INVITED PAPER

System Design Automation: Challenges and Limitations

This paper discusses to what extent the VLSI-design paradigm can be transposed to hardware/software systems that interact continuously with an external environment, through the application of the principles of separation of concerns, component-based design, semantic coherency, and correctness by construction.

By Joseph Sifakis

ABSTRACT | Electronic design automation (EDA) has enabled the integrated circuit industry to sustain exponentially increasing product complexity growth until today, while maintaining consistent product development timeline and costs. We argue that the success of EDA-based design relies on the application of four interrelated principles: 1) separation of concerns implying a decomposition of a design flow into steps, each step dealing with specific aspects, namely user requirements, functional design, and implementation; 2) component-based design enabling the reasoned construction of complex systems as the composition of components; 3) semantic coherency meaning that descriptions used in successive design steps are semantically related through adequate semantic mappings; this implies, in particular, that the formalisms used at each design step are rooted in well-defined semantics; and 4) correctness by construction meaning that it is possible to guarantee essential properties of the designed system incrementally and compositionally along the design process. The paper discusses to what extent the EDA paradigm can be adapted to general mixed hardware/software (HW/SW) systems design through the application of these principles. It presents an overview of the problems raised by the rigorous system design of mixed HW/SW systems. Then, it presents a unified abstract framework for addressing these problems by identifying main research avenues.

KEYWORDS | Computer-aided engineering; computer-aided software engineering; design automation; design methodology; system software; systems engineering

Manuscript received October 22, 2014; revised June 9, 2015 and September 22, 2015; accepted September 25, 2015. Date of publication October 15, 2015; date of current version October 26, 2015.
The author is with the Rigorous System Design Laboratory (RiSD), École polytechnique fédérale de Lausanne (EPFL), Lausanne 1015, Switzerland (e-mail: joseph.sifakis@epfl.ch).

Digital Object Identifier: 10.1109/JPROC.2015.2484060

I. INTRODUCTION

Electronic design automation (EDA) has enabled the integrated circuit industry to sustain exponentially increasing product complexity growth until today, while maintaining consistent product development timeline and costs. We argue that the success of EDA-based design relies on the application of four interrelated principles.

1) Separation of concerns implying a decomposition of the design flow into steps, each step dealing with specific aspects, namely high-level synthesis, logic synthesis, schematic capture, and layout. Such a modular decomposition is essential for coping with complexity, in particular by decoupling functional design from implementation.
2) Component-based design enabling the reasoned construction of complex systems as the composition of components. Fabricators generally provide libraries of types of components for their production processes, with simulation models that fit standard simulation tools. Complex system models can be obtained as the composition of standardized components and validated before implementation.
3) Semantic coherency meaning that descriptions used in successive design steps are related through adequate semantic mappings. This is essential for ensuring traceability along the design flow and understanding how choices at some step may impact design properties in subsequent steps.
4) Correctness by construction meaning that it is possible to guarantee essential properties of the designed system incrementally and compositionally along the design process, especially by using architectures. Designers reuse reference architectures that ensure by construction essential functional and extra-functional properties.

0018-9219 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

We discuss to what extent the EDA paradigm can be transposed to general mixed hardware/software (HW/SW) systems design, in particular through the application of these principles.

Systems can be described by models that specify how the functionality of their software is implemented by using resources of an execution platform, such as time, memory, and energy. A key issue is building faithful system models from models of their application software equipped with variables representing resources and their dynamics [1]. This need is well understood and has motivated the development of research on cyber–physical systems addressing various issues including modeling [2] and design [3].

System design is the process leading from a set of requirements to a system meeting these requirements. It can be decomposed into two main steps: 1) writing an application software satisfying functional (platform-independent) requirements; and 2) implementing the software on a given execution platform.

System design can be studied as a model-based formal systematic process supported by a methodology. The latter should be based on divide-and-conquer strategies involving a set of steps leading from requirements to an implementation. At each step, a particular humanly tractable problem must be solved by addressing specific classes of requirements. The methodology should clearly identify segments of the design process that can be supported by tools to automate tedious and error-prone tasks. It should also clearly distinguish points where human intervention and ingenuity are needed to resolve design choices through requirements analysis and confrontation with experimental results. Identifying adequate design parameters and channeling the designer's creativity are essential for achieving design goals.

System design methodologies should propose strategies for overcoming the following theoretical obstacles and limitations.

1) Requirements formalization: Despite progress in formalizing requirements over the past decades (e.g., by using temporal logics), we still lack methods and tools for rigorous requirements specification.
2) Intractability of synthesis/verification: Very large scale integration (VLSI) design has largely benefited from fully automated synthesis/verification techniques. Unfortunately, such techniques become intractable for software that has a potentially infinite number of states. Program synthesis from abstract specifications and program verification do not admit exact algorithmic solutions. For system design, correctness should be sought beyond the synthesis/verification paradigm, as explained in Section VI.
3) HW/SW interaction: We need theory and methods for predicting precisely the behavior of some given software running on a hardware platform with known characteristics. This difficulty lies in the fundamental difference between hardware and software. Software models ignore physical time and resources while hardware behavior is bound to timing and resource constraints. Program execution dynamics inherits hardware-dynamic properties that cannot be precisely characterized or estimated due to inherent uncertainty and the resulting unpredictability.

We present a vision for rigorous system design as an accountable model-based process for building systems of guaranteed quality cost effectively. This vision is substantiated by a unifying framework, for discussing important trends in system design and tackling relevant research challenges. It has been influenced by Sangiovanni-Vincentelli's seminal ideas about "platform-based design" [4]. A detailed account of existing work in the area is beyond the scope of this paper. Several of the mentioned challenges and techniques have been discussed, addressed, and to some extent implemented in tools and methodologies. The most prominent approaches and frameworks are the environments supporting heterogeneous models of computation for system design PtolemyII [5], MetroII [6], and OpenMeta [7]. The presented vision has been amply implemented in the behavior–interaction– priority (BIP) component framework [8] at Verimag (Grenoble, France) and corroborated by numerous experimental results showing both its relevance and feasibility.

Section II discusses the concept of system correctness and provides a rationale for rigorousness. In the subsequent sections, we study how the four principles of EDA-based design can be applied to systems. We show significant differences and discuss their impact on design automation.

We propose a single model-based framework in which the application of the principles yields well-defined research problems. Section VII presents final remarks about the nature of system design and its importance.

II. RIGOROUS SYSTEM DESIGN

A. Concept of System Correctness

The concept of system correctness differs from pure function software correctness in many respects as it encompasses both functional and extra-functional requirements. It conjoins two types of requirements: 1) trustworthiness ensuring that nothing bad would happen; and 2) optimization for performance, cost effectiveness, and tradeoffs between them.

We consider trustworthiness to mean that the system can be trusted, and that it will behave as expected despite: a) software design and implementation errors; b) failures of the execution infrastructure; c) interaction with potential users including erroneous actions and threats; and d) interaction with the physical environment including disturbances and unpredictable events.

Note that the term "trusted computing" has been promoted by several software vendors such as Microsoft and Sun Microsystems, which primarily focus on security [9], [10]. Nonetheless, it is generally admitted that trustworthiness is a much broader concept that should be viewed as a holistic system property, encompassing both security and safety [11]–[14]. One confusing aspect is that our confidence in systems is sometimes based on both the artefact itself and on the humans who deliver it [15]. Therefore, many approaches that focus on computational trust models in different domains like sociology and psychology have been proposed [16], [17].

The development of trustworthy systems has given rise to an abundant literature, including research papers and reports as well as a plethora of research projects. There is no generally accepted definition of the concept to date. Existing approaches either focus on properties that can be formalized and checked effectively, e.g., by using formal methods [18] or are addressing a very broad spectrum of mostly unrelated topics; see, e.g., [19].

Some authors use the term dependability instead of trustworthiness. This term coined by the "fault-tolerance" community [13] characterizes a measure of a system's availability, reliability, and its maintainability. Dependability does not cover security aspects. Another important difference with trustworthiness is that the latter can be defined in a purely qualitative manner.

The proposed definition considers trustworthiness as a global property that must be addressed throughout the computing environment. Among the four types of hazards, only software design errors are born in software. The others are born in the system as a whole in interaction with its physical and human environment.

Optimization requirements deal with optimizing functions subject to constraints on resources such as time, memory, and energy, dealing with: 1) performance which characterizes how well the system does with respect to user demands concerning quantities such as throughput, jitter, and latency; 2) cost effectiveness which characterizes how well resources are used with respect to economic criteria such as storage efficiency, processor load/availability, and energy efficiency; and 3) tradeoffs between performance and cost effectiveness.

Note that our definition of optimization requirements leaves out minimization of parameters that characterize physical characteristics of a given execution platform, e.g., weight and length of cables. In the studied approach, we consider that the designer may choose different execution platforms as they are.

Trustworthiness requirements determine the set of legal states—a state may be trustworthy or not. They may be functional as well as extra-functional. Optimization requirements characterize sets of execution sequences. These are usually integral constraints over sequences, e.g., minimize energy or maximize throughput for some time period. For systems, they are mostly extra-functional requirements, in contrast to platform-independent optimization requirements applied to application software.

Trustworthiness and optimization requirements are difficultly reconcilable. As a rule, improving trustworthiness entails nonoptimized use of resources. Conversely, resource optimization may jeopardize trustworthiness. For example, if resource utilization is pushed to the limits, deadlocks may occur; enhancing trustworthiness by massive redundancy costs extra resources. Designers should seek trustworthiness and try to optimize resources at the same time. A key design issue is ensuring trustworthiness without disregarding optimization.

B. Rationale for Rigorousness

For decades, formal methods and verification in particular have been considered as the main avenues for achieving correctness. Despite spectacular progress, verification techniques suffer from inherent well-known limitations [20]. One comes from our inability to apprehend and formally express user's needs by requirements for complex systems. Another stems from inherent theoretical limitations of verification techniques such as model checking, abstract interpretation, and static analysis. These are applied to global system models whose state space size increases exponentially with the number of constituent components. Attempts to apply compositional verification to component-based systems, such as assume/guarantee techniques, failed to make any significant breakthrough [21].

Formal verification hides a much more important challenge which is faithful modeling of systems. Whatever is the progress of the state of the art in verification, a central problem is how to build faithful models, in particular of mixed HW/SW systems. There are very simple systems, e.g., the node of a wireless sensor network, that we do not know how to model faithfully, in particular to verify extra-functional properties.

In conclusion, formal verification can be applied to systems whose criticality justifies relatively high development costs. Typical examples are purely software components that are extensively used such as operating systems and compilers. For interactive HW/SW systems, it is much more difficult to formalize their requirements and come up with faithful models.

The application of verification to resource optimization requirements is limited to the validation of scheduling and resource management policies on abstract system models.

We advocate a shift of focus from formal methods to rigorous system design. The key motivation is to overcome current limitations through the application of EDA-based principles. In particular, we put emphasis on correctness by construction. The idea is as simple as that of constructive proof in Euclidean geometry. When designers use algorithms, architectures, and protocols that have been proven correct, they do not need any additional proof of correctness if they have an adequate methodology for doing this.

A rigorous system design flow is a formal accountable and iterative process for deriving trustworthy and optimized implementations from application software and models of its execution platform and its external environment.

Rigorous system design is model based: successive system descriptions are obtained by application of correct-by-construction source-to-source model transformations. Furthermore, accountability implies the obligation of account-giving behavior in the design process. It covers two aspects: 1) the designer provides evidence that his choices are motivated by the need to meet properties implied by requirements; and 2) for already established properties, the designer guarantees that they still hold in subsequent stages.

In practice, accountability can be eased through the use of assurance case methodologies. These allow structuring the designer's reasoning to gain confidence that systems will work as expected. Assurance cases have been applied in practice to present the support for claims about properties or behaviors of a system [22]–[24]. The focus is not on formalization but rather on systematizing the connection between requirements and design choices. Correctness-by-construction techniques discussed in Section VI provide a basis for accountability.

Currently there exist a few rigorous system design approaches dedicated to the development of safety critical real-time systems. Some are based on the synchronous programming paradigm [25]. Others are represented mainly by flows based on the ADA standard [26].

III. SEPARATION OF CONCERNS

Separately addressing functional from extra-functional requirements is essential from a methodological point of view. This also identifies two main gaps in a design flow. First, application software is developed that should be correct with respect to the functional requirements. Then, a correct implementation meeting both functional and extra-functional requirements should be derived by progressive refinement of the application software taking into account features of the execution platform. This idea is commonly adopted in many system design methodologies. The model-driven architecture (MDA) approach developed by the Object Management Group (OMG) [27] introduces a set of models to successively describe requirements, the application software, and platform-specific models intended to represent the dynamic behavior of the application software on a given execution platform. MDA is related to multiple standards including UML. Today MDA is rather a conceptual framework. Similar separation of concern principles is followed and effectively implemented in frameworks such as model-integrated computing (MIC) [28], MetroII [6], Ptolemy [5], and BIP [8].

In the following, we discuss two issues. One is providing support for system programming to ease the transition from requirement specifications; the other deals with developing faithful system models from application software.

A. Domain-Specific Languages

In system software development, the key issue is managing the complexity resulting from interactions with the environment and among various subsystems. Using general purpose programming languages, such as C or Java, may be counterproductive and error prone. These languages are adequate mainly for sequential transformational programs computing functions.

For systems modeling, we need powerful primitives encompassing direct description of different types of synchronization. Problems that have straightforward solutions by using automata-based formalisms are hard to tackle by using standard programming languages. For instance, programming communicating automata in Java may involve several technical difficulties because of intricate thread semantics and semantic variations of the wait/notify mechanism. As another example, consider a system of n components that are synchronized by rendezvous. If a description formalism offers only a single synchronization primitive, broadcast (weak synchronization), it is extremely hard to model strong synchronization. This is extensively discussed in Section IV-B

To enhance software productivity and safety, system designers are provided with domain-specific languages (DSLs) dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

Synchronous programming languages such as SCADE and Matlab/Simulink are widely used in the development of safety-critical control systems. Data-flow programming models are advantageously used to develop multimedia applications. They allow explicit description of task parallelism and allow schedulability analysis. Other DSLs are the various derivatives of UML such as MARTE and SysML. Nonetheless, these frameworks comprise a very large number of concepts and primitives and are poorly suited to formalization.

B. Building Faithful System Models

In a model-based design approach, implementations should be derived from a system model which faithfully describes the dynamic behavior of the application software on the execution platform. A key idea is to progressively apply source-to-source transformations to the application software which: 1) refine atomic statements to express them as sequences of primitives of the execution platform; 2) express synchronization constraints induced by the resources on the refined actions; and 3) associate with the refined actions parameters representing the resources needed for their execution (e.g., execution times).

System models have, in addition to the variables of the application software, state variables representing

resources. These variables are subject to two types of antagonistic requirements:

- user-defined requirements dealing with reaction times, throughput and cost, such as deadlines, periodicity, memory capacity, and power or energy limitations;
- platform-dependent requirements dealing with the amount of resources needed for executing actions such as execution times and energy consumption.

When an action is executed, resource variables are updated accordingly, in addition to software variables. Thus, states of system models are valuations of both the application software variables and of resource variables. Any execution sequence of a system model corresponds modulo some adequate abstraction criterion, to a sequence of its application software.

Building faithful system models is still an unexplored and ill-understood problem. Clearly, owing to a lack of predictability, system models can only approximate the behavior of the real systems they represent. As it is impossible to precisely estimate the amount of resources needed for the execution of an action from a given state, exact values are replaced by bounds. For example, computing tight estimates of worst case execution times is a hard problem that requires: 1) faithful modeling of the hardware and features such as instruction pipelines, caches, memory hierarchy, etc.; and 2) symbolic analysis techniques based on static analysis and abstract interpretation. Due to theoretical limitations, the latter can compute only safe approximations of these bounds.

An additional difficulty is that incremental and parallel modification of resource variables in a model should be consistent with physical laws governing resources. For instance, physical time is steadily increasing while in system models time progress may stop, block, or may involve Zeno runs [29]. This is a significant difference between model time and physical time. A typical example of model time is that of time in simulation programs that explicitly handle its progress.

As physical time progress cannot be blocked, deadline misses occurring in the actual system correspond to deadlocks or time locks in the relevant system model. Similarly, lack of sufficient resources is reflected in system models by the inability to execute actions. These observations lead to the notion of feasibility of system models. System model feasibility and associated analysis techniques deserve thorough study [30].

A basic principle widely used in all areas of engineering is that performance changes monotonically with resources. Typically for a building, enhanced mechanical resistance is achieved by increasing the strength of the materials of its components. Consequently, analysis for worst case and best case values of resource parameters suffices to determine performance bounds. Unfortunately, for systems, the intuitive idea that safety of implementation is preserved for increasing performance turns out to be wrong. This phenomenon called timing anomaly [31] limits our capability to analyze system model feasibility. A direct consequence of timing anomalies is that safety for worst case execution time (WCET) does not guarantee safety for smaller execution times. As shown in [30], timing anomalies may appear in nondeterministic systems. Avoidance of timing anomalies advocates for approaches guaranteeing determinism by construction [32].

IV. COMPONENT-BASED DESIGN

A. Component Heterogeneity

Using components throughout a system design flow is essential for enhanced productivity and correctness. Any tractable theory for component-based construction requires a relatively small number of types of components. Electrical engineers apply Kirchhoff's laws that distinguish between four different types of components. Similarly, a hardware engineer designs processors out of memories, ALUs, buses, multiplexers, etc. Currently, system designers deal with heterogeneous components, with different characteristics, from a large variety of viewpoints, each highlighting the various dimensions of a system. This contrasts with standard engineering practices based on the disciplined composition of a limited number of types of components.

There exist a large number of component frameworks, including software component frameworks, systems description languages, and hardware description languages. Currently, there is no agreement on a common concept of component. This is mainly due to heterogeneity of components and their associated composition operations. There exist various sources of heterogeneity [1]. Hardware components and some application software components are synchronous, while general purpose software components are asynchronous. Furthermore, engineers use a variety of mechanisms to ensure component coordination including semaphores, rendezvous, broadcast, method call, buses etc. The difference between actor-based and thread-based programming is an additional source of heterogeneity. In actor models, components have no shared memory and operate on message passing. In thread-based models, components may share resources used by multiple threads. The actor paradigm is safer as it enforces a discipline for structuring coordination between components with disjoint state spaces. As a rule, thread-based models are not composable and require an explicit management of shared resources that may introduce deadlocks and races [33]. Nonetheless, the advantages conferred by actors are largely reduced due to the lack of efficient code generation techniques [34].

Is it possible to express component coordination in terms of composition operators? We need a unified composition paradigm based on operational semantics to describe and analyze the coordination between components

in terms of tangible, well-founded, and organized concepts characterized by:

1) orthogonality: clear separation between behavior and coordination constraints;
2) minimality: minimal set of primitives;
3) expressiveness: achievement of a given coordination with a minimum of mechanism and a maximum of clarity.

Existing theoretical frameworks for composition are based on a single operator (e.g., product of automata, function call). Poor expressiveness of these frameworks may lead to complicated designs: achieving a given coordination between components often requires additional components to manage their interaction [35]. For instance, if the composition is by strong synchronization (rendezvous), modeling broadcast requires components for choosing the maximal among several possible strong synchronizations.

Most component composition frameworks fail to strictly separate behavior from coordination. In most programming languages as well as in various process algebras cloned from CCS, coordination is expressed in terms of communication primitives spread across component behavior. Architecture description languages, e.g., [36] and [37], advocate separation of concerns between behavior and coordination while they allow the use of behavior in connectors.

B. Component Frameworks

We summarize here key ideas overarching the unified composition paradigm that has been applied in the development of the BIP framework [8].

A component framework consists of a set of atomic components $B = \{B_i\}_{i \in I}$ and a glue $GL = \{gl_k\}_{k \in K}$, a set of glue operators on these components. Atomic components are characterized by their behavior specified as a transition relation involving actions, e.g., by operational semantics. A glue operator gl is a memoryless composition operator. The meaning of gl is specified using a class of structured operational semantics rules that only restrict the behavior of the composed components. The technical definition [24] gives the behavior of a composite component $gl(C_1, \ldots, C_n)$ as a partial function of transition relations of the composed components $C_1, \ldots, C_n$. Restriction is either through multiparty interaction (synchronous execution of actions from distinct components) or through influence (an action of a component is allowed depending on the state of other components). A practically useful case of influence is by using priorities.

A component framework can be considered as the algebra of well-formed terms built from a set of atomic components equipped with a congruence relation $\approx$ compatible with strong bisimulation on transition systems. It can be shown that a component framework enjoys the following flattening property [35]: if a composite component is of the form $gl_1(C_1, gl_2(C_2, \ldots, C_n))$, then there exists an operator gl such that $gl_1(C_1, gl_2(C_2, \ldots, C_n) \approx gl(C_1, C_2, \ldots, C_n)$. This property is essential for separating behavior from glue and treating glue as an independent entity that can be studied and analyzed separately [38].

The comparison between different formalisms and models is often made by flattening their structure and reducing them to behaviorally equivalent models (e.g., automata, Turing machine). This leads to a notion of expressiveness which is not adequate for the comparison of high-level languages. All programming languages are deemed equivalent (Turing complete) without regard to their adequacy for solving problems. For component frameworks, separation between behavior and coordination mechanisms is essential.

A notion of expressiveness for component frameworks characterizing their ability to coordinate components is proposed in [35]. It allows the comparison of two component frameworks with glues GL and GL', respectively, the same set of atomic components and equipped with the same congruence relation $\approx$.

We say that GL' is more expressive than GL if for any composite component $gl(C_1, \ldots, C_n)$ obtained by using $gl \in GL$ there exists $gl' \in GL'$ such that $gl(C_1, \ldots, C_n) \approx gl'(C_1, \ldots, C_n)$. That is, any coordination expressed by using GL can be expressed by using GL'. Such a definition allows a comparison of glues characterizing coordination mechanisms. The glue GL is universally expressive if it can express any coordination achieved by using glue operators.

The interested reader may find in [35] a comparison of existing formalisms according to this notion of expressiveness and also a weaker (less discriminating) notion. The main result is that universal expressiveness can be achieved by combining two types of glue operators: 1) multiparty interaction; and 2) priorities. It has motivated the development of the BIP framework. Any language that does not directly support these primitives is less expressive. Thus, to solve arbitrary coordination problems, additional coordinator components are needed.

A taxonomy of existing component frameworks based on their expressiveness would allow a rigorous comparison of their coordination languages, in particular of their respective merits and their deficiencies.

V. SEMANTICALLY COHERENT DESIGN

System designers use multilanguage frameworks integrating programming and modeling languages. Most of the languages lack formal operational semantics, their meaning being defined by user manuals and their supporting tools. Quite often, system programming and implementation tasks do not take into account models established for validation evaluation purposes. Using semantically unrelated languages in a design flow breaks continuity of activities and may jeopardize its overall coherency.

To maintain the overall coherency of a design flow, all the used formalisms should be interrelated through a

common model. For instance, design flows may involve DSLs based on different models of computation. Coherency of design flows can be achieved by translating all these formalisms into a common expressive language with well-defined operational semantics.

Of course, other types of semantics may be used to define the meaning of the host language, especially because they may lead to more elegant and concise formalizations. Nonetheless, ultimately operational semantics is absolutely necessary to define correct implementation.

The necessity of using a common host language to serve as meta-language for embedding DSLs is widely recognized, e.g., [39]. The proposed notion of embedding is inspired from these works and takes advantage of the assumption that the host language is more expressive than the source language. A similar idea originated out of Passerone's dissertation work dealing with defining common semantic domains for heterogeneous models of computation [40], [41]. The main difference is that the host language and thus the resulting embedding is a choice of the designer.

Note that an important trend in industrial flows is the translation of the various languages used by the designers, in particular of DSLs into a single host language, namely C for embedded applications or Java for web-based ones. Of course, these translations are done in some *ad hoc* manner. Furthermore, they are not embeddings as the structure of the source language is usually lost in the translation by flattening. We advocate structure-preserving translations that require expressive host languages.

An embedding is a semantics-preserving and structure-preserving mapping between two component-based languages rooted in operational semantics. Consider two component-based languages H and L with well-defined operational semantics. We assume that the terms (programs) of these languages can be compared through a common semantic congruence $\approx$ and require that H is more expressive than L. An embedding of L into H is defined as a two-step transformation involving functions χ and σ, respectively.

The first step defines a homomorphism that fully preserves the structure of the translated language. It takes into account the "programmer's view" of the language by translating all the coordination primitives explicitly manipulated by the programmer. It consists in transforming a term t of L into a term $\chi(t) \in H$. The function χ is structure preserving. It associates with components and glue operators of L, components and glue operators of H so that: 1) if B is an atomic component of L, then $\chi(B)$ is an atomic component of H; 2) $\chi(t) = \chi(gl)(\chi(C_1), \ldots, \chi(C_n)) \in H$, for any term $t = gl(C_1, \ldots, C_n) \in L$.

The second step adds the glue and the behavior needed to orchestrate the execution of the translated component $\chi(t)$, by respecting the semantics of L. It consists in transforming a term $t \in L$ into a term $\sigma(t) \in H$, by using a semantics-preserving function σ. The function σ can be expressed by using two auxiliary functions σ_1 and σ_2 associating, respectively, with any term $t \in L$ its semantic glue $\sigma_1(t)$ and an execution engine $\sigma_2(t)$ both expressed in H, so that $\sigma(t) = \sigma_1(t)(\chi(t), \sigma_2(t)) \approx t$.

This constructive definition involves separate translation of the coordination mechanisms explicitly handled by the programmer from additional coordination mechanisms implied by the operational semantics of the source language.

Fig. 1 illustrates the principle. On the left, the software written in L is a set of components with their glue. The structured operational semantics of L defines an execution engine that coordinates the execution of components as specified by the glue. Embeddings preserve the structure of the source: atomic components of L are translated into atomic components of H with additional ports used by the component representing the execution engine of L in H.

Embedding real-life languages is a nontrivial task as it requires a formalization of their intuitive semantics. The interested reader can find in [8] papers dealing with the definition and implementation of embeddings into BIP for languages such as nesC, DOL, Lustre, and Simulink.

Embedding languages for modeling cyber–physical systems involves an additional difficulty: the definition of precise enough operational semantics coping with the inherent complexity of the synchronous execution mechanisms required.

VI. CORRECTNESS BY CONSTRUCTION

Correct-by-construction approaches are at the root of any mature engineering discipline. They are scalable and do not suffer limitations of correctness by checking.

System developers extensively use algorithms, protocols, and architectures that have been proven to be correct. They also use compilers to get across abstraction

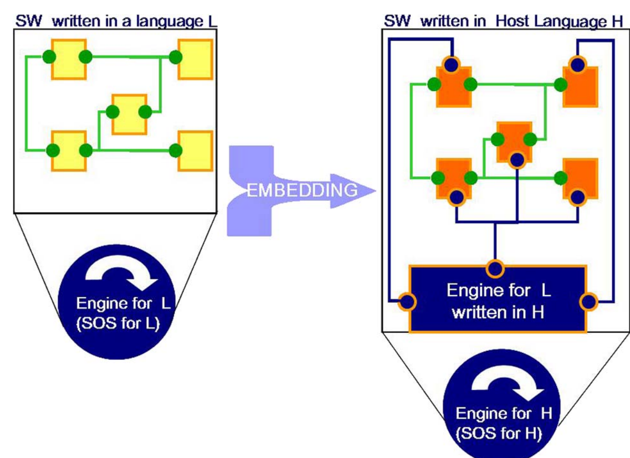

Fig. 1. ***Embedding language L into the host language H.***

levels and translate high-level languages into (semantically equivalent) object code. All of these results and techniques largely account for our ability to master complexity and develop systems cost effectively. Nonetheless, we still lack theory and methods for combining them in principled and disciplined fully correct-by-construction flows.

We propose a methodology to ensure correctness by construction gradually throughout the design process by acting in two different directions:

- horizontally, within a design step, by providing rules for enforcing global properties of composite components (horizontal correctness) while preserving essential properties of atomic components;
- vertically, between design steps, to guarantee that if some property is established at some step then it will be preserved at all subsequent steps (vertical correctness).

Horizontal correctness and vertical correctness stem from the need to distinguish between correctness within a design step and the relationship of established properties between steps. This distinction is extensively applied in contract-based methodologies, e.g., [42].

A. Horizontal Correctness

Horizontal correctness addresses the following problem: for a given component framework with set of atomic components $B = \{B_i\}_{i \in I}$ and glue $GL = \{gl_k\}_{k \in K}$, build a component C meeting a given property P, from a set of atomic components $B_1, \ldots, B_n$ of B.

The construction process of component C is bottom-up. Increasingly complex composite components are built from atomic components by using glue operators. Two principles can be used in this process to obtain a component meeting P: property enforcement and property composability.

1) Property Enforcement: Property enforcement consists in applying architectures to coordinate the behavior of a set of components so that the resulting behavior meets a given property. Depending on the expressiveness of the glue operators, it may be necessary to use additional components to achieve the coordination meeting the property.

Architectures depict design principles and paradigms that can be understood by all, and allow thinking on a higher plane and avoiding low-level mistakes. They are a means for ensuring global properties characterizing the coordination between components. Using architectures is key to ensuring trustworthiness and optimization in networks, OS, middleware, HW devices, etc.

System developers extensively use libraries of reference architectures enforcing functional and/or nonfunctional properties, for example, fault-tolerant architectures, architectures for resource management and quality of service (QoS) control, time-triggered architectures, security architectures, and adaptive architectures. The proposed definition is general and can be applied not only to hardware or software architectures but also to protocols and distributed algorithms [43].

An architecture [44] is a family of operators on components $A(n)[X]$ parameterized by their parity n and a family of characteristic properties $P(n)$ such that:

- $A(n)$ transforms a set of components $C_1, \ldots, C_n$ into a composite component $A(n)[C_1, \ldots, C_n] = gl(n)(C_1, \ldots, C_n, D(n))$ where $gl(n)$ is a generic glue operator and $D(n)$ is a set of coordinating components;
- $A(n)[C_1, \ldots, C_n]$ meets the characteristic property $P(n)$.

Architectures are partial operators as the interactions of gl should match actions of the composed components. They are solutions to a coordination problem characterized by P. The desired coordination is achieved by applying the glue operator gl to the set of the arguments augmented with coordinating components D.

For instance, in distributed architectures, interactions are point to point by asynchronous message passing. Other architectures adopt a specific topology (e.g., ring architectures, hierarchically structured architectures). These restrictions entail reduced expressiveness of the glue operator gl that must be compensated by using the additional set of components D for coordination. The characteristic property assigns a meaning to the architecture that can be informally understood without the need for explicit formalization. Typically, a client–server architecture guarantees atomicity of transactions and fault-tolerance properties.

2) Property Composability: In a design process it is often necessary to combine more than one architectural solution on a given set of components to achieve a global property. System engineers use libraries of solutions to specific problems and they need methods for combining them without jeopardizing their characteristic properties. For example, a fault-tolerant architecture combines a set of features building into the environment protections against trustworthiness violations. These include: 1) triple modular redundancy mechanisms ensuring continuous operation in case of single component failure; 2) hardware checks to be sure that programs use data only in their defined regions of memory, so that there is no possibility of interference; and 3) default to least privilege (least sharing) to enforce file protection. Is it possible to obtain a single fault-tolerant architecture consistently combining these features? The key issue here is interference of the integrated solutions. This is a very common phenomenon known as feature interaction in telecommunication systems, interference among web services, and interference in aspect programming. It can be understood as a violation of property composability defined below.

Consider two architectures A_1 and A_2, enforcing, respectively, properties P_{A1} and P_{A2} on a set of components

$C_1, \ldots, C_n$. That is, $A_1[C_1, \ldots, C_n]$ and $A_2[C_1, \ldots, C_n]$ satisfy, respectively, the properties P_{A1} and P_{A2}. Is it possible to find an architecture $A(C_1, \ldots, C_n)$ that meets both properties? For instance, if A_1 ensures mutual exclusion and A_2 enforces a scheduling policy, is it possible to find architectures on the same set of components that satisfies both properties?

Results for composability of safety properties can be found in [44] which provides a method for computing an architecture $A_1 \oplus A_2$ enforcing two safety properties P_{A1} and P_{A2}, from two architectures A_1 and A_2, enforcing P_{A1} and P_{A2}, respectively.

To put this vision for horizontal correctness into practice, we need to develop a repository of reference architectures classified according to their characteristic properties. A list of these properties can be established; for instance, architectures for mutual exclusion, time triggered, security, fault tolerance, clock synchronization, adaptive, scheduling, etc. Is it possible to find a hierarchical classification of architectures induced by a hierarchy of characteristic properties? Moreover, is it possible to determine a minimal set of basic properties and corresponding architectural solutions from which more general properties and their corresponding architectures can be obtained?

B. Vertical Correctness

Moving downwards in the abstraction hierarchy requires component refinement. This can be achieved by transforming a composite component $C = gl(C_1, \ldots, C_n)$ into an refined component C' by using an architecture A, $C' = A[C'_1, \ldots, C'_n] = gl'(C'_1, \ldots, C'_n, D)$, preserving correctness of the initial system modulo some observation criterion.

This transformation is by refining the actions of components $C_1, \ldots, C_n$ to obtain new components $C'_1, \ldots, C'_n$. Action refinement in some component C_i consists in replacing an action a by its implementation as a sequence of actions $str(a) \ldots cmp(a)$. The first and the last element of this sequence correspond, respectively, to the start and the completion of the refined action. Action refinement also induces a refinement of the state space of the initial components: new state variables are introduced to control the execution of the refined actions. The glue operator gl' includes interactions involving refining actions. It contains in particular for each interaction a of gl, interactions $str(a)$ and $cmp(a)$ corresponding to the start and the completion of a.

An instance of this problem is finding a distributed implementation for a system $gl(C_1, \ldots, C_n)$ where gl specifies multiparty interactions between components. In that case, gl' includes only point-to-point interactions implementing asynchronous message passing coordinated by an additional set of components D. These contain memory where the exchanged messages are queued. Atomic actions of the initial components are refined by sequences of send/receive actions implementing a protocol.

We say that $C' = gl'(C'_1, \ldots, C'_n, D)$ refines $C = gl(C_1, \ldots, C_n)$, denoted by $C \geq C'$, if:

- all traces of C' are traces of C modulo the observation criterion associating to each interaction of C the corresponding finishing interaction of C';
- if C is deadlock free, then C' is deadlock free;
- $\geq$ is compatible with the congruence relation $\approx$. That is, for any components C_1, C_2, and C, if $C_1 \approx C_2$ and $C_1 \geq C$, then $C_2 \geq C$;
- $\geq$ is stable under substitution that is for any components C_1 and C_2 and any architecture A, $C_1 \geq C_2$ implies $A[C_1, X] \geq A[C_2, X]$ where X is an arbitrary tuple of components.

In this definition, the second condition guarantees preservation of deadlock freedom and precludes emptiness of the set of the traces of C'. Stability of $\geq$ under substitution is essential for reusing refinements and correctness by construction. As a rule, proving refinements requires nontrivial inductive reasoning on the structure of the terms representing systems.

The top of Fig. 2 depicts, in the form of a Petri net, a refinement which associates to $gl(C_1, C_2)$ the system $gl'(C'_1, C'_2, D)$ where gl consists of a single interaction a and gl' consists of the interactions $str(a)$ (start a), $rcv(a)$ (receive a), $ack(a)$ (acknowledge a), and $cmp(a)$ (complete a). So, the interaction a is refined by the sequence: $str(a)rcv(a)ack(a)cmp(a)$. The coordination component D contains two places for synchronization. The two systems are observationally equivalent for the criterion that considers as silent the interactions $str(a)$, $rcv(a)$, and $ack(a)$ and associates $cmp(a)$ with a.

Note that this type of refinement is not stable for substitution, as depicted in the bottom of Fig. 2. Two conflicting interactions a and b of the system on the left side are refined to obtain a system where a deadlock may occur when the transition $str(a)$ is executed.

In [45] and [46], refinement techniques are applied to generate correct implementations.

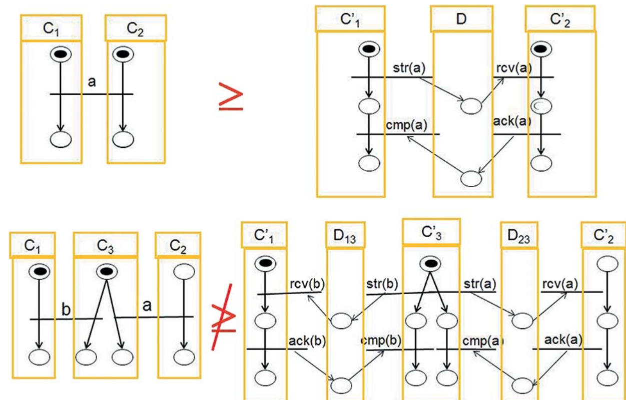

Fig. 2. *Interaction refinement by using send/receive primitives.*

To attain extra-functional correctness, designers need theory and methods for choosing, among different equally trustworthy designs, those better fitting the resources of the computing infrastructure. This is often achieved through the application of design space exploration techniques, e.g., [4]. Currently, these techniques are mostly experimental and consist in evaluating, on a system model, the impact of design parameters on optimization criteria.

VII. CONCLUSION

System design formalization raises a multitude of deep theoretical problems, including the conceptualization of needs and their expression as formal requirements, the development of functionally correct application software, and the optimized implementation on a given platform. So far, it has attracted little attention from scientific communities and is often relegated to second class status. This can be explained by several reasons. One is the predilection of the academic world for simple and elegant theories. Another is that system design is by nature multidisciplinary. Its formalization requires consistent integration of heterogeneous system models supporting different levels of abstraction, including logics, algorithms, and programs as well as physical system models.

We advocate system design as a process involving source-to-source correct-by-construction scalable transformations. Semantic coherency can be achieved using a single expressive component framework. The development of rigorous system design flows could leverage on the large body of existing "constructivity" results, e.g., algorithms, protocols, and architectures. A key idea is formalizing and composing architectures as a means for enforcing design properties. This allows correctness (almost) for free provided we develop an adequate composability theory.

Endowing system design with scientific foundations is a major scientific challenge. Even if the identified goals are reached, system design will never achieve the degree of automation of VLSI design; there will always remain gaps that can be bridged only by creative thinking and insightful analysis. ■

Acknowledgment

The author would like to thank the members of the behavior–interaction– priority (BIP) teams at Verimag, Grenoble, France and the École polytechnique fédérale de Lausanne (EPFL), Lausanne, Switzerland. He is also indebted to three anonymous reviewers for their comments and criticism that have drastically contributed to improving the paper.

REFERENCES

[1] T. A. Henzinger and J. Sifakis, "The discipline of embedded systems design," *Computer*, vol. 40, pp. 36–44, 2007.

[2] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 13–28, Jan. 2012.

[3] P. Bogdan and R. Marculescu, "Towards a science of cyber-physical systems design," in *Proc. IEEE/ACM 2nd Int. Conf. Cyber-Phys. Syst.*, 2011, pp. 99–108.

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

[5] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, and Y. Zhao, "Heterogeneous concurrent modeling and design in Java," Univ. California Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/ERL M05/21, 2005.

[6] A. Davare *et al.*, "MetroII: A design environment for cyber-physical systems," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1, Mar. 2013, Art. no. 49.

[7] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson, "OpenMETA: A model and component-based design tool chain for cyber-physical systems," in *From Programs to Systems—The Systems Perspective in Computing (FPS 2014)*. Grenoble, France: Springer-Verlag, 2014.

[8] Verimag, Rigorous design of component-based systems—The BIP component framework. [Online]. Available: http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html

[9] Microsoft, "Trusted computing group." [Online]. Available: http://www.trustedcomputinggroup.org

[10] Sun Microsystmes, "Liberty alliance." [Online]. Available: http://www.projectliberty.org

[11] L. Bernstein, "Trustworthy software systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 1, pp. 4–5, 2005.

[12] "The Second National Software Summit, Software 2015: A national software strategy to ensure U.S. security and competitiveness," U.S. Cntr. Nat. Softw. Studies, Tech. Rep., 2005.

[13] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan./Mar. 2004.

[14] L. J. Hoffman, K. L. Jenkins, and J. Blum, "Trust beyond security: An expanded trust model," *Commun. ACM*, vol. 49, no. 7, pp. 94–101, 2006.

[15] K. W. Miller and J. Voas, "The metaphysics of software trust," *IT Professional*, vol. 11, no. 2, pp. 52–55, 2009.

[16] D. H. Mcknight and N. L. Chervany, "The meanings of trust," in *Trust in Cyber-Societies*. Cambridge, MA, USA: MIT Press, 2001, pp. 27–54, ser. Lecture Notes in Artificial Intelligence.

[17] S. P. Marsh, "Formalising trust as a computational concept," Ph.D. dissertation, Univ. Stirling, Stirling, Scotland, 1994.

[18] M. Butler, M. Leuschel, S. L. Presti, and P. Turner, "The use of formal methods in the analysis of trust," in *Trust Management*, vol. 2995. Berlin, Germany: Springer-Verlag, 2004, pp. 333–339, ser. Lecture Notes in Computer Science.

[19] D. K. Mulligany and F. B. Schneider, "Doctrine for cybersecurity," Cornell Univ., Ithaca, NY, USA, Tech. Rep., May 2011.

[20] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *Commun. ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[21] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, 2008, Art. no. 7.

[22] The Adelard safety case development, ASCAD Manual Adelard, London, U.K., 1998.

[23] S. Cruanes, G. Hamon, S. Owre, and N. Shankar, "Tool integration with the evidential tool bus," in *Verification, Model Checking, and Abstract Interpretation*, vol. 7737, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds. Berlin, Germany: Springer-Verlag, 2013, pp. 275–294, ser. Lecture Notes in Computer Science.

[24] Software Engineering Institute, Carnegie Mellon University, "Assurance cases." [Online]. Available: http://www.sei.cmu.edu/dependability/tools/assurancecase/

[25] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA, USA: Kluwer, 1993.

[26] D. A. Watt, B. A. Wichmann, and W. Findlay, *Ada: Language and Methodology*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1987.

[27] Object Management Group (OMG), "Model Driven Architecture (MDA) Guide Revision 2.0 Document—Ormsc/14-06-01 (MDA Guide Revision 2.0)," 2014.

[28] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[29] M. Heymann, F. Lin, G. Meyer, and S. Resmerita, "Analysis of Zeno behaviors in a class of hybrid systems," *IEEE Trans.*

Autom. Control, vol. 50, no. 3, pp. 376–383, Mar. 2005.

[30] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, 2010, pp. 229–238.

[31] J. Reineke *et al.*, "A definition and classification of timing anomalies," in *Proc. 6th Int. Workshop Worst-Case Execution Time (WCET) Anal.*, Dresden, Germany, Jul. 4, 2006.

[32] E. A. Lee and S. Matic, "On determinism in event-triggered distributed systems with time synchronization," presented at the Int. IEEE Symp. Precision Clock Synchronization (ISPCS) Meas. Control Commun., Vienna, Austria, Oct. 1–3, 2007.

[33] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[34] M. Bozga, M. Jaber, and J. Sifakis, "Source-to-source architecture transformation for performance optimization in BIP," *IEEE Trans. Ind. Inf.*, vol. 6, no. 4, pp. 708–718, Nov. 2010.

[35] S. Bliudze and J. Sifakis, "A notion of glue expressiveness for component-based systems," in *CONCUR 2008—Concurrency Theory*, vol. 5201. Berlin, Germany: Springer-Verlag, 2008, pp. 508–522, ser. Lecture Notes in Computer Science.

[36] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proc. Conf. Centre Adv. Studies Collaborative Res.*, 1997, pp. 169–183.

[37] J. Magee and J. Kramer, "Dynamic structure in software architectures," in *Proc. 4th ACM SIGSOFT Symp. Found. Softw. Eng.*, 1996, pp. 3–14, ACM Press.

[38] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *Computer Aided Verification*, vol. 5643. Berlin, Germany: Springer-Verlag, 2009, pp. 614–619, ser. Lecture Notes in Computer Science.

[39] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky, "Scala-virtualized: Linguistic reuse for deep embeddings," *Higher-Order Symbolic Comput.*, vol. 25, no. 1, pp. 165–207, 2012.

[40] R. Passerone, "Semantic foundations for heterogeneous systems," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Univ. California Berkeley, Berkeley, CA, USA, May 2004.

[41] J. Burch, R. Passerone, and A. Sangiovanni-Vincentelli, "Refinement preserving approximations for the design and verification of heterogeneous systems," *Formal Methods Syst. Design*, vol. 31, no. 1, pp. 1–33, Aug. 2007.

[42] P. Nuzzo *et al.*, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, 2014, DOI: 10.1109/ACCESS.2013.2295764.

[43] S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis, "Optimized distributed implementation of multiparty interactions with restriction," *Sci. Comput. Programm.*, vol. 98, no. 2, pp. 293–316, 2015.

[44] P. C. Attie, E. Baranov, S. Bliudze, M. Jaber, and J. Sifakis, "A general framework for architecture composability," in *Software Engineering and Formal Methods*, vol. 8702. Berlin, Germany: Springer-Verlag, 2014, pp. 128–143, ser. Lecture Notes in Computer Science.

[45] P. Bourgos *et al.*, "Rigorous system-level modeling and analysis of mixed HW/SW systems," in *Proc. 9th IEEE/ACM Int. Conf. Formal Methods Models Codesign*, 2011, pp. 11–20.

[46] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *Proc. Design Autom. Test Eur. Conf. Exhibit.*, 2014, DOI: 10.7873/DATE.2014.167.

ABOUT THE AUTHOR

Joseph Sifakis is a Professor at the École polytechnique fédérale de Lausanne (EPFL), Lausanne, Switzerland. He is the founder of the Verimag laboratory, Grenoble, France, which he directed for 13 years. His current research interests cover fundamental and applied aspects of embedded systems design. The main focus of his work is on the formalization of system design as a process leading from given requirements to trustworthy, optimized, and correct-by-construction implementations. He has actively worked to reinvigorate European research in embedded systems as the scientific coordinator of the «ARTIST» European Networks of Excellence, for ten years. He has been involved in many major industrial projects led by leading companies such as Airbus, EADS, France Telecom, Astrium, and STMicroelectronics.

Dr. Sifakis has received the Turing Award for his contribution to the theory and application of model checking, the most widely used system verification technique today, in 2007. He is a member of the French Academy of Sciences, the French National Academy of Engineering, Academia Europea, and the American Academy of Arts and Sciences.