications where the models and challenge problems are explained in detail, e.g., [26]–[28].

Although MBD has become widely accepted in practice, it is by no means a solved problem. In what follows, we identify some of the main research challenges in each of the three domains listed above.

[4]News on emissions control software emerging during the writing of this paper provide a noticeable illustration of the scope of these functions and their potential (see https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal).

A. Modeling

As mentioned above, a significant challenge is how to build accurate models. This is hard enough to justify entire communities specializing in modeling specific types of systems from various domains (mechanical, chemical, biological, etc.).

But there is also another challenge in modeling, especially in the MBD context, namely, how to come up with the right modeling languages. This is a language design problem. The question is how to be able to describe CPSs in the best possible way, where "best" includes ease and economy of description, understandability, mathematical rigor, lack of vagueness and ambiguity in semantics, and several other criteria. Here, we are not necessarily advocating a single CPS modeling language. There are likely to be many languages specialized to the field they all target. As such, these languages' primary users will be field specialists with little background in computer science or engineering, programming languages, etc., which brings up research problems in domain-specific languages.

It is important for these modeling languages to be executable [29], that is, amenable to automated, computer-aided analysis. At a minimum this would mean simulation; exhaustive verification would be even better.

B. Verification

The second element of MBD is analysis. The goal of analysis is to convince the designer that the system she designed is the system she intended to design, that is, free of errors, performance and cost issues, etc. The goal is to do that before actually building the system, in order to minimize surprises at the later stages.

In this sense, analysis is itself a vast field, including all types of correctness, performance, reliability, and cost analyses, using simulation-based or analytic (closed-form) methods. As systems become more complex, analytic methods are more difficult to apply, and computer-aided methods become more prevalent. In this context, we view formal, exhaustive verification methods such as model checking as the biggest research challenge in the domain of analysis, and one of the key challenges in system design in general.

The model-checking problem can be stated as follows: given a system S and a property ϕ, check (ideally fully automatically) whether S satisfies ϕ; and if it does not, explain why. S is a mathematical model of the system (the one produced by the modeling process). ϕ is also a mathematical statement capturing the specification of the system (or parts of it) in a rigorous and unambiguous manner. Formal specification languages such as temporal logics are the languages of choice when writing ϕ, especially in the case of dynamical systems, where we want to specify properties about the behavior of the system in time. Temporal logics have the advantage of being rigorous on the one hand, and relatively close to natural language, e.g., English (in fact translating natural language to temporal logic is an active area of research).

An example specification, written in so-called linear-temporal logic (LTL), is the formula $\Box p$ (always p) which states that p holds all along a given execution of the system, i.e., at every state visited during the execution. Here, p is an atomic proposition which can be evaluated on a single state (either it holds on that state, or it does not). For example, p might model all "legal" states of a program, where no exception has been thrown. $\Box p$ is a safety property. Safety properties, roughly speaking, state that the system must never do anything wrong. The dual class of liveness properties state that the system must do something right. For example, the LTL formula $\Diamond p$ (eventually p) states that some state where p holds will be reached during the system execution. This can be used to model, for example, program termination. Both safety and liveness are necessary, since requirements from one class alone can be satisfied by trivial systems. For example, the system that does nothing (deadlocks immediately) never does anything wrong, and therefore trivially satisfies all safety properties.

Notions such as safety and liveness are generic and apply to all kinds of systems, just like determinism and receptiveness. We therefore view topics such as safety and liveness as an integral part of the foundations of system design.

Another way to specify and verify systems is by comparing them to each other using equivalence and other relations on transition systems, such as bisimulation [17]. This is another fundamental topic in system design, as such relations can also serve to define abstractions of systems and to reduce their size. Bisimulation has been used, for instance, to collapse infinite timed or hybrid transition systems to finite ones, amenable to verification [12], [14].

The biggest drawback in formal verification is that it is computationally very expensive. Model checking is often plagued by the state explosion problem. Model checking algorithms work by exploring, in one way or another, the state space of the system, that is, the set of all reachable states. State explosion refers to the fact that this state space is typically huge and impossible to exhaustively explore in a reasonable amount of time, or within the available memory of the computing platform. For example, a program with ten 32-b integer variables has potentially $(2^{32})^{10} = 2^{320}$ reachable states. The situation is even worse in the context of CPSs where models also involve continuous variables, with an infinite state space. In fact, verification problems are typically undecidable for hybrid system models [30]. Nevertheless, advances are constantly being made in the verification field, and many practical verification tools currently exist, for the hardware, software, and CPS domains.

For a detailed exposition of the topics of verification, model checking, temporal logics, etc., we refer the reader to the textbooks [16], [17], as well as [31] which is more practice-oriented focusing on the widespread model checker *Spin*. Several verification success stories have been reported in the literature, for instance, see [32]–[35]. Verification methods for CPS and hybrid systems are discussed in [12] and [36]. An introduction to several of these topics (as well as other fundamental topics in CPS, such as real-time scheduling) is provided in [11].

C. Implementation

Implementation is a primary challenge in MBD since, in the end, building models is not enough. A real system also needs to be built. The challenge here is twofold: first, how to generate systems automatically from models; second, how to guarantee that the properties of the model are preserved in the generated system. On the first point, we envisage computer-aided techniques such as code generation, which already exist and are widely available in the industry. Regarding the second point, the key requirement is semantical preservation: the generated system must behave equivalently to the original model, otherwise, the analysis results obtained from the model are rendered meaningless. Strict semantical preservation is an ideal which is difficult if not impossible to achieve in most cases. For instance, a controller developed in theory as a function on real numbers does not admit a strictly equivalent implementation on a computer using finite-precision arithmetic. Luckily, strict semantical preservation is not always required. In many cases, techniques can be developed to preserve the essential properties of the original model which are required to achieve some overall goal. The remaining properties of the original model, those not contributing to this goal, can be relaxed during implementation. Examples of this approach are the works [37] and [38] which propose semantics-preserving techniques for the implementation of synchronous models on different types of asynchronous (distributed, or multitasking) execution platforms.

V. COMPOSITIONALITY

The generic definition system=state+dynamics (Section II) is not entirely satisfactory. It is too monolithic, in the sense that it requires to capture the entire system in "one shot." This is possible only for small systems. Imagine having to describe the entire set of states as well as the dynamics of a large system, e.g., a car, an airplane, a biological cell, or even an entire organism. Such complex systems nevertheless have a certain structure, and can be better understood and described as systems of systems, that is, compositions of subsystems, each of which may itself be composed of *sub*subsystems, and so on. Given this observation, we can provide a nonmonolithic definition of systems as follows:

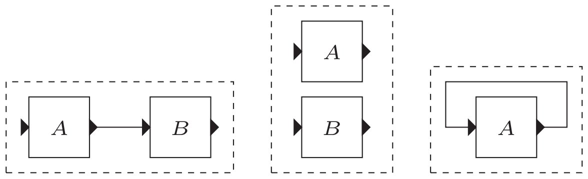

Fig. 1. *Composition in series (left), in parallel (middle), and by feedback (right).*

$$\begin{aligned}
\text{system} &= \text{atomic system} \mid \text{composite system}\\
\text{atomic system} &= \text{state} + \text{dynamics}(+\ \text{inputs/outputs})\\
\text{composite system} &= \text{set of subsystems} + \text{composition}\\
\text{dynamics} &= \text{rules defining how state evolves}\\
&\quad\ \text{in time}\\
\text{composition} &= \text{rules defining how subsystems}\\
&\quad\ \text{interact.}
\end{aligned}$$

Just like there are many kinds of system dynamics, there are many types of system interaction, studied in the (broadly speaking) theory of concurrent systems. At the basic level of transition systems, two composition paradigms are fundamental: synchronous and asynchronous. In synchronous composition, all subsystems move in "lock step," that is, they all perform a transition simultaneously. In asynchronous composition, also called interleaving, only one subsystem makes a move (transition) at a time, while the states of the others remains unchanged.

When systems are viewed denotationally, they can be composed in series, in parallel, or in feedback,[5] as illustrated in Fig. 1. But this is only a high-level, structural view of composition, which does not provide all the necessary information in order to derive the behavior of the product system. For instance, we may ask: What do the connections in Fig. 1 represent? In some models they represent "wires" with zero delay, so that in serial composition the output of A is immediately available as an input to B. This can be seen as a synchronous mode of composition. In other models like dataflow the connections represent first-in–first-out (FIFO) queues [40]. Since writing to a queue is decoupled from reading, the composition here is asynchronous. Yet in other models,

[5]Our discussion focuses on composition of systems with static structure, where the set of subsystems and the interaction rules are constant over time. Systems with dynamic structure are also very important, especially in certain application domains, e.g., biology. State-transition models can be extended to deal with dynamic reconfiguration and creation/death of subsystems [37].

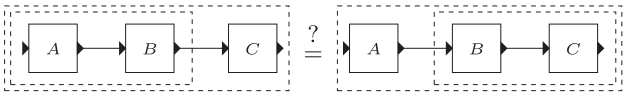

Fig. 2. ***Associativity implies that the composition on the left and the one on the right should be equivalent.***

there are no explicit connections and interaction between subsystems is achieved by shared memory. This is the model typically used for concurrent software: asynchronous threads communicating via shared variables.

Properly defining the semantics of composition is often nontrivial, in particular if one wants to obtain a compositional theory where composition operators form an algebra with some natural and desirable properties. For instance, one such property is associativity, illustrated in Fig. 2 for the case of serial composition. Here, the property states that composing first A with B, and then their product with C, should be equivalent to composing A with the product of B and C.

Associativity is only one example of a compositionality property. Compositionality itself is an overloaded term with many meanings. Without attempting to be exhaustive, we discuss several other types of compositionality in the rest of the paper. Generally speaking, we can view compositionality as having two aims: first, to master complexity by allowing complex systems to be constructed from simpler components (e.g., see [41]); second, to master heterogeneity by allowing to compose systems of different kinds (e.g., see [42] and [43]).

Composition and compositionality should probably be the most important concepts in modern system thinking. Unfortunately, they are not always easy to achieve, and are therefore often neglected in system design methods and tools. It is important to emphasize that we do not view compositional methods as an alternative to MBD, but as an integral part of it [20]. Compositional methods add new tools to the panoply of MBD tools. Using these new tools may incur some extra cost, but it also enables to do things that could not be done without them. In the rest of this section, we illustrate some of these compositional tools, from our own work.

A. Interfaces for Compositionality of Hierarchical Modeling Languages and for Modular Code Generation

Successful tools such as Simulink from the industry (Mathworks) and Ptolemy from the academia (University of California Berkeley) use hierarchical modeling as a powerful mechanism for structuring large models and managing their complexity. For example, in a language based on block diagrams, hierarchy allows to encapsulate a block diagram into a composite block (sometimes called subsystem). The internals of the composite block can then be hidden, and the composite block can be further connected and encapsulated. Hierarchies of arbitrary depth can be built in this manner. Unfortunately, the way hierarchy is handled in state-of-the-art tools is limited, because the encapsulation process may lose information. The reason for this is that several of the models on which these tools are based are noncompositional in a fundamental sense [44]–[47].

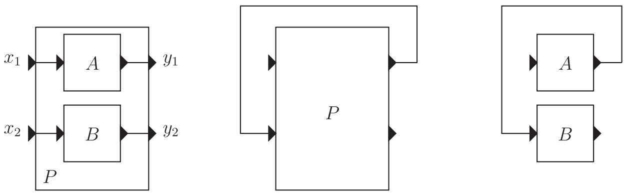

Fig. 3. ***Hierarchical block diagram (left), a possible way to connect product block P (middle), and the same model after flattening P (right).***

To illustrate the problem consider the example in Fig. 3. The hierarchical block diagram shown to the left corresponds to the parallel composition of blocks A and B. The product is block P. Suppose we now want to connect P in feedback, as shown in the middle of the figure. One expects this composition to be equivalent to the serial composition of A and B, which is indeed obtained if one "flattens" the hierarchy as shown to the right. Unfortunately, this is not always the case. The model with feedback is often considered illegal, since it appears to contain a cyclic dependency (sometimes called an algebraic loop). Indeed, without knowing the internals of block P, it appears that its output y_1 depends on its input x_2, and therefore cannot be connected in feedback.

The problem arises from a fundamental limitation of the basic model of Mealy machines, which are not closed under parallel composition. A Mealy machine is typically defined as a tuple $(I, O, S, s_0, \delta, \lambda)$, where I is the set of input values, O is the set of output values, S is the set of states, $s_0 \in S$ is the initial state, $\delta : S \times I \rightarrow S$ the transition function, and $\lambda : S \times I \rightarrow O$ is the output function. The functions δ and λ can be viewed as the interface with which an external user interacts with the machine. This is much like the notion of interfaces in object-oriented programming.

The signature of λ indicates that generally the output depends instantaneously on the input. When composing two machines, the sets of inputs, outputs, and states of the product machine are typically defined to be the Cartesian products of the corresponding sets of the composed machines, thus $I_1 \times I_2$, $O_1 \times O_2$, and $S_1 \times S_2$. Then, the product machine is defined to have a single-output function with signature $(S_1 \times S_2) \times (I_1 \times I_2) \rightarrow (O_1 \times O_2)$. This "monolithic" interface loses input–output (non)dependency information. For example, it does not allow to capture the fact that, in Fig. 3, output y_1 does not depend on input x_2.

A solution to this problem is to consider nonmonolithic interfaces [44], [46], that is, Mealy machines with several output functions, as many as needed to accurately represent the input–output dependencies of the original model. In the example of Fig. 3, P needs to have two output functions, one representing the dependency of y_1 on x_1, and another the dependency of y_2 on x_2. Interesting questions arise, such as how many output functions does a system generally need, how to generate them automatically from its submodels, etc. These questions are intrinsically related to the problem of modular code generation, that is, generation of code from blocks such as P, independently from context, and without flattening. In fact, modular code generation was the original motivation behind our work [44], [46], where the details on how the above questions are answered can be found. Lublinerman *et al.* [46] also report on several case studies of automatic generation of nonmonolithic interfaces and code from real-life Simulink models from the automotive domain.

Closure under composition is a type of a compositionality property. As the discussion above shows, the standard Mealy machine model (with a single-output function) is not closed under parallel composition, in the sense that the parallel composition of two Mealy machines cannot be represented by an equivalent Mealy machine. On the other hand, Mealy machines with multiple-output functions are closed under parallel composition, and therefore can be considered a "more compositional" model than standard Mealy machines. It is worth noting that Moore machines are closed under parallel composition. In Moore machines, the output function depends only on the current state and not on the input, i.e., it has signature $\lambda : S \to O$. For this reason, Moore machines cannot express stateless ("combinational") components such as input–output functions.

Noncompositionality problems (in the sense of nonclosure under composition) arise not just in the model discussed so far (essentially, hierarchical block diagrams with synchronous semantics *á la* Simulink), but in other hierarchical models as well. For instance, the hierarchical version of the popular dataflow model SDF [48] is shown to be noncompositional in [47], using the example reproduced here in Fig. 4.

A hierarchical SDF graph is shown to the left of Fig. 4, consisting of two SDF actors A and B connected in series. A needs one token in order to fire, consumes it, and produces two tokens each time it fires. B consumes three tokens and produces one token each time it fires. In SDF, as in dataflow models in general, connections represent FIFO queues, so that when connecting A to B in series, the tokens produced by A are stored in the input queue of B. When enough tokens are available (three in this case) B can fire.

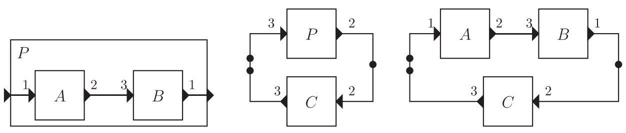

Fig. 4. ***Hierarchical SDF graph (left), a possible way to connect product block P with another SDF actor C (middle), and the same model after flattening P (right). Bullets represent initial tokens. Example taken from [45].***

We would like to represent the product actor P as an atomic SDF actor. The natural way to do so is to consider that P consumes three tokens and produces two tokens each time it fires. This corresponds to P internally firing A three times, followed by B two times. Doing so, the queue connecting A and B remains empty after every firing of P, since the total number of tokens produced by A $(3 \times 2 = 6)$ equals the total number of tokens consumed by B $(2 \times 3 = 6)$.

Now, suppose we connect P with another SDF actor C, as shown in the middle of Fig. 4. The "bullets" in this figure represent initial tokens, so that the queue from C to P contains two initial tokens, and the one from P to C contains one initial token. This model deadlocks: P cannot fire, because it needs three tokens, but only two are initially available; C cannot fire either, because it needs two tokens, but only one is available.

But consider what happens when we flatten P, as shown to the right of Fig. 4. In this model, there is no deadlock. Indeed, A can fire twice, consuming the two initial tokens and producing four tokens for B. B can then fire once, consuming three out of these four tokens, and producing one at each output queue, which now has two tokens in total, including the initial one. This allows C to fire, consuming these two tokens and producing three. A and B can fire once each after that, bringing the system to its initial state, from which the same sequence of firings can repeat indefinitely.

At a fundamental level, the problem here is again a problem of noncompositionality, as in the example of Fig. 3. Just like Mealy machines are not closed under parallel composition, SDF actors are noncompositional, in the sense that the serial composition of two SDF actors is not an SDF actor. (In fact neither is the parallel composition [47].) And just like Mealy machines can be made compositional by extending their interface to a nonmonolithic one, the model of SDF can be extended to become compositional. This extension is too involved to summarize here: see [47] for details.

B. Interfaces for Incremental Design and Verification

Suppose we have designed a system P consisting of three subsystems, A, B, and C, connected in some way.

Suppose further that we have verified that P satisfies some specification ϕ. As is often the case, at some point in the lifecycle of P, we may need to replace one of its components with another one. Suppose we want to replace component B with a new component B', yielding a new system P'. Suppose that the overall specification does not change, and is still ϕ. The problem that the designer faces is that P' does not necessarily satisfy ϕ. The question then becomes how to avoid rechecking ϕ on the new system P' from scratch? This question is important, since verification is expensive no matter what form it takes (simulation, testing, or model checking). It would be therefore quite beneficial to have incremental verification methods, which allow us to reuse the fact that P satisfies ϕ when verifying P'.

A class of compositional frameworks well suited both for incremental design and verification are theories which include a notion of refinement. The essential elements of these theories are: 1) a notion of component; 2) one or more composition operators; and 3) a notion of refinement, which is a binary relation between components. In addition, the theories often include standard verification notions, such as properties, satisfaction of properties by components, etc. In what follows, let us assume for simplicity a single composition operator, which we denote $\circ$. Refinement is denoted by $\sqsubseteq$ and satisfaction of properties by $\models$. Refinement theories provide two fundamental theorems.

1) Preservation of properties by refinement, which can be stated as follows: if component A satisfies property ϕ, and A' refines A, then A' also satisfies ϕ. Using our notation

$$(A \models \phi \wedge A' \sqsubseteq A) \Rightarrow A' \models \phi.$$

2) Preservation of refinement by composition, which can be stated as: if A' refines A and B' refines B, then the composition of A' and B' refines the composition of A and B. Using our notation

$$(A' \sqsubseteq A \wedge B' \sqsubseteq B) \Rightarrow A' \circ B' \sqsubseteq A \circ B.$$

Preservation of refinement by composition can be seen as another type of compositionality property.

Combined these two theorems are powerful, and can be used, for instance, to reduce the incremental verification problem to a problem of checking refinement between two components. To illustrate this, consider again the example of replacing B by B', discussed in the beginning of this section. We know that $P \models \phi$, i.e., $A \circ B \circ C \models \phi$ (note that we implicitly assume associativity here, so that $(A \circ B) \circ C = A \circ (B \circ C)$). Suppose we can prove that $B' \sqsubseteq B$, i.e., B' refines B. Then, by preservation of refinement by composition, we can deduce that $A \circ B' \circ C \sqsubseteq A \circ B \circ C$, i.e., that $P' \sqsubseteq P$ (here we implicitly assume that every component refines itself, i.e., $A \sqsubseteq A$ for every A). Now, by preservation of properties by refinement we can conclude that $P' \models \phi$. What is interesting in this approach is that the verification question $A \circ B' \circ C \models \phi$ is reduced to the refinement-checking question $B' \sqsubseteq B$. The latter is presumably easier to answer, e.g., less computationally expensive, since the components B and B' are typically small, whereas the composition of a number of them can be large.

Compositional frameworks with refinement abound, e.g., [49]–[51] to mention only a few. Here, we focus our attention on so-called interface theories [52], [53], which have the additional characteristic of composition being a partial operation. This is important, as it allows to express incompatibility, that is, the notion that two components are not compatible, and thus their composition is illegal. Checking compatibility in system design is as useful as type checking in programming languages. Type checking can be seen as an inexpensive, "lightweight" verification method. The reason is not only that verification algorithms are typically more computationally expensive than type-checking algorithms. It is also the fact that verification requires a property to be checked, in addition to the program on which to check it, whereas type checking only requires the program. Thus, type checking places no extra burden on the programmer. Interface theories can be seen as type theories for dynamical systems.

To be able to express compatibility, it is essential that systems be allowed to be non-input-receptive, that is, able to declare certain inputs as illegal (at certain times). See [54] for an extensive discussion of this point. Here, we illustrate what this means by example.

Consider Fig. 5. The figure shows two (separate) block diagrams. Each diagram uses the block labeled $\sqrt{\ }$ which computes the square root function. Such blocks are common in tools like Simulink. In the leftmost diagram, the square-root block is connected in series to a block producing the constant -1. Assuming we are not dealing with imaginary numbers, the square root block requires its input x to be nonnegative. Therefore, the

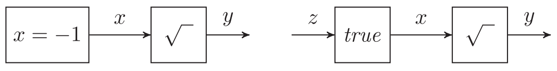

Fig. 5. *Two block diagrams where the square root block is connected in series with two other blocks (the constant –1, left; a block for which nothing is known, right). Relational interfaces [52] can be used to specify input–output relationships and illegal inputs, and to detect incompatibility in both connections.*

connection in the leftmost diagram is illegal, i.e., the two blocks are incompatible.

To catch such incompatibilities, we can use the framework of relational interfaces [54] and specify each block using a logical formula on its input and output variables, called a contract. The constant block has only an output variable x, and is specified by the contract $x = -1$. The square root block has input x and output y, and can be specified by the contract

$$x \geq 0 \wedge y = \sqrt{x} \tag{1}$$

where $\wedge$ denotes logical conjunction. Contracts of this kind can be seen as rich component types (richer than, say, the function signature `real` $->$ `real`).

It is important to emphasize the use of conjunction rather than implication in (1). We could have written the contract as $x \geq 0 \rightarrow y = \sqrt{x}$, but the meaning would be different. Formula $x \geq 0 \rightarrow y = \sqrt{x}$ states that *if* the input x is nonnegative, *then* the output will be the correct square root value. This formula is trivially satisfied when x is negative (in which case y can be anything). On the other hand, contract (1) states explicitly that all inputs $x < 0$ are illegal, because (1) in conjunction with $x < 0$ is unsatisfiable. This is how incompatibility can be detected automatically in this example: by forming the conjunction $(x = -1) \wedge (x \geq 0 \wedge y = \sqrt{x})$ and checking satisfiability of the resulting formula.

Interestingly, conjunction is not always sufficient for detecting incompatibility. To see this, consider the rightmost diagram of Fig. 5. Here, the constant block is replaced by a block about which very little is known. In particular, we cannot guarantee anything about its output x, and thus define its contract to be the formula true, meaning that any value for x is possible, independently of the value of the input z. Clearly, this diagram also contains an incompatibility: if nothing can be guaranteed about x, then x might be negative, which violates the input requirements of the square-root block. However, the conjunction $(\text{true}) \wedge (x \geq 0 \wedge y = \sqrt{x})$ is satisfiable, which does not indicate any incompatibility *a priori*.

The solution is to add extra constraints to the conjunction above, specifically, the term

$$\forall x : \text{true} \rightarrow x \geq 0 \tag{2}$$

and in general

$$\forall x : \big(\phi_1 \rightarrow (\exists y : \phi_2)\big) \tag{3}$$

for two components with contracts ϕ_1 and ϕ_2 connected in series [54]. Intuitively, the subformula $\exists y : \phi_2$ in (3) is a constraint on x, characterizing the set of legal inputs of the downstream component. Overall, (3) states that, given input z, every output x that the upstream component ϕ_1 may produce, is a legal input for the downstream component ϕ_2. If this is not the case, z cannot be considered a legal input of the overall system, since it cannot ensure the compatibility of the subsystems.

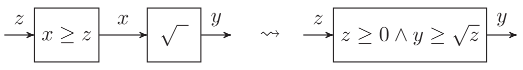

Fig. 6. ***Serial composition of two relational interfaces (left) and resulting relational interface (right).***

In our example, formula (2) is obtained from (3) by replacing ϕ_1 with true and ϕ_2 with (1). Then, subformula $\exists y : \phi_2$ becomes $\exists y : x \geq 0 \wedge y = \sqrt{x}$, which is equivalent to $x \geq 0$. Next, (2) becomes $\forall x : \text{true} \rightarrow x \geq 0$. This simplifies to $\forall x : x \geq 0$ since true $\rightarrow \phi$ is equivalent to ϕ. And $\forall x : x \geq 0$ is equivalent to false (i.e., unsatisfiable), since it states that all numbers are non-negative. In this way, we are able to detect incompatibility in the rightmost diagram.

Detecting such incompatibilities is akin to type checking, as mentioned above. The same techniques can be used to infer new constraints on the inputs, which is akin to type inference. Consider the example of Fig. 6. The block diagram shown to the left is similar to the one on the right of Fig. 5, except that now something is known about the leftmost block: namely, that its output x is guaranteed to be no less than its input z. This is specified as the relational interface $x \geq z$ for this block. The question is: Are the two blocks compatible? The answer is yes: although x might be negative (for instance, $z = x = -1$ satisfies $x \geq z$), there is a way to constrain external input z so as to guarantee that x is nonnegative. The weakest such constraint on z is $z \geq 0$. Interestingly, this is obtained directly from (3) by replacing ϕ_1 with $x \geq z$ and ϕ_2 with (1). Then, (3) becomes $\forall x : x \geq z \rightarrow x \geq 0$, which after quantifier elimination is found equivalent to $z \geq 0$. The final contract of the product system is shown to the right of Fig. 6. It is obtained by taking the conjunction of the two original contracts and (3), and then eliminating internal variable x by existential quantification: $\exists x : (\phi_1 \wedge \phi_2 \wedge (3))$.

As a final example, we illustrate how refinement can be used in this framework. Consider a new square root component, capable of handling negative inputs. Denote this new component by $\sqrt{\ }^*$. Its contract is defined to be $y = \sqrt{x}$, where y will be an imaginary number when $x < 0$. Now, according to the definition of refinement in the relational interface theory, we can prove that $\sqrt{\ }^*$ refines $\sqrt{\ }$. This in turn implies that $\sqrt{\ }^*$ can replace $\sqrt{\ }$ while preserving properties of the original system. In particular, if the system with $\sqrt{\ }^*$ has no incompatibilities, then the system with $\sqrt{\ }$ is guaranteed to have no incompatibilities either.

Formally, for two components A and A' with contracts ϕ and ϕ', and assuming for simplicity that both components have a single input variable x and a single-output variable y, A' refines A iff the following conditions hold:

$$\forall x : \big((\exists y : \phi) \rightarrow (\exists y : \phi')\big) \tag{4}$$
$$\forall x : \big((\exists y : \phi) \rightarrow (\forall y : \phi' \rightarrow \phi)\big). \tag{5}$$

Intuitively, condition (4) states that if an input value is legal in A then it is also legal in A'. Observe that this is the case in $\sqrt{\ }$ and $\sqrt{\ }^*$: every legal input of $\sqrt{\ }$ (i.e., every nonnegative real number) is also legal for $\sqrt{\ }^*$ (i.e., is a real number). Condition (5) states that, for those inputs which are legal in A, every output that A' may produce is also a possible output of A. Again, this holds for $\sqrt{\ }$ and $\sqrt{\ }^*$ since for every nonnegative number x their outputs are identical and equal to $\sqrt{x}$.

Note that refinement allows A' to accept inputs which are illegal for A, and places no requirements on what A' outputs when given these extra inputs. This is correct since, when we replace A with A', the upstream component feeding inputs to A' is guaranteed not to provide these extra inputs (otherwise the original composition with A would be incompatible). Indeed, as shown in [54], this notion of refinement captures substitutability (when can A' replace A) in a necessary and sufficient way.

Automating the techniques presented above requires algorithms and tools for checking satisfiability of formulas, as well as auxiliary methods for quantifier elimination and formula simplification. These techniques are generally available in constraint solvers, SAT (satisfiability) and SMT (SAT modulo theory) solvers, theorem provers, and similar tools. The computational complexity in theory and practice varies depending on the type of logic used. In the examples above, we used first-order logic with arithmetic constraints. These examples also focus on simple, stateless components. The theory of [54] can handle components with state, but is limited to safety properties. Extensions of the theory to handle liveness properties have been proposed in [55], so that contracts can be written, for example, in temporal logic. Applications of the theory to Simulink models from the automotive domain are reported in [56].

C. Interfaces for Cosimulation

We end this section by briefly mentioning another important muse of interfaces for compositionality, from the domain of simulation. There, designers often find themselves with a plethora of models, developed using different modeling and simulation tools. The problem is how to somehow "connect" these models (and the corresponding tools) and simulate them together, without translating all models into the language of a single tool. Such a translation is not always possible, since different tools specialize in different domains. Even when possible, translation can be costly and brittle, and may result in nonoptimal models (or models not exploiting the optimized engine of a certain tool).

Cosimulation refers to performing simulation with multiple, interoperating tools. Recently, the functional mockup interface (FMI) has been proposed as a cosimulation standard (see https://www.fmi-standard.org/). FMI defines a standard API (i.e., an interface) which submodel components [called functional mockup units (FMUs)] must implement. Different FMUs can be generated from different tools and loaded to a master simulation engine (also called a master algorithm). Since all the FMUs implement the same API, the master algorithm does not need to know what is the internal model within each FMU. The master only knows the FMI API, and executes the FMUs via method calls to this API.

There are several challenges with cosimulation in general and FMI in particular. One important question is what is the right interface? This question applies to modular simulation approaches in general, and has received different answers in FMI but also in tools such as Simulink or Ptolemy [42], [43]. Understanding the pros and cons of these different solutions is an interesting problem. One criterion is to what extent the interface can be used to capture different modeling languages. This is not simply a software engineering problem, as there exist several semantic gaps between the source modeling language and the target API. How to close such semantic gaps is an interesting problem, studied in [57] for the FMI API. Another problem is, given an interface, how to design a master simulation algorithm with good properties, such as determinism and reproducibility of simulation results. Such a master algorithm is proposed in [58], in the context of FMI. The FMI framework and this master algorithm have been used to connect timed and hybrid system modeling tools for cosimulation case studies [59].

VI. CONCLUSION

Separate disciplines have emerged to study systems of different kinds, but is there a science of design which applies to engineering systems in general? We do not answer the question in this paper, but provide some food for thought and for follow-up work. Our view of systems is influenced from our own computer science background and work in formal methods and verification. We believe that these fields have a lot to contribute to the design of CPSs, not only because many of these systems are safety critical, but also because these fields provide some fundamental system principles (e.g., see discussions in Sections II and IV-B, and also [2], [11], and [62]). Our discussion focused on compositionality as a key principle in system design, and on interfaces as a versatile tool with many uses in system composition. Interfaces abstract components, hiding internal information and exposing only what is relevant for composability. We note that although our discussion in Section V focused on an input–output view of systems, this is by no means the only possible view. Another widely used view, especially useful for modeling physical systems, is acausal (or equational) modeling, adopted in languages like Modelica [60].

Abstraction itself is another key principle which plays a prominent role in system design, beyond interfaces. Abstraction is essential to reduce complexity and make system models and designs more understandable. At the same time, it is necessary to qualify which properties are preserved by different types of abstraction. Systematic studies of this problem have been motivated by and found several usages in the field of formal verification (e.g., see [34] and [61]).

Compositionality also goes beyond the logical setting that we discussed here. Compositional frameworks exist for performance, schedulability, and other types of analysis (e.g., see [63]–[66]). Compositionality can also be extended beyond the standard notion of composition of components, e.g., to a notion of superposition of views [17]. Finally, compositionality has been studied in the context of continuous-time and control systems. Recent work in that context includes the work on passivity for compositional control design [67].

In addition to the above, important topics in system design include: the interface between discrete and continuous systems, which raises interesting questions on how to best integrate physical notions such as time and robustness into the computational world [68], [69]; design-space exploration [70]; and the related topic of system synthesis. Traditionally, synthesis emerged as a formal method to generate automatically correct-by-construction systems from their specifications [71]. This facing the same (or worse) scalability problems than verification, a less "purist" approach to synthesis is now emerging. Examples of this approach are automatic completion of incomplete programs or protocols (e.g., [72]–[75]).

Clearly, many other disciplines are also essential to system design, such as game and decision theory, optimization theory, scheduling theory, operations research, and many more. Sometimes these theories take a macroscopic view of systems (which is only fitting to master complexity of large systems). It is an interesting challenge to combine this view with the more microscopic view of engineering disciplines. ■

REFERENCES

[1] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *FM 2006: Formal Methods*, vol. 4085, LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Berlin, Germany: Springer-Verlag, 2006, pp. 1–15.

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

[3] E. Lee, "Cyber physical systems: Design challenges," in *Proc. 11th IEEE Int. Symp. Object Oriented Real-Time Distrib. Comput.*, May 2008, pp. 363–369.

[4] L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang, "Cyber-physical systems: A new frontier," in *Proc. IEEE Int. Conf. Sensor Netw. Ubiquitous Trustworthy Comput.*, Jun. 2008, pp. 1–9.

[5] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proc. 47th ACM/IEEE Design Autom. Conf.*, Jun. 2010, pp. 731–736.

[6] K.-D. Kim and P. Kumar, "Cyber-physical systems: A perspective at the centennial," *Proc. IEEE*, vol. 100, no. Special Centennial Issue, pp. 1287–1308, May 2012.

[7] R. Poovendran *et al.*, "Special issue on cyber-physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 6–12, Jan. 2012.

[8] D. G. Luenberger, *Introduction to Dynamic Systems—Theory, Models & Applications*. New York, NY, USA: Wiley, 1979.

[9] A. V. Oppenheim, A. S. Willsky, and I. T. Young, *Signals and Systems*. Prentice-Hall, 1983.

[10] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York, NY, USA: Springer-Verlag, 1995.

[11] R. Alur, *Principles of Cyber-Physical Systems*. Cambridge, MA, USA: MIT Press, 2015.

[12] P. Tabuada, *Verification and Control of Hybrid Systems: A Symbolic Approach*. New York, NY, USA: Springer-Verlag, 2009.

[13] Z. Kohavi, *Switching and Finite Automata Theory*, 2nd. New York, NY, USA: McGraw-Hill, 1978.

[14] R. Alur and D. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, 1994.

[15] J. Brock and W. Ackerman, "Scenarios: A model of non-determinate computation," in *Proc. Int. Colloq. Formalization Programming Concepts*, London, U.K., 1981, pp. 252–259.

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

[17] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[18] D. Broman, E. Lee, S. Tripakis, and M. Törngren, "Viewpoints, formalisms, languages, tools for cyber-physical systems," in *Proc. 6th Int. Workshop Multi-Paradigm Model.*, 2012.

[19] J. Reineke and S. Tripakis, "Basic problems in multi-view modeling," in *Proc. Tools Algorithms Construction Anal. Syst.*, 2014.

[20] S. Tripakis, "Foundations of compositional model-based system design," in *Cyber-Physical Systems: From Theory to Practice* D. Rawat, J. Rodrigues, and I. Stojmenovic, Eds., Boca Raton, FL, USA: CRC Press. [Online]. Available: https://www.crcpress.com/Cyber-Physical-Systems-From-Theory-to-Practice/Rawat-Rodrigues-Stojmenovic/9781482263329.

[21] R. N. Charette, "This car runs on code," *IEEE Spectrum*, Feb. 2009. [Online]. Available: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

[22] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proc. IEEE*, vol. 95, no. 2, pp. 356–373, Feb. 2007.

[23] J. Gorzelany, "What to do (and how to find out) if your car is being recalled-updated," *Forbes*, 2014. [Online]. Available: http://www.forbes.com/sites/jimgorzelany/2014/10/23/what-to-do-if-your-car-is-being-recalled/.

[24] G. Nicolescu and P. J. Mosterman, Eds., *Model-Based Design for Embedded Systems*, Boca Raton, FL, USA: CRC Press, 2010.

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

[26] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Powertrain control verification benchmark," in *Proc. 17th Int. Conf. Hybrid Syst., Comput. Control*, 2014, pp. 253–262.

[27] X. Jin, J. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Benchmarks for model transformations and conformance checking," in *Proc. 1st Int. Workshop Appl. Verif. Continuous Hybrid Syst.*, 2014. [Online]. Available: http://alumni.cs.ucr.edu/~jinx/.

[28] J. V. Deshmukh *et al.*, "Piecewise affine approximations for a powertrain control benchmark," in *Proc. 2nd Int. Workshop Appl. Verif. Continuous Hybrid Syst.*, 2015, [Online]. Available: http://alumni.cs.ucr.edu/~jinx/.

[29] J. Fisher and T. A. Henzinger, "Executable cell biology," *Nature Biotechnol.*, vol. 25, no. 11, pp. 1239–1249, Nov. 2007.

[30] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *J. Comput. Syst. Sci.*, vol. 57, no. 1, pp. 94–124, 1998.

[31] G. Holzmann, *The Spin Model Checker*. Reading, MA, USA: Addison-Wesley, 2003.

[32] R. Kaivola *et al.*, "Replacing testing with formal verification in Intel Core TM i7 processor execution engine validation," in *Proc. 21st Int. Conf. Comput. Aided Verif.*, 2009, pp. 414–429.

[33] G. Klein *et al.*, "Sel4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010.

[34] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, no. 7, pp. 68–76, Jul. 2011.

[35] C. Newcombe *et al.*, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015.

[36] S. Mitra, T. Wongpiromsarn, and R. Murray, "Verifying cyber-physical interactions in safety-critical systems," *IEEE Security Privacy*, vol. 11, no. 4, pp. 28–37, Jul. 2013.

[37] S. Tripakis *et al.*, "Implementing synchronous models on loosely time-triggered architectures," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008.

[38] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 2, pp. 1–40, Feb. 2008.

[39] J. Fisher *et al.*, "Dynamic reactive modules," in *Proc. CONCUR*, 2011, pp. 404–418.

[40] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr. Inf. Process.*, 1974.

[41] W. de Roever, H. Langmaack, and A. E. Pnueli, *Compositionality: The Significant Difference*, Lecture Notes in Computer Science, Berlin, Germany: Springer-Verlag, 1998.

[42] J. Eker *et al.*, "Taming heterogeneity—The Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[43] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee, "A modular formal semantics for Ptolemy," *Math. Struct. Comput. Sci.*, vol. 23, pp. 834–881, Aug. 2013.

[44] R. Lublinerman and S. Tripakis, "Modularity vs. reusability: Code generation from synchronous block diagrams modularity," in *Proc. ACM Design Autom. Test Eur.*, Mar. 2008, pp. 1504–1509.

[45] R. Lublinerman and S. Tripakis, "Modular code generation from triggered and timed block diagrams," in *Proc. 14th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2008, pp. 147–158.

[46] R. Lublinerman C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams—Modularity vs. code size," in *Proc. 36th ACM SIGPLAN-SIGACT Symp. Principles Programm. Lang.*, Jan. 2009, pp. 78–89.

[47] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, pp. 83:1–83:26, Mar. 2013.

[48] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[49] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI Quart.*, vol. 2, pp. 219–246, 1989.

[50] M. Broy, "Compositional refinement of interactive systems," *J. ACM*, vol. 44, no. 6, pp. 850–891, 1997.

[51] R. Alur and T. Henzinger, "Reactive modules," *Formal Methods Syst. Design*, vol. 15, pp. 7–48, 1999.

[52] L. de Alfaro and T. Henzinger, "Interface automata," in *Foundations of Software Engineering (FSE)*, New York, NY, USA: ACM, 2001.

[53] L. de Alfaro and T. Henzinger, "Interface theories for component-based design," in *EMSOFT'01*, Lecture Notes in Computer Science, Berlin, Germany: Springer-Verlag, 2001, vol. 2211.

[54] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *ACM Trans. Programm. Lang. Syst.*, vol. 33, no. 4, pp. 14:1–14:41, Jul. 2011.

[55] V. Preoteasa and S. Tripakis, "Refinement calculus of reactive systems," in *Proc. 14th ACM/IEEE Int. Conf. Embedded Softw.*, 2014.

[56] I. Dragomir, V. Preoteasa, and S. Tripakis, "Compositional semantics and analysis of hierarchical block diagrams," in *23rd Int. SPIN Symposium on Model Checking of Software*, LNCS, Springer, 2016.

[57] S. Tripakis, "Bridging the semantic gap between heterogeneous modeling formalisms and FMI," in *Proc. Int. Conf. Embedded Comput. Syst., Architect. Model. Simul.*, 2015.

[58] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, S. Tripakis, M. Wetter, and M. Masin, "Determinate composition of FMUs for co-simulation," in *Proc. 13th ACM/IEEE Int. Conf. Embedded Softw.*, 2013.

[59] S. Bogomolov *et al.*, "Co-simulation of hybrid systems with SpaceEx and Uppaal," in *Proc. 11th Int. Modelica Conf.*, 2015. [Online]. Available: http://www.ep.liu.se/ecp_article/index.en.aspx?issue=118;article=017.

[60] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation With Modelica 3.3: A Cyber-Physical Approach*, 2nd. New York, NY, USA: Wiley, 2014.

[61] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM Symp. POPL*, 1977.

[62] P. Caspi *et al.*, "Guidelines for a graduate curriculum on embedded software and systems," *ACM Trans. Embedded Comput. Sys.*, vol. 4, no. 3, pp. 587–611, 2005.

[63] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. Int. Symp. Circuits Syst.*, 2000.

[64] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 30:1–30:39, May 2008.

[65] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *4th IEEE Int. Conf. Softw. Eng. Formal Methods (SEFM 2006)*, Sept. 11–15, 2006, Pune, India, pp. 3–12, 2006.

[66] M. Geilen, S. Tripakis, and M. Wiggers, "The earlier the better: A theory of timed actor interfaces," in *Proc. 14th Int. Conf. Hybrid Syst., Comput. Control*, 2011.

[67] J. Sztipanovits *et al.*, "Toward a science of cyber-physical system integration," *Proc. IEEE*, vol. 100, no. 1, pp. 29–44, Jan. 2012.

[68] T. A. Henzinger, "Two challenges in embedded systems design: Predictability and robustness," *Philosoph. Trans. Roy. Soc. Lond. A, Math. Phys. Eng. Sci.*, vol. 366, no. 1881, pp. 3727–3736, 2008.

[69] E. A. Lee, "Computing needs time," *Commun. ACM*, vol. 52, no. 5, pp. 70–79, May 2009.

[70] A. Sangiovanni-Vincentelli, "Quo vadis SLD: Reasoning about trends and challenges of system-level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

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

[72] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 404–415, Oct. 2006.

[73] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Design Implement.*, 2014.

[74] R. Alur *et al.*, "Synthesizing finite-state protocols from scenarios and requirements," in *Hardware and Software: Verification and Testing*Lecture Notes in Computer Science, Berlin, Germany: Springer-Verlag, 2014, vol. 8855, pp. 75–91.

[75] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Automatic completion of distributed protocols with symmetry," in *Proc. 27th Int. Conf. Comput. Aided Verif.*, 2015.

ABOUT THE AUTHOR

Stavros Tripakis received the Ph.D. degree in computer science from the Verimag Laboratory, Joseph Fourier University, Grenoble, France, in 1998.

He is an Associate Professor at Aalto University, Espoo, Finland and an Adjunct Associate Professor at the University of California at Berkeley, Berkeley, CA, USA. He was a Postdoctoral Researcher at the University of California Berkeley from 1999 to 2001, a CNRS Research Scientist at Verimag from 2001 to 2006, and a Research Scientist at Cadence Research Labs, Berkeley, from 2006 to 2008. His research interests include formal methods, computer-aided system design, and cyber-physical systems.

Dr. Tripakis was Co-Chair of the 10th ACM & IEEE Conference on Embedded Software (EMSOFT 2010), and Secretary/Treasurer (2009–2011) and Vice-Chair (2011–2013) of ACM SIGBED. His h-index is 40 (Google Scholar).