Foundations and Trends® in
Electronic Design Automation
Vol. 6, No. 4 (2012) 293–362
© 2013 J. Sifakis
DOI: 10.1561/1000000034

Rigorous System Design

By Joseph Sifakis

Contents

Foundations and Trends® in
Electronic Design Automation
Vol. 6, No. 4 (2012) 293–362
© 2013 J. Sifakis
DOI: 10.1561/1000000034

Rigorous System Design

Joseph Sifakis

RiSD Laboratory, EPFL, Lausanne, Switzerland, Joseph.Sifakis@epfl.ch

Abstract

The monograph advocates rigorous system design as a coherent and accountable model-based process leading from requirements to correct implementations. It presents the current state of the art in system design, discusses its limitations, and identifies possible avenues for overcoming them.

A rigorous system design flow is defined as a formal accountable and iterative process composed of steps, and based on four principles: (1) separation of concerns; (2) component-based construction; (3) semantic coherency; and (4) correctness-by-construction. The combined application of these principles allows the definition of a methodology clearly identifying where human intervention and ingenuity are needed to resolve design choices, as well as activities that can be supported by tools to automate tedious and error-prone tasks. An implementable system model is progressively derived by source-to-source automated transformations in a single host component-based language rooted in well-defined semantics. Using a single modeling language throughout the design flow enforces semantic coherency. Correct-by-construction techniques allow well-known limitations of a posteriori verification to be overcome and ensure accountability. It is possible to explain, at each design step, which among the requirements are satisfied and which may not be satisfied.

The presented view for rigorous system design has been amply implemented in the BIP (Behavior, Interaction, Priority) component framework and substantiated by numerous experimental results showing both its relevance and feasibility.

The monograph concludes with a discussion advocating a system-centric vision for computing, identifying possible links with other disciplines, and emphasizing centrality of system design.

1

Introduction

1.1 About Design

Design is the process that leads to an artifact meeting given requirements. These comprise functional requirements describing the functionality provided by the system and extra-functional requirements dealing with the way in which resources are used for implementation and throughout the artifact's lifecycle.

Design is a universal concept, a par excellence intellectual activity linking the immaterial world of concepts to the physical world. It is an essential area of human experience, expertise, and knowledge which deals with our ability to mold our environment so as to satisfy material and spiritual needs. The built world is the result of the accumulation of artifacts designed by humans.

Design has at least two different connotations in different fields and contexts. It may be simply a plan or a pattern for assembling objects in order to build a given artifact. It also may refer to the creative process for devising plans or patterns. In this monograph we adopt the latter denotation with a focus on the formalization and analysis of the process.

Design can be decomposed into two phases. The first is *proceduralization*, leading from requirements to a procedure (executable description) prescribing how the anticipated functionality can be realized by executing sequences of elementary functions. The second is *materialization* leading from a procedure to an artifact meeting the requirements (Figure 1.1). A main concern is how to meet extra-functional requirements by using available resources cost-effectively.

Design is an essential component of any engineering activity. It covers multiple disciplines including electrical, mechanical, thermal, civil, architectural, and computing systems engineering. Design processes should meet two often antagonistic demands: (1) *productivity* to ensure cost-effectiveness; (2) *correctness* which is essential for acceptance of the designed artifacts, especially when they involve public safety and security.

Design is a "problem-solving process". As a rule, requirements are declarative. They are usually expressed in natural languages. For some application areas, they can be formalized by using logics. When requirements are expressed by logical specifications, proceduralization can be considered as a synthesis problem: procedures are executable models meeting the specifications. Model synthesis from logical requirements often runs into serious technical limitations such as non-computability or intrinsically high complexity. For all these reasons, in many areas of engineering, design remains to a large extent an empirical activity relying on the experience and expertise of engineering teams. New complex products are seldom designed from scratch. Their designs follow principles and reuse solutions that have proven their worth. Even if some segments of the design process are fully automated by using tools (e.g., CAD tools), there exist gaps that can be bridged only by creative thinking and insightful analysis.

Design formalization raises a multitude of deep theoretical problems related to the conceptualization of needs in a given area and their effective transformation into correct artifacts. 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 design is by nature multidisciplinary. Its formalization

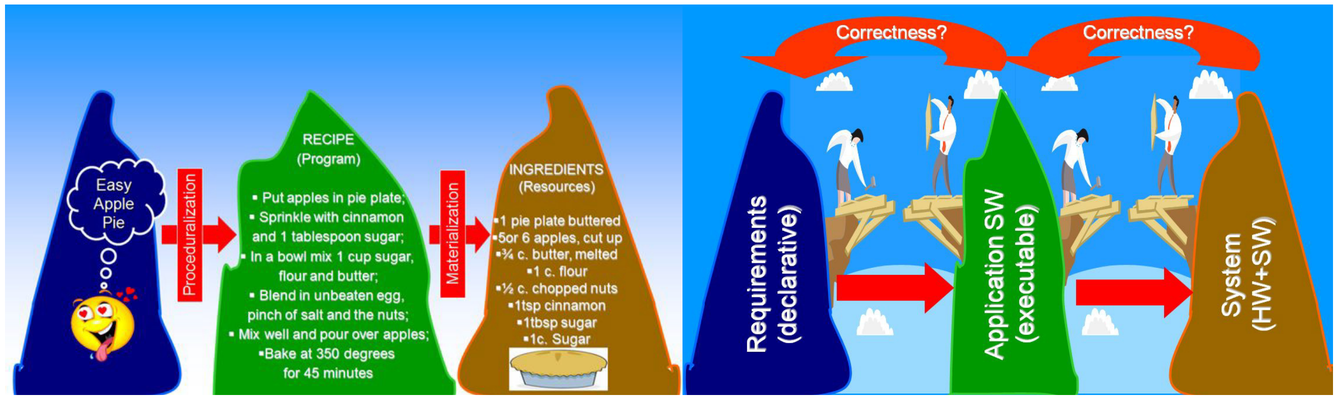

Easy Apple Pie
Proceduralization
RECIPE (Program)
▪ Put apples in pie plate;
▪ Sprinkle with cinnamon and 1 tablespoon sugar;
▪ In a bowl mix 1 cup sugar, flour and butter;
▪ Blend in unbeaten egg, pinch of salt and the nuts;
▪ Mix well and pour over apples;
▪Bake at 350 degrees for 45 minutes
Materialization
INGREDIENTS (Resources)
▪1 pie plate buttered
▪5or 6 apples, cut up
▪¾ c. butter, melted
▪1 c. flour
▪½ c. chopped nuts
▪1tsp cinnamon
▪1tbsp sugar
▪1c. Sugar
Correctness?
Correctness?
Requirements (declarative)
Application SW (executable)
System (HW+SW)

Fig. 1.1 Design is a universal concept applicable from cooking to computing systems.

requires consistent integration of heterogeneous system models supporting different levels of abstraction including logics, algorithms and programs as well as physical system models.

1.2 System Design

The monograph deals with the formalization of the design of mixed hardware/software systems. As a rule, these are interactive systems continuously interacting with an external environment. Their behavior is driven by stimuli from the environment, which, in turn, is affected by their outputs. They drastically differ from function systems which compute an action on an input, producing an output some time later, and stopping. Interaction systems can receive new inputs and produce new outputs while they are already in operation. They are expected to operate continuously.

Interactive systems are inherently complex and hard to design due to unpredictable and subtle interactions with the environment, emergent behaviors, and occasional catastrophic cascading failures, rather than to complex data and algorithms. Compared to function software, their complexity is aggravated by additional factors such as concurrent execution, uncertainty resulting from interaction with unpredictable environments, heterogeneity of interaction between hardware and software, and non-robustness (small variations in a certain part of the system can have large effects on overall system behavior). Henceforth, the term "system" stands for interactive system.

In system design, proceduralization leads to an application software meeting the functional requirements. Materialization consists in building an implementation from application software and models of its execution platforms. As program synthesis is intractable, writing trustworthy application software requires a good deal of creativity and skills. Materialization also requires a deep understanding of how the application software interacts with the underlying hardware and, in particular, how dynamic properties of its execution are determined by the available physical resources.

The monograph advocates rigorous system design as a coherent and accountable process aimed at building systems of guaranteed quality cost-effectively. We need to move away from empirical approaches to a well-founded discipline. System design should be studied as a formal systematic process supported by a methodology. The latter should be based on divide-and-conquer strategies consisting of 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 designers' creativity are essential for achieving design goals.

The design methodology should take into consideration theoretical obstacles as well as the limitations of the present state of the art. It should propose strategies for overcoming as many of the obstacles as possible. The identified theoretical obstacles are the following:

Requirements formalization: Despite progress in formalizing requirements over the past decades (e.g., by using temporal logics), we still lack theoretical tools for the disciplined specification of extra-functional requirements.

For instance, security and privacy requirements should take into account human behavior which is mostly unpredictable and hardly amenable to formalization. Exhaustive and precise specification of system security threats depends on our ability to figure out all possible attack strategies of intruders. Similarly, for privacy violation we need theory for predicting how global personal data can be inferred by combining and interpreting partial data.

Another difficulty is linking user-defined requirements to concrete properties satisfied by the system. This is essential for checking system correctness. The simple requirement that "when an elevator cabin is moving all doors should be closed" may be implied by a mutual exclusion property at system level. To prove formally such an implication, requirements should be analyzed to relate system states to stimuli provided by user interfaces.

Intractability of synthesis/verification: Designers need automated techniques either to synthesize programs from abstract specifications or to verify derived models against requirements. Both problems do not admit exact algorithmic solutions for infinite state systems.

Hardware–Software interaction: We currently have no theory 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 is immaterial. Software models ignore physical time and resources. Hardware is subject to laws of physics. Its behavior is bound to timing constraints, its resources are limited by their physical characteristics. Program execution dynamics inherit hardware-dynamic properties. These properties cannot be precisely characterized or estimated owing to inherent uncertainty and the resulting unpredictability.

Despite these obstacles and limitations, it is important to study design as a systematic process. As absolute correctness is not achievable, we advocate *accountability*, that is, the possibility to assert which among the requirements are satisfied and which may not be satisfied. Accountability can be enhanced by using property-preservation results: if some essential property holds at some design step then it should hold in all subsequent steps. We present rigorous design as a process rooted in four principles.

Separation of concerns: The separation between proceduralization and materialization is crucial for taming complexity. It allows separation of *what* functionality is provided by the system by focusing only on functional requirements, from *how* this functionality is implemented by using resources. Rigorous system design is a formally defined process decomposed into steps. At each step the designer develops a model of the system to be designed at some abstraction level. Within each step, abstraction is progressively reduced by replacing conceptual constructs and primitives by more concrete ones. The final model is a blueprint for building the physical implementation.

Component-based construction: Components are essential for enhanced productivity and correctness through reuse and architectures. In contrast to many other engineering disciplines, computing systems engineering lacks a component taxonomy and theory for component composition. Electrical and mechanical engineering are based on the use of a few component types. Electrical engineers build circuits from elements of predictable behavior such as resistances, capacitances, and inductances. System designers deal with a large variety of heterogeneous components with different characteristics and unrelated coordination principles: synchronous or asynchronous, object-based or actor-based, and event-based or data-based. This seriously limits our ability to ensure component interoperability in complex systems.

Semantic coherency: The lack of a framework for disciplined component-based construction is reflected in the existence of a large variety of languages used by designers. Application software may be written in Domain-Specific Languages (DSL) or general purpose programming languages. Specific languages may be used for modeling, simulation, or performance analysis. These languages often lack well-founded semantics and this is a main obstacle to establishing semantic coherency of the overall design process. Frequently, validation and performance analyses are carried out on models that cannot be rigorously related to system development formalisms. This introduces gaps in the design process which seriously lessen productivity and limit our ability for ensuring correctness. To overcome these limitations, designers should use languages rooted in well-founded semantics defined in a common host language. This language should be expressive enough to establish source-to-source translations between the hosted languages, in particular for enhanced traceability of analysis results at different abstraction levels.

Correctness-by-construction: Correctness-by-checking suffers from well-known limitations. An alternative approach is achieving correctness-by-construction. System designers extensively use algorithms, architectures, patterns, and other principles for structuring interaction between components so as to ensure given properties. These can be described and proven correct in well-founded languages and made available to system designers. A key issue is how to combine existing solutions to partial problems and their properties in order to solve design problems. For this we need theory and rules for building complex designs meeting a given requirement by composing properties of simpler designs.

The monograph proposes a view for rigorous system design and identifies the main obstacles and associated scientific challenges. This view summarizes key ideas and principles of a research program pursued for more than 10 years at Verimag. It has been amply implemented in the BIP (Behavior, Interaction, Priority) component framework [30] and substantiated by numerous experimental results showing both its relevance and feasibility.

BIP consists of a language for component-based construction and an associated suite of system design tools. The language allows the modeling of composite, hierarchically structured systems from atomic components characterized by their behavior and their interface. Components are coordinated by layered application of interactions and of priorities. Interactions express synchronization constraints between actions of the composed components, while priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements, e.g., to express scheduling policies. Interactions are described in BIP as the combination of two types of protocols: rendezvous, to express strong symmetric synchronization and broadcast, to express triggered asymmetric synchronization. The combination of interactions and priorities confers BIP expressiveness not matched by any other existing formalism. It defines a clean and abstract concept of architecture separate from behavior. Architecture in BIP is a first-class concept with well-defined semantics that can be analyzed and transformed. BIP relies on rigorous operational semantics that has been implemented by specific run-time systems for centralized, distributed, and real-time execution.

The monograph is structured as follows.

Section 2 presents significant differences between programs and systems. Section 3 discusses the concept of correctness characterized by two types of hardly reconcilable requirements: trustworthiness and optimization. Trustworthiness requirements capture qualitative correctness while optimization requirements are constraints on resources. Their interplay determines levels of criticality in system design. Section 4 presents existing approaches for system design and their limitations.

We discuss how existing rigorous design paradigms can be transposed to system design. Section 5 discusses the four principles for rigorous system design and their application in the BIP framework. Section 6 presents a system-centric vision for computing, discusses possible links with other disciplines and emphasizes on centrality of system design.

2

From Programs to Systems — Significant Differences

Programs in high-level languages abstract from resources and have a platform-independent behavior: the application software of a system describes pure functionality. In programming languages, time and resources may appear only as external parameters that can be linked to corresponding physical quantities of the execution environment. A real-time program can be considered as an interactive machine where waiting times can be controlled by using mechanisms such as timeouts or watchdogs. At semantic level, reaching a deadline is an external event that is not treated differently from any other external event, e.g., hitting a wall.

The shift of focus from programs to systems should be accompanied by research for extending the Theory of Computing, which focuses on the computation of functions as a terminating process, taking as inputs values of their arguments and producing corresponding results. By its nature, it is of little help for studying systems. As a rule, system behavior is non-terminating and non-deterministic. It can be characterized as an input/output relation between timed histories of values. The interested reader may refer to work on interactive extensions of

Turing machines, which are non-terminating and interacting with an environment [15, 35].

Systems are described by models that specify how the functionality of their software is implemented by using resources such as time, memory, and energy. In system models resources are represented by state variables, in addition to the variables of the application software. Special care should be taken when dealing with resource variables. For each action, the amount of consumed and liberated resources should be specified and resource variables should be modified accordingly.

Extension of current models should not be limited to basic models such as Turing machines and automata. Modeling languages should also be enriched to take into account physical resources and the interplay between systems and their physical environment. We need theory and methods for building faithful system models by extending models of their application software with variables representing resources and their dynamics [19]. This need is now well-understood and has motivated the development of research on Cyber-physical systems addressing various issues including modeling [13] and design [6].

3
Achieving Correctness

3.1 Correctness versus Design Productivity

System designers strive to reconcile two often conflicting demands: design productivity and design correctness.

Productivity characterizes the efficiency of the design process. It can be enhanced through: (1) reuse of components; (2) automation of the design flow by using appropriate methods, and tools; (3) skills and expertise of system designers. These three factors should synergize harmoniously. System designers should have the appropriate background for handling components and tools. Using overly sophisticated tools becomes counter-productive if their users are not adequately trained.

Correctness means compliance to requirements. Formally checking correctness consists in comparing a system model against requirements. It is a relative judgment: “are we building the system right with respect to the requirements?” It would be an answer to the question “are we building the right system” if firstly, requirements could be correctly formalized; and secondly, if system models could faithfully represent the system behavior interacting with its environment.

As flawless system design is not attainable, owing to both theoretical limitations and cost-effectiveness considerations, system designers target *levels of criticality*. These correspond to trade-offs between correctness and productivity. Furthermore, they determine which types of requirements are relevant and to what extent these requirements should be met, e.g., probability of failure or disparity between nominal and observed values of significant parameters.

The proposed concept of system correctness conjoins two types of properties: (1) trustworthiness ensuring that nothing bad would happen; (2) optimization for performance, cost-effectiveness, and trade-offs between them. It differs from pure function software correctness in many respects as it encompasses mostly extra-functional properties while software correctness deals primarily with functional properties; optimization is a concern when we deal with specific implementations.

Trustworthiness and optimization requirements are difficultly reconcilable. As a rule, improving trustworthiness entails non-optimized 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.

3.2 Trustworthiness Requirements

Roughly speaking, non-trustworthiness means that the system can reach some "illegal" state. That is to say, trustworthiness requirements include, in addition to functional requirements, all the extra-functional requirements which qualitatively characterize system correctness.

The study of trustworthy systems has given rise to an abundant amount of literature including research papers and reports as well as a plethora of research projects [33]. Existing approaches either focus on properties that can be formalized and checked effectively (e.g., by using formal methods [9]) or by addressing a very broad spectrum of mostly unrelated topics (see e.g., [29]).

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.

Trustworthiness must be addressed throughout the computing environment. Among the above hazards, only software design errors are limited to application software. The others require the analysis of a system model in interaction with its physical and human environment.

Note that some trustworthiness definitions take into consideration the fact that our confidence in systems is often based on both the artifact itself and on the humans who deliver it. As systems are becoming increasingly sophisticated in their processing and dynamic "learning", our trust in them becomes similar to our trust in humans. Therefore, the corresponding approaches focus on computational trust models in different domains like sociology and psychology. They consider that a thorough understanding of those social, psychological, and engineering aspects of trust is necessary to develop an appropriate trust model [27].

We believe that using such a general concept for the current state of the art is of little technical interest. Thus we prefer to abstract from subjective factors, and study the concept as it is determined by the relevant intrinsic system properties.

3.3 Optimization Requirements

Optimization requirements deal with optimization of functions subject to constraints involving 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) trade-offs between performance and cost-effectiveness.

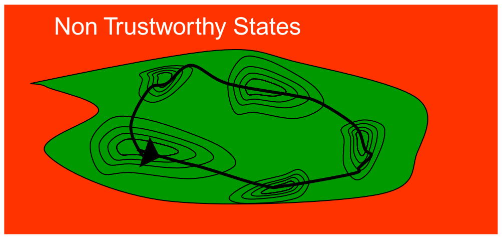


Fig. 3.1 Optimization requirements characterize execution sequences on trustworthy states.

Optimization requirements characterize sets of execution sequences, while trustworthiness requirements determine the set of legal states — a state may be trustworthy or not (Figure 3.1). For systems, they are mostly extra-functional requirements, in contrast to platform-independent optimization requirements applied to software.

A key issue is ensuring trustworthiness without disregarding optimization. This is much easier for monolithic or single processor implementations (e.g., safety critical designs). It is much harder to get optimized trustworthy designs for many-core or distributed implementations.

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. Currently, these techniques are mostly experimental and consist in evaluating, on a system model, the impact of design parameters on optimization criteria. Design parameters include the number and type of processing cores, sizes and organization of memories, interconnect, scheduling and arbitration policies, etc. These determine resource parameters of the system model on which performance, efficiency, and trade-offs between them can be evaluated.

To discover optimized solutions, design should be *parsimonious*, that is, design choices should be implied only by requirements. Designers often preclude possible solutions by eliminating alternatives based on the idea that the flexibility they afford would be difficult or impossible to exploit later [20]. Very early in the design process, they favor specific programming models or implementation principles which drastically reduce the design space. For instance, developing an encoder in plain C leads to solutions hard to parallelize and implement on a multiprocessor platform. Alternatively, programming in a data-flow language can help discover parallelism and enables the use of adequate scheduling policies by applying existing theory [24].

A prerequisite for parsimonious design is using appropriate programming languages to unveil inherent data or task parallelism and non-determinism. Optimized implementations can be obtained by *adequate design choices*: reducing parallelism (through mapping on the same processor), reducing non-determinism (through scheduling), or fixing parameters such as quality, precision, frequency, and voltage.

3.4 Levels of Criticality

The criticality of a system is the degree to which a violation of some trustworthiness requirement could have a dramatic impact on human life, the environment, or significant assets. For safety-critical systems, such as flight controllers or nuclear plant controllers, violation can come from the system itself or from its interaction with its physical environment. Security-critical systems are sensitive to attacks or any kind of malevolent interactions with humans (e.g., smart cards).

Critical systems development is costly and requires special techniques guaranteeing absence of critical failures. Often it must be certified according to safety-critical systems standards such as DO178B for airborne systems and ISO26262 for automotive systems. The Common Criteria for Information Technology Security Evaluation is the technical basis for certification of secure IT products.

Currently, critical systems are confined to small size systems owing to both limitations of the present state of the art and high development costs. These are roadblocks to their extensive use in areas such as automotive or health applications. For instance, projects for “active” safety in cars based on “drive-by-wire” or “brake-by-wire” may not become a reality because of these limitations. Similarly, there exists a huge potential for healthcare applications that are still at an experimental stage.

In addition to safety and security, other criteria can be chosen to characterize system criticality. Mission-critical systems used in space, telecommunications, and data centers must ensure, even at limited capacity, completion of tasks. Another criterion is business criticality evaluated as the impact of a downtime on the revenues generated from system exploitation (e.g., data center systems). Finally, best-effort systems are systems whose failures have a limited impact leading to error states from which quick recovery is possible. Their development focuses primarily on optimized use of resources, provided their availability remains above a certain threshold. Best-effort systems comprise a large variety of applications and services (e.g., web-based applications).

4

Existing Approaches and the State of the Art

We provide a succinct overview of main approaches for system development, successful paradigms of rigorous design, and discuss limitations of the state of the art and associated technical challenges.

4.1 System Development Methodologies

Existing development methodologies are of limited interest for systems. They prescribe only general principles and fail to provide rigorous support and guidance.

The *V-model methodology* is an extension of the waterfall model, which considers development as a sequence of phases from requirements specification to implementation and validation [21]. It decomposes system development into two flows.

1. One top-down starting from requirements and involving a hierarchical decomposition of the system into components and associated requirements. Components are designed separately so as to meet their requirements.
2. The other is bottom-up and consists in progressively assembling, integrating, and testing the designed components.

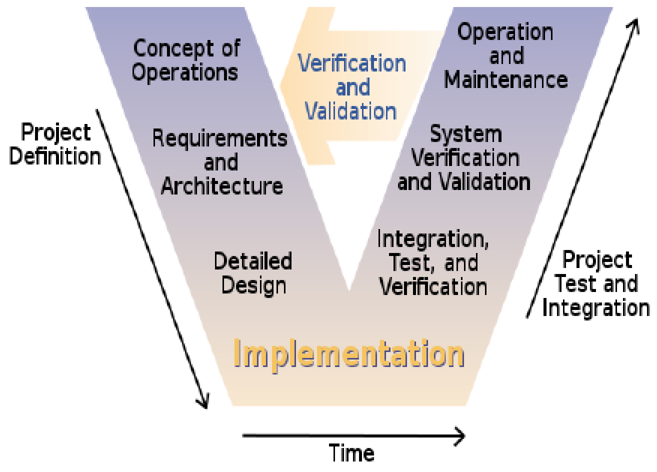

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Integration, Test, and Verification
Detailed Design
Project Test and Integration
Implementation
Time

Fig. 4.1 The V-model (*source*: Wikipedia).

This methodology, depicted in Figure 4.1, can be criticized for four main reasons:

(1) It assumes that all the system requirements are initially known, and can be clearly formulated and understood.
(2) It assumes that system development is top-down from a set of requirements. Nonetheless, systems are never designed from scratch; they are built by incrementally modifying existing systems and component reuse.
(3) It considers that global system requirements can be broken down into requirements satisfied by system components. Furthermore, it implicitly assumes a compositionality principle: atomic components are proven correct with respect to their individual requirements and then correctness of the whole system can be inferred from their correctness. This principle is not applicable to emergent properties (e.g., mutual exclusion). Furthermore, the number of unanticipated interactions scales exponentially with the number of the components, in the course of integration. So, inevitably, a re-design cycle begins. In fact, the two sides of the "V" are ever more interconnected with increasingly frequent re-designs and incremental modification.

(4) It relies mainly on correctness-by-checking (verification or testing).

Agile development has been proposed as an alternative to the V-design methodology [3]. It puts emphasis on incremental development of solutions and collaborative team work. It considers that coding and designing go hand in hand: designs should be modified to reflect adjustments made to the requirements. So, design ideas are shared and improved on during a project. The main merit of this methodology is its criticism of the V-model rather than a disciplined and well-structured way for tackling system development.

4.2 Rigorous Design Techniques

There exist two classes of successful rigorous design techniques. One is applied to hard real-time systems ensuring trustworthy control of aircraft, cars, power plants, and medical devices. The other comes from hardware engineering. VLSI design and associated EDA tools have enabled the IC industry to sustain almost four orders of magnitude in product complexity growth since the 80386, while maintaining a consistent product development timeline.

We present below the main characteristics of these techniques which also explain their main reasons of success.

Hard real-time systems design techniques: The design of hard real-time systems relies on the use of domain-specific languages for programming application software and of associated rigorous implementation methods.

Synchronous programming languages are used for the development of synchronous reactive systems [16]. Synchronous programs can be considered as a network of strongly synchronized components. Their execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation. As a rule, synchronous implementations are monolithic on bare metal. An implementation is correct if the worst-case execution times (WCET) for steps are less than the requested response time for the system.

Asynchronous design techniques are represented mainly by flows using the ADA standard [36]. Implementations are event-driven based on dedicated multi-tasking run-time environments. Fixed priority scheduling policies are used for sharing resources between components. Scheduling theory allows predictable response times for components with known period and time budgets [10].

Finally, time-triggered techniques are based on the use of specific programming models and associated implementation techniques on dedicated platforms [22]. They can guarantee by construction correctness with respect to timing requirements.

An interesting question is whether these paradigms can be extended to encompass mixed-criticality systems. Clearly, existing techniques adopt specific scheduling principles that can guarantee by construction, satisfaction of essential properties. For a successful generalization of these paradigms, we need theory for composing components each equipped with specific scheduling requirements, and deriving global scheduling policies for the resulting system, as well as the minimal amount of resources for meeting these requirements. This vision goes far beyond the current state of the art.

VLSI design techniques: These techniques support rigorous design flows leading from structural component-based descriptions expressed in an HDL to their physical implementation based on a powerful well-understood abstraction hierarchy. They use a limited and well-defined number of synchronous components. Homogeneity of the model of computation greatly simplifies the analysis of component interaction. Correctness is achieved mainly by construction through extensive use of architectures and of powerful synthesis tools. Additionally, VLSI design techniques benefit from mature algorithmic verification technology based on efficient representation and computation of Boolean functions (e.g., BDDs).

Clearly, the above techniques provide only instructive templates. Integrated circuits consist of a limited number of fairly homogeneous components. Hard real-time design techniques are costly and not adequate for general purpose systems. Their main reasons for success are: (1) coherent and accountable design flows, supported by tools and often enforced by standards; and (2) correct-by-construction design enabled by extensive use of architectures and formal design rules. The application of these principles to system design is hampered by several obstacles such as the lack of a common component model, the heterogeneity of models of computation, the variety of architectural styles, and the intractability of synthesis for infinite state systems.

4.3 The Limits of Correctness-by-Checking for Systems

Correctness-by-checking is ensured by validation of requirements on a real system prototype or on a system model. Validation of system prototypes can be done only by testing, that is, by exercising the system's behavior according to test cases and checking whether the observed response agrees with requirements. Validation of models can be either ad hoc (e.g., by simulation) or by formal verification. In the latter case, requirements and system models are expressed in formal languages related through a satisfaction relation.

Formal verification and algorithmic verification, in particular, constitutes one of the main breakthroughs for quality assurance in both hardware and software. It has drastically contributed to gaining mathematical confidence that both common and critical computing applications meet their specifications [11]. Despite spectacular progress of the state of the art over the past decades, verification techniques suffer from inherent well-known limitations [28].

A general discussion about these limitations is out of the scope of this monograph. We explain below how they are aggravated for system verification, compared to software and hardware verification.

There exist three main sources of limitations regarding the ability: (1) to apprehend and formally express user's needs by requirements; (2) to faithfully model the system to be verified; and (3) to overcome inherent theoretical limitations of verification techniques.

Expressing requirements: Requirements should be written in a formal language easy to understand and use by engineers. Expression of trustworthiness properties requires first and foremost a clear understanding and characterization of "bad situations" to be avoided. The identification of these situations is quite straightforward for non-interactive systems. For interactive systems, this may turn out to be non-trivial as it is necessary to figure out for a given property, all the relevant patterns of interaction between a system and its environment. This is typically the case for security properties whose expression may require a deep analysis (e.g., anticipating all possible malevolent actions coming from a system's environment).

Expression of optimization properties requires an even more subtle reasoning over system execution sequences. Currently, we lack adequate languages for formally expressing quality of service requirements involving bit rate, jitter, and latency.

From a more pragmatic point of view, the use of rigorous requirement specification languages for real-life systems seems to be problematic. In fact, requirements specifications must meet two fundamental properties:

- *Soundness*, that is, there exists at least one system meeting the specifications (no contradiction). Checking this property even for decidable requirements specification languages may be questionable owing to the intrinsic complexity of decision algorithms.
- *Completeness,* that is, requirements specify tightly enough the system's behavior. There is no technical criterion characterizing completeness for declarative languages — writing requirements in a declarative language may be an endless game!

Modeling: System models should be faithful, that is, to say that whatever property is satisfied for the model also holds for the real system. Furthermore, they should be generated automatically from system descriptions.

Currently, we master automatic generation of faithful models for hardware systems. For software systems, we can generate models for checking functional requirements, provided they are written in languages with well-defined semantics. The operational semantics of the language can be used to formally define a transition system on which verification techniques can be applied. Nonetheless, for interactive systems we lack faithful detailed modeling techniques. Generating faithful models even for very simple systems, such as the node of a wireless sensor network, requires understanding intricate interaction between application software and the underlying execution platform, including hardware-dependent software and hardware.

Verification techniques: Formal verification allows exhaustive checking. Currently, automated verification techniques such as model checking, abstract interpretation, and static analysis are all monolithic. They are applied to global transition systems whose size increases exponentially with the number of the components of the system to be verified. A direct consequence of this state explosion phenomenon is that current verification techniques are limited to small or medium size systems and to specific properties.

Attempts to apply compositional verification to component-based systems, such as assume/guarantee techniques, failed to make any significant breakthrough [12]. The main obstacle to overcome is breaking up a global requirement into a set of local requirements such that (1) each requirement is met by a constituent component; and (2) their conjunction implies the global requirement.

As a conclusion, correctness-by-checking contributes to trustworthiness but it is limited to requirements that can be formalized and checked efficiently (mainly verification of functional properties for application software). For the same reasons, its application to optimization requirements is limited to the validation of scheduling and resource management policies on abstract system models. In any case, verification is applied to medium size systems when it is possible to make automated proofs or when the cost of faults is high.

For optimization requirements, a more natural approach for their satisfaction is by enforcing rather than by checking. That is, instead of checking a requirement depending on some parameters, sets of parameter values for which it is satisfied should be determined. This can be achieved either by synthesis techniques subject to even more severe limitations than checking techniques or by using adaptive control techniques [2]. The latter allow on-line adaptation of the values by monitoring system execution.

4.4 The Integration Wall — Mixed-Criticality Systems

4.4.1 Mixed-Criticality Systems

Increasing systems integration inevitably leads to systems-of-systems of mixed criticality that are geographically distributed, are heterogeneous and use various communication media. A key issue for integration is mastering interaction of critical and non-critical features and error containment. Preventing failures of non-critical components from affecting the behavior of critical components raises difficult problems. However, the theory with which to tackle them is lacking.

There exist several incarnations of this systems-of-systems vision. The most general one is the Internet of Things which is intended to develop global services by interconnecting everyday objects. One instance of this vision is Smart Grids for efficient and reliable energy management. Another instance is Intelligent Transport Systems to improve safety and reduce vehicle wear, transportation times, and fuel consumption.

Integration of mixed-criticality systems of guaranteed trustworthiness is currently an unattainable goal. The main reason is that critical systems and best-effort systems are developed following two completely different and diverging design paradigms. We outline below the technical reasons leading to such a separation and identify avenues for achieving enhanced integration.

4.4.2 The Issue of Predictability

Systems must provide a service meeting given requirements in interaction with uncertain environments. Roughly speaking, uncertainty can be characterized as the difference between average and extreme system behavior. There are two main sources of uncertainty.

1. *The system's external environment*: Non-determinism comes either from inputs with time-varying characteristics (e.g., varying throughput) or from the fact that the external environment is inherently complex and its behavior can be understood and described only at some level of abstraction.

How to figure out all possible security threats devised by an experienced hacker?

2. *The hardware execution platform*: Hardware has inherently non-deterministic behavior owing to manufacturing errors or aging. It also exhibits time non-determinism since execution times of even simple instructions cannot be precisely estimated due to the use of memory hierarchies and speculative execution. Depending on the size and the location of the data, execution times can vary between a best-case execution time (BCET) and a worst-case execution time (WCET) that may be 10 times larger.

Uncertainty directly affects predictability, that is to say the degree to which qualitative or quantitative system properties can be asserted. Lack of predictability is further aggravated as exact analysis techniques are impossible owing to non-computability of all essential system properties. For instance, timing analysis techniques allow upper approximations of WCET, which may be many orders of magnitude larger.

Uncertainty and the resulting non-predictability have a deep impact on the way we design systems. They limit our ability to design complex critical systems.

4.4.3 The Gap between Critical and Best-Effort System Design

Currently, there exist two diverging system design paradigms.

1. *Critical systems design* focuses on ensuring satisfaction of trustworthiness properties. It is based on worst-case analysis of all the potentially dangerous situations. Currently, design principles lead to over-provisioned systems. They consist in statically reserving all the resources needed for safe or secure operation. The amount of physical resources may be some orders of magnitude higher than necessary. This incurs high production costs as well as increased energy consumption.
   For example, response times of a critical real-time system must be guaranteed to be less than a given deadline. These

are computed from safe approximations of the WCET of its tasks which may be many orders of magnitude larger than the real WCET. The resulting hardware platforms for such systems are excessively over-dimensioned.

Another principle from critical systems engineering consists in using massive redundancy to enhance reliability. Redundancy techniques such as Triple Modular Redundancy (TMR), entail high development and operational costs (e.g., energy consumption). They are appropriate only for hardware when probabilities of failure in redundant components are independent. With the increasing miniaturization of transition features, this assumption is less and less valid. Application of TMR techniques to systems is costly as it requires different versions of software in redundant components.

2. *Best-effort design* focuses primarily on meeting optimization requirements of complex non-critical systems. It is based on average-case analysis and dynamic resource management. Designers apply QoS management techniques to optimize speed, memory, bandwidth, and power. Physical resources are provisioned for availability in nominal cases. In critical situations, e.g., a spike in service demands, the system service may be degraded or denied.

The separate design between critical and best-effort systems is a means for coping with non-predictability by focusing on essential properties for each class of systems. Nonetheless, this separation may be the source of hurdles, as attested by serious technical problems experienced by car manufacturers over the past decade. A modern car has currently more than 50 ECUs (Electronic Control Units) which are electronic components ensuring services of various levels of criticality. These are coordinated by using federated architectures where ECUs share and exchange information by using networks such as CAN or Time-Triggered Networks. An advantage of these architectures is physical isolation between critical and less critical functions. Even so, poor dependability of interconnect has a profound influence on the overall system quality. Furthermore, there are many cost and weight arguments in favor of reducing the number of ECUs and interconnect by integrating different functions, developed by different suppliers, into a single ECU. This leads to the concept of integrated architecture where a single integrated distributed hardware base is used for the execution of jobs from different subsystems [23].

Recently, a number of efforts have been made to develop integrated architectures, including Integrated Modular Avionics (IMA) in the aerospace domain and AUTOSAR in the automotive domain. Research efforts should focus on jointly addressing trustworthiness and optimization and seeking integrated solutions throughout system design.

5

Four Principles for Rigorous System Design

5.1 Rigorous System Design

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. It relies on divide-and-conquer strategies involving iteration on a set of steps and clearly identifying points where human intervention and ingenuity are needed to resolve design choices as well as segments that can be supported by tools to automate tedious and error-prone tasks.

Rigorous system design is model-based: successive system descriptions are obtained by application of correct-by-construction source-to-source of a single expressive model rooted in well-defined semantics. An additional demand is accountability, that is, the possibility to explain which among the requirements are satisfied and which may not be satisfied.

As explained in the Introduction, we advocate four principles for rigorous system design discussed in detail below.

5.2 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 the design flow. First, application software is developed that is correct with respect to the functional requirements. Then, a correct implementation meeting both functional and extra-functional requirements is derived by progressive refinement of the application software taking into account features of the execution platform.

We discuss the main obstacles to be overcome for bridging these gaps. We also identify relevant research directions for overcoming these obstacles.

5.2.1 From Requirements to Application Software

In system software development the key issue is managing the complexity resulting from interactions with the environment and amongst various subsystems. Using general-purpose programming languages, such as C or Java, may be counter-productive and error-prone. These languages are adequate mainly for sequential transformational programs computing functions. Existing programming languages and technology should be improved in two directions:

Raising abstraction: Programming should get as close as possible to the declarative style so as to simplify reasoning and relegate software generation to tools. There exist many approaches for enhanced abstraction including logical, constraint-based, and functional programming. These leave to interpreters or compilers that task of synthesizing executable descriptions. In our opinion, for system design, the most adequate abstractions are offered by automata-based formalisms. The latter focus on system behavior description as a set of transitions involving actions guarded by conditions, such as behavioral programming [18], BIP, and scenario-based formalisms which capture a system's behavior as a set of scenarios from which behavior can be synthesized [26].

Support for System Programming: General-purpose programming languages do not provide adequate support for concurrency and communication. For systems we need powerful primitives encompassing direct description of different types of synchronization. Problems that have straightforward solutions by using automata 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.

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. For instance, 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 are amenable to schedulability analysis. Other examples of DSLs are nesC, an extension to C designed to embody the structuring concepts and execution model of the TinyOS platform for wireless sensor networks, and BEPL, an orchestration language for business processes and services.

5.2.2 From Application Software to Implementation

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 constraints:

- User-defined constraints which express requirements dealing with reaction times, throughput, and cost, such as deadlines, periodicity, memory capacity, and power or energy limitations.

- Platform-dependent constraints expressing 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 variables of both the application software variables and resource variables. Any execution sequence of a system model corresponds modulo some adequate abstraction, to a sequence of its application software.

Building faithful system models is still an unexplored and poorly understood problem. Queuing network theory allows macroscopic modeling based on architectural abstractions that cannot take into account values of data and data-dependent choice. Detailed, albeit ad hoc system models are usually written in languages such as systemC or TLM. Nonetheless, these languages are not rooted in rigorous semantics and it is impossible to establish faithfulness of models.

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, and memory hierarchy and; (2) symbolic analysis techniques based on static analysis and abstract interpretation. Owing to theoretical limitations, the latter can compute only rough 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. This is a significant difference between model time and physical time. 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, a 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.

An approach for analyzing model feasibility and better understanding of the interplay between user-defined constraints and platform-dependent constraints has been proposed for timed systems in [1]. The approach consists in comparing two system models: (1) an *ideal system model*, representing the system's behavior taking into account user-defined constraints for unlimited resources; and (2) a *physical system model* where both types of constraints are applied. For a given ideal system model, many different physical models can be obtained by changing the quantities of resources needed for the execution of each action. Thus, a physical model *is the ideal model equipped with* a function ϕ assigning to an action the quantity of resources needed for its execution. The function $1/\phi$ characterizes the *performance of the execution platform*. For $\phi = 0$ the two models coincide and performance is infinite. For some function ϕ, a physical model can be considered as a *safe implementation* of the ideal model if all its execution sequences are also execution sequences of the ideal model. An interesting problem is how to determine the worst performance ensuring safe implementations. Unfortunately, the intuitive idea that safety of implementation is preserved for increasing performance turns out to be wrong. That is if $\phi' < \phi$, safety for ϕ does not imply safety for ϕ', in general. This phenomenon limits our capability to analyze system model feasibility. When it relates to time-performance, it is called timing anomaly [31]. A direct consequence of timing anomalies is that safety for WCET does not guarantee safety for smaller execution times. Preservation of safety by time-performance is called *time robustness* in [1] where it is shown that this property holds for deterministic models.

Resource robustness is essential for analyzing system models. It captures a basic principle widely used in all areas of engineering. As a rule, performance changes monotonically with resource parameters. For example, 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.

Failure to determine more general robustness conditions would concur with the thesis that predictability boils down to determinism.

5.3 Component-Based Design

Using components throughout a system design flow is essential for enhanced productivity and correctness. 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. We advocate a common component framework for systems engineering encompassing heterogeneous composition.

A key issue for the integration of languages used by system designers is the definition of a general Common Component Model. There exist a large number of component frameworks, including software component frameworks, systems description languages, and hardware description languages [3, 16, 36, 22, 12]. Nonetheless, despite an abundant literature and a considerable volume of research, there is no agreement on a common concept of component. This is mainly due to heterogeneity of components and associated composition operations. There exist various sources of heterogeneity [30]:

- *Heterogeneity of computation*: components may be synchronous or asynchronous.
- *Heterogeneity of interaction*: various mechanisms are used to coordinate the execution of components including semaphores, rendezvous, broadcast, method call, etc.
- *Heterogeneity of abstraction*: components are used at different abstraction levels from application software to its implementation.
- *Heterogeneity of programming styles*: components may be actors (local control and disciplined communication) or objects (transfer of locus of control).

We lack component frameworks based on a unified composition paradigm for describing and analyzing coordination between components in terms of tangible, well-founded, and organized concepts. Coordination should be expressed by using architectural constraints which are composition operators on components. Their meaning can be defined by using operational semantics rules specifying the behavior of a composite component as a function of the behavior of its constituent components.

5.3.1 Needs and State of the Art

We need theory, models, and tools for the cost-effective building of complex systems by assembling heterogeneous components.

System descriptions used along a design flow should be based on a single semantic model to maintain its overall coherency by guaranteeing that a description at step $n + 1$ meets the essential properties of a description at step n. The semantic model should be expressive enough to directly encompass component heterogeneity. 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 [5]. For instance, if the composition is by strong synchronization (rendezvous), modeling broadcast requires components for choosing the maximal amongst several possible strong synchronizations. We need frameworks providing families of composition operators for a natural and direct description of coordination mechanisms such as protocols, schedulers, and buses.

We discuss below general concepts and requirements for system component frameworks formulated in [32].

5.3.2 Component Frameworks

A *component framework* consists of a set of *atomic components* $\mathrm{B} = \{\mathrm{B}_i\}_{i \in \mathrm{I}}$ and a *glue* $\mathrm{GL} = \{\mathrm{gl}_k\}_{k \in \mathrm{K}}$, set of operators on these components. Atomic components are characterized by their behavior specified as a transition system. The glue GL includes general composition operators (behavior transformers).

The meaning of a glue operator gl can be specified by using a set of operational semantics rules defining the transition relation of the composite component $\mathrm{gl}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ as a partial function of transition relations of the composed components $\mathrm{C}_1, \ldots, \mathrm{C}_n$. If from state s_i, component C_i can perform an action a_i by executing transitions of the form $\mathrm{s}_i - \mathrm{a}_i \rightarrow \mathrm{s}'_i$, then $\mathrm{gl}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ can execute transitions of the form $(\mathrm{s}_1, \ldots, \mathrm{s}_n) - \mathrm{a} \rightarrow (\mathrm{s}''_1, \ldots, \mathrm{s}''_n)$ where a is an *interaction*, a non-empty subset of $\{\mathrm{a}_1, \ldots, \mathrm{a}_n\}$ such that $\mathrm{s}''_i = \mathrm{s}'_i$ if $\mathrm{a}_i \in \mathrm{a}$ and $\mathrm{s}''_i = \mathrm{s}_i$, otherwise. A technical definition for glue operators is provided in [5].

A component framework can be considered as a term algebra equipped with a congruence relation $\approx$ compatible with strong bisimulation on transition systems. A composite component is any (well-formed) expression built from atomic components.

Moreover, glue operators must meet the following requirements:

(1) *Incrementality*: If a composite component is of the form $\mathrm{gl}(\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_n)$ for $n \geq 2$, then there exists glue operators gl_1 and gl_2 such that $\mathrm{gl}(\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_n) \approx \mathrm{gl}_1(\mathrm{C}_1, \mathrm{gl}_2(\mathrm{C}_2, \ldots, \mathrm{C}_n))$. Notice that incrementality is a kind of generalized associativity. It requires that coordination between n components can be expressed by first coordinating $n-1$ components and then by coordinating the resulting component with the remaining argument.

(2) *Flattening*: Conversely, if a composite component is of the form $\mathrm{gl}_1(\mathrm{C}_1, \mathrm{gl}_2(\mathrm{C}_2, \ldots, \mathrm{C}_n))$ then there exists an operator gl such that $\mathrm{gl}_1(\mathrm{C}_1, \mathrm{gl}_2(\mathrm{C}_2, \ldots, \mathrm{C}_n)) \approx \mathrm{gl}(\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_n)$. This property is essential for separating behavior from glue and treating glue as an independent entity that can be studied and analyzed separately.

It should be noted that almost all existing frameworks fail to meet both requirements. Process algebras are based on two composition operators (some form of parallel composition and hiding) which are orthogonal to behavior, but fail to meet the flattening requirement. General component frameworks, such as [14, 25], adopt more expressive notions of composition by allowing the use of behavior for coordination between components and thus do not separate behavior from interaction. Furthermore, most of these frameworks are hardly amenable to formalization through operational semantics.

Expressiveness: Comparison between different formalisms and models is often made by flattening their structures 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 [5]. It allows the comparison of two component frameworks with glues GL and GL$'$, respectively, equipped with the same congruence relation $\approx$.

We say that GL$'$ is more expressive than GL if for any composite component $\mathrm{gl}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ obtained by using $\mathrm{gl} \in \mathrm{GL}$ there exists $\mathrm{gl}' \in \mathrm{GL}'$ such that $\mathrm{gl}(\mathrm{C}_1, \ldots, \mathrm{C}_n) \approx \mathrm{gl}'(\mathrm{C}_1, \ldots, \mathrm{C}_n)$. 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. For instance, is multiparty interaction by rendezvous more expressive than broadcast?

There exists one most expressive component framework defined by the *universal glue* $\mathrm{GL}_{\mathrm{univ}}$ which contains all possible glue operators. An interesting question is whether the same expressiveness can be achieved with a minimal set of operators. Results in [5] bring a positive answer to this question. It is shown that the glue of the BIP framework [30] combining two classes of operators, interactions and priorities, is as expressive as $\mathrm{GL}_{\mathrm{univ}}$. Furthermore, this glue is minimal in the sense that it loses universal expressiveness if either interactions or priorities are removed.

A consequence of these results is that most existing formal frameworks using only interaction such as process algebras are less expressive. It can be shown that they are even less expressive by using the following weaker notion of expressiveness.

GL' is weakly more expressive than GL if for any component $\mathrm{gl}(\mathrm{C}_1,\ldots,\mathrm{C}_n)$ with $\mathrm{gl} \in \mathrm{GL}$ there exist $\mathrm{gl}' \in \mathrm{GL}'$ and a finite set of atomic components $\{\mathrm{C}'_1,\ldots,\mathrm{C}'_k\}$ such that $\mathrm{gl}(\mathrm{C}_1,\ldots,\mathrm{C}_n) \approx \mathrm{gl}'(\mathrm{C}_1,\ldots,\mathrm{C}_n,\mathrm{C}'_1,\ldots,\mathrm{C}'_k)$. That is, to realize the same coordination as gl, additional behavior is needed. It can be shown that glues including only interactions fail to match universal expressiveness even under this definition [5]. Adding new atomic components does not suffice if the behavior of the composed components is not modified.

Getting rid of the Babel syndrome: is it possible to find a rigorous approach for dealing with components and their composition? It is hard to bring precise technical answers to this question. Solutions should be sought by proposing taxonomy of existing component frameworks based on a single reference component model. The latter should be expressively complete and rooted in well-defined semantics. Ideally, the taxonomy should be obtained through specialization of the reference component model by identifying types of atomic components and sets of associated composition operators (glues).

5.4 Semantically Coherent Design

System designers deal with a variety of languages with different features and characteristics including domain-specific languages, functional or imperative programming languages, modeling, and simulation languages. Consistent and effective use of such languages in a design flow is essential for:

- preserving its overall coherency by relating system descriptions and their properties for different abstraction levels and purposes (validation, performance evaluation, code generation); and
- evaluating the impact of choices at different design steps.

System designers use multi-language frameworks consistently integrating programming and modeling languages and their associated supporting tools. These include, in particular, DSLs which are high-level languages encompassing specific programming models (e.g., data-flow, event-driven, time-triggered, synchronous languages) in adequacy with the application domain as well as the skills and culture of system developers. System development in plain procedural programming languages (e.g., C and Java) may be counter-productive and error-prone. Most of these languages lack formal operational semantics, their meaning being defined by user manuals and their supporting tools. Using semantically unrelated languages in a design flow breaks continuity of activities and jeopardizes its overall coherency.

To enforce coherency in design frameworks, their languages, DSLs in particular, are translated into a common general-purpose programming language like C, C++, or Java. The concept of embedding discussed below defines a principle for structure-preserving language translation.

We 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 congruence $\approx$ defined at semantic level 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 is 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 $\chi(t) \in \mathrm{H}$. 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, $\chi(\mathrm{B})$ is an atomic component of H; and (2) for any term $t = \mathrm{gl}(\mathrm{C}_1, \ldots, \mathrm{C}_n) \in \mathrm{L}$, $\chi(t) = \chi(\mathrm{gl})(\chi(\mathrm{C}_1), \ldots, \chi(\mathrm{C}_n)) \in \mathrm{H}$.
- 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 $t \in \mathrm{L}$ into a term $\sigma(t) \in \mathrm{H}$, 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 $t \in \mathrm{L}$ its semantic glue $\sigma_1(t)$ and an execution engine $\sigma_2(t)$ both expressed in H, so that $\sigma(t) = \sigma_1(t)(\chi(t), \sigma_2(t)) \approx t$.

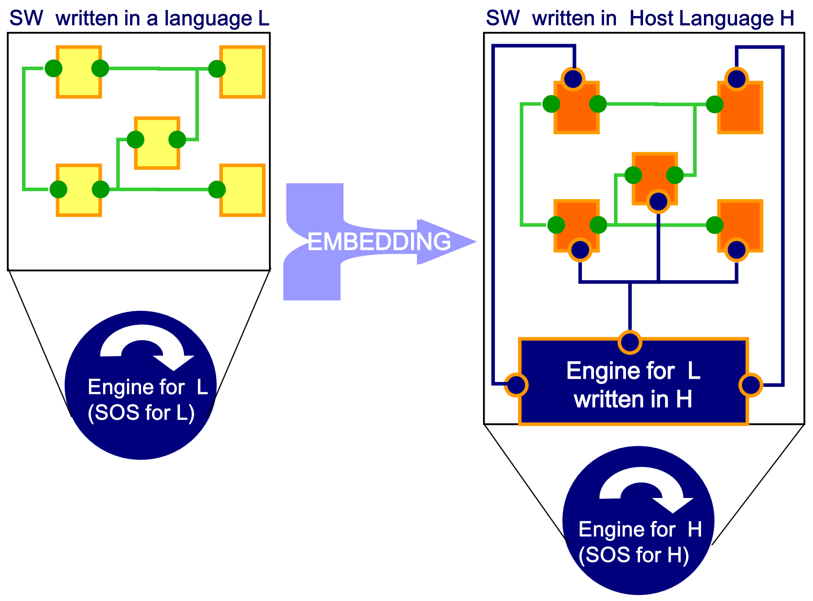


Fig. 5.1 Embedding language L into the host language H.

Embeddings translate separately the coordination mechanism explicitly handled by the programmer from additional coordination mechanisms implied by the operational semantics. Figure 5.1 illustrates the concept of embedding. On the left, the software written in L is a set of components with their glue. The structured operational semantics (SOS) of L defines an execution engine that coordinates the execution of components as specified by the glue. The embedding preserves the structure of the source. Atomic components of L are translated into atomic components of H with additional ports. These are used by the additional component representing the execution engine of L in H.

Embedding real executable languages is a non-trivial task as it requires a formalization of their intuitive semantics. The interested reader can find in [30] papers dealing with the definition and implementation of embeddings into BIP for languages such as nesC, DOL, Lustre, and Simulink.

Figure 5.2 explains the principle of translation of the Lustre language [16] through an example. Lustre is a synchronous data-flow

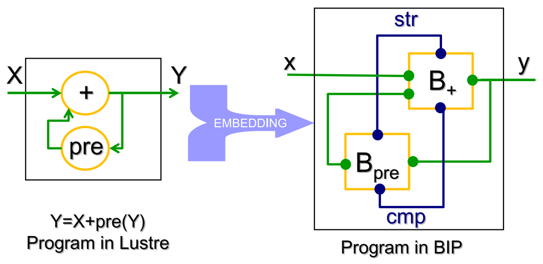


Fig. 5.2 Embedding Lustre into BIP.

language. The meaning of a program is a system of recurrence equations. Programs can be represented as block diagrams consisting of functional nodes that synchronously transform their input data streams into output strings. A node computes a function by exhibiting cyclic behavior: when a cycle starts, it reads its current input values and computes the corresponding function.

The considered Lustre program is an integrator. The variables X and Y represent flows which are sequences of integers. The + operator computes the sum of its inputs. The pre-operator is a unit delay. It is easy to see that the program computes the sum of the integer values of the input flow $X = (x_0, x_1, \ldots, x_i, \ldots)$ and produces the output flow $Y = (x_0, x_0 + x_1, \ldots, x_i + x_{i-1} + \cdots + x_0, \ldots)$, assuming that the initial state of the pre-operator is 0.

The structure of the BIP program obtained through embedding is shown on the left of the figure. There is a one-to-one correspondence between the components of the two programs. The behavior of the BIP components is modeled by event-driven automata B_+ and $\mathrm{B}_{\mathrm{pre}}$ extended with data. The automata are synchronized by interactions (rendezvous between actions). They explicitly represent the control needed for the execution of each node. They synchronously start and complete cycles by executing interactions str and cmp, respectively. This simple example does not require extra coordination components in the translated system. Notice that the translation preserves the structure of the Lustre program. Data-flow connections are replaced by interactions. Connectors str and cmp are used to model synchronous execution of components which is implicit in Lustre.

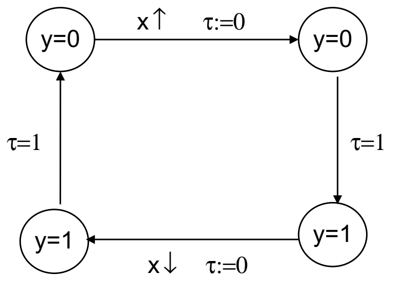


Fig. 5.3 Timed automaton representing a unit delay $y(t) = x(t-1)$.

Embedding languages for modeling physical systems such as Modelica into an executable language raises additional problems. These languages are declarative with continuous dynamics parameterized by a common time parameter. They model systems as networks of intrinsically parallel components with data-flow connectors. When they are translated into executable languages, their inherent parallelism is simulated by using a shared state variable representing time. All events are dated by using a common time base.

To illustrate these difficulties, consider a unit delay equational specification $y(t) = x(t-1)$, where x and y are binary variables and t is time. Its behavior can be represented by the timed automaton in Figure 5.3 with four states, provided that there is at most one change of x in one time unit. The automaton detects for the input x, raising edge ($x\uparrow$) and falling edge ($x\downarrow$) events and reacts within time unit. Reaction times are enforced by using a clock τ. Notice that the number of states and clocks needed to represent a unit delay by a timed automaton increases linearly with the maximum number of changes allowed for x in one time unit.

5.5 Correct-by-Construction Design

5.5.1 Principles

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. Testing may be still necessary, but its role is to validate the correct-by-construction process rather than to find bugs.

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

Essential Properties

We present principles for a correct-by-construction methodology focusing on the preservation of two types of essential properties.

- *Invariants* are state predicates preserved by the transition relation. If an invariant holds at some state, then it holds at all its successor states. State invariants characterize sets that are over-approximations of reachability sets. Notice that composition by using glue operators preserves the invariants of their arguments.
- *Deadlock-freedom* means that at least one action is enabled from any reachable state. It is the weakest progress property. Notice that composition of deadlock-free components does not give a deadlock-free component, in general.

Restriction to these two types of properties is motivated by pragmatic reasons. Most trustworthiness requirements can be captured as the conjunction of essential properties. Preservation for other types of properties such as individual deadlock-freedom of components, liveness, and quantitative properties seems to be far beyond the current state of the art. From now on, the term "correctness" will refer to satisfaction of these two types of properties.

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; and

- *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).

5.5.2 Horizontal Correctness

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

Property enforcement

Property enforcement consists in applying *architectures* to restrict 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 a coordination to satisfy the property.

Architectures depict design principles, paradigms that can be understood by all, allowing 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 ensuring both functional and non-functional properties, for example, fault-tolerant architectures, architectures for resource management and 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, distributed algorithms, schedulers, etc.

An architecture is a context $\mathrm{A}(n)[\mathrm{X}] = \mathrm{gl}(n)(\mathrm{X}, \mathrm{D}(n))$, where $\mathrm{gl}(n)$ is a glue operator and $\mathrm{D}(n)$ a set of coordinating components, with a characteristic property $\mathrm{P}(n)$, parameterized by an integer n such that:

- $\mathrm{A}(n)$ transforms a set of components $\mathrm{C}_1, \ldots, \mathrm{C}_n$ into a composite component $\mathrm{A}(n)[\mathrm{C}_1, \ldots, \mathrm{C}_n] = \mathrm{gl}(n)(C_1, \ldots, \mathrm{C}_n, \mathrm{D}(n))$, by preserving essential properties of the composed components, that is,
  1. *Deadlock-freedom*: if components C_i, are deadlock-free then $\mathrm{A}_n[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ is deadlock-free too;
  2. *Invariants*: any invariant of a component C_i is also an invariant of $\mathrm{A}_n[\mathrm{C}_1, \ldots, \mathrm{C}_n]$.
- $\mathrm{A}(n)[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ meets the *characteristic property* $\mathrm{P}(n)$.

Architectures are partial operators as the interactions of gl should match actions of the composed components. They are solutions to a coordination problem specified by P by using a particular set of interactions specified by gl. For instance, for 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 (e.g., mutual exclusion, scheduling policy, clock synchronization).

Property Composability:

In a design process, it is often necessary to combine more than one architectural solution on a 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 feature interaction in the integrated solution. Non-interaction of features is characterized below as property composability based on our concept of architecture.

Consider two architectures $\mathrm{A}_1, \mathrm{A}_2$, enforcing respectively properties $\mathrm{P}_{\mathrm{A}1}, \mathrm{P}_{\mathrm{A}2}$ on a set of components $\mathrm{C}_1, \ldots, \mathrm{C}_n$. That is, $\mathrm{A}_1[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ and $A_2[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ satisfy respectively the properties $\mathrm{P}_{\mathrm{A}1}$, $\mathrm{P}_{\mathrm{A}2}$. Is it possible to find an architecture $\mathrm{A}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ 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 satisfy both properties?

A theoretical solution to this problem can be formulated by showing that the set of architectures satisfying a given property for a given set of components $\{C_1, \ldots, \mathrm{C}_n\}$ is a lattice equipped with a partial relation $\langle$ between architectures satisfying a given property. The top element of the lattice is the most liberal architecture, that is, the architecture enforcing no property. The bottom element represents all the coordination constraints that lead to deadlocked systems and thus do not correspond to architectures. The partial order $\langle$ is defined by: $\mathrm{A}_1 \langle \mathrm{A}_2$ if $\mathrm{A}_1[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ satisfies a property P then $\mathrm{A}_2[\mathrm{C}_1, \ldots, \mathrm{C}_n]$ satisfies P. The architecture satisfying both $\mathrm{P}_{\mathrm{A}1}$ and $\mathrm{P}_{\mathrm{A}2}$ can be defined as $\mathrm{A}_1 \oplus \mathrm{A}_2$, where $\oplus$ is a partial operation denoting the greatest lower bound of A_1 and A_2 if it is different from the bottom element of the architecture lattice.

In [4] properties of the $\oplus$ operation on glue operators are studied and applied for building incrementally correct-by-construction components.

To put this vision for horizontal correctness into practice, we need to develop a repository of reference architectures. The repository should classify existing architectures according to their characteristic properties. There exists a plethora of results on distributed algorithms, protocols, and scheduling algorithms. Most of these results focus on principles of solutions and discard essential operational details. Their correctness is usually established by assume/guarantee reasoning: a characteristic global property is implied from properties of the integrated components. This is enough to validate the principle but does not entail correctness of particular implementations. Often, these principles of solutions do not specify concrete coordination mechanisms (e.g., in terms of operational semantics), and ignore physical resources such as time, memory, and energy. The reference architectures included in the repository, should be

- described as executable models in the chosen component framework;
- proven correct with respect to their characteristic properties; and
- characterized in terms of performance, efficiency, and other essential non-functional properties.

For enhanced reuse, reference architectures should be 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 taxonomy 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?

The example of the decomposition of fault-tolerant architectures into basic features can be applied to other architectures. Time-triggered architectures usually combine a clock synchronization algorithm and a leader election algorithm. Security architectures integrate a variety of mitigation mechanisms for intrusion detection, intrusion protection, sampling, embedded cryptography, integrity checking, etc. Communication protocols combine sets of algorithms for signaling, authentication, and error detection/correction. Is it possible to obtain by incremental composition of features and their characteristic properties, architectural solutions that meet given global properties? This is an open problem whose solution would greatly enhance our capability to develop systems that are correct-by-construction and integrate only the features needed for a target characteristic property.

5.5.3 Vertical Correctness

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

This transformation consists in refining the actions of components $\mathrm{C}_1,\ldots,\mathrm{C}_n$ to obtain new components $\mathrm{C}'_1,\ldots,\mathrm{C}'_n$. Action refinement in some component C_i, consists in replacing an action a by its implementation as a sequence of actions str(a) ... cmp(a). The first and last elements 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 $\mathrm{gl}(\mathrm{C}_1,\ldots,\mathrm{C}_n)$, 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.

The top of Figure 5.4 depicts, in the form of a Petri net, the principle for this refinement which associates with $\mathrm{gl}(\mathrm{C}_1,\mathrm{C}_2)$ the system $\mathrm{gl}'(\mathrm{C}'_1,\mathrm{C}'_2,\mathrm{D})$, 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, 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

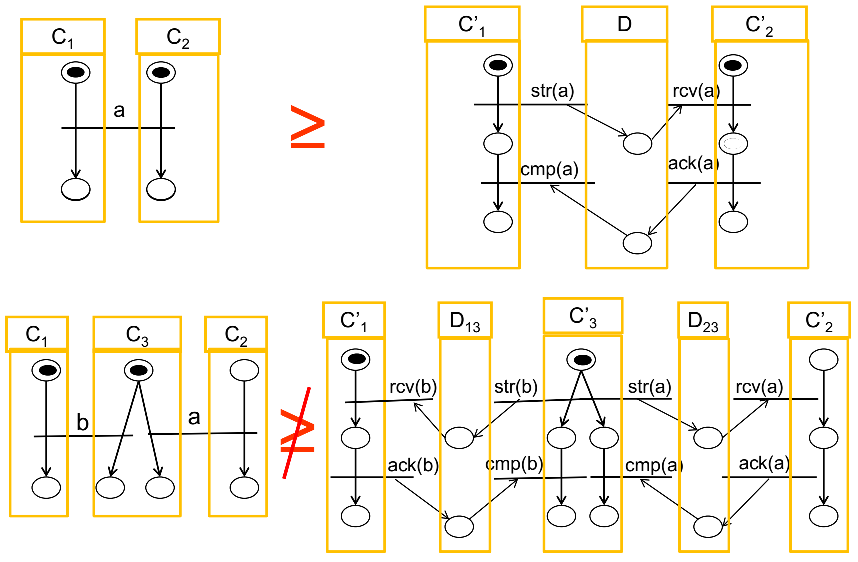


Fig. 5.4 Interaction refinement by using Send/Receive primitives.

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.

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

(1) All traces of S′ are traces of S modulo the observation criterion associating with each interaction of S the corresponding finishing interaction in S′;
(2) If S is deadlock-free then S′ is deadlock-free; and
(3) The relation $\geq$ is stable under substitution, that is for any systems S_1, S_2 and any architecture A: $S_1 \geq S_2$ implies $A[S_1, X] \geq A[S_2, X]$ where X is an arbitrary tuple of components.

Notice that in this definition we require only the inclusion of the observable traces. Nonetheless, condition 2 guarantees preservation of deadlock-freedom and precludes emptiness of the set of the traces of S′. The stability of $\geq$ under substitution is essential for reusing

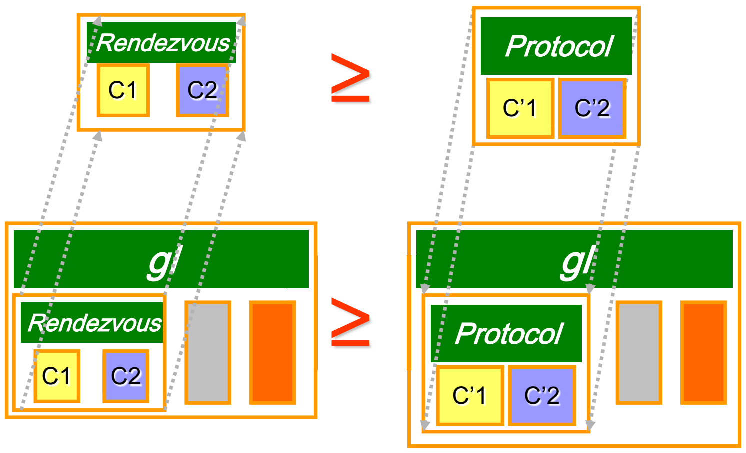


Fig. 5.5 Stability of the refinement relation under substitution.

refinements and correctness-by-construction. As a rule, proving this property requires non-trivial inductive reasoning on the structure of the terms representing systems. Figure 5.5 depicts the stability rule.

Preservation of semantics under action refinement has been extensively studied (e.g., [34]). Nevertheless, existing results have been developed for less expressive frameworks (e.g., process algebras). As already pointed out in the literature, a key issue for refinement stability is how causality and conflict relations between interactions of A[S_1,X] are inherited in A[S_2, X]. A conflict resolution between two interactions a_1, a_2 of A[S_1, X] is resolved by choosing and executing atomically one of these actions. The same conflict in A[S_2, X] is resolved between interactions str(a_1) and str(a_2) without taking into account the possibility of completion of the corresponding execution sequence. The example at the bottom of Figure 5.4 shows non-stability of the refinement relation provided at the top. The refinement of interactions a and b in the system consisting of three components C_1, C_2, and C_3 gives a system with a potential deadlock. For the initial state shown in Figure 5.4, only interaction b is possible while the refined system can block if bgn(a) is selected and executed.

To attain this vision for vertical correctness, we need to develop component-refinement theory and tools to allow moving downstream in the abstraction hierarchy from application software to an implementable system model. Application software usually involves high-level primitives supporting abstractions such as:

- atomicity of interactions between components — in particular multiparty interaction; and
- a logical notion of time assuming zero-time actions and synchrony of execution with respect to the physical environment.

The generated system model should be obtained as a refinement of application software parameterized by a mapping associating: (1) components of application software to processing elements of the platform; (2) data of the application software with memories of the platform; and (3) interactions of the application software with execution paths or protocols of the platform.

5.6 Putting Rigorous System Design into Practice in BIP

We show how the principles and technical ideas advocated in previous sections have been implemented in a system design flow supported by the BIP framework.

Figure 5.6 illustrates a rigorous system design flow that uses BIP as a unifying semantic model to ensure coherency between the different design steps. The design flow involves four distinct steps that translate the application software into a BIP model and progressively derive an implementation by applying source-to-source transformations. These ensure vertical correctness by construction as the obtained BIP models are refinements of the original model. In particular, they preserve the application software's safety properties. The D-Finder compositional verification tool is used for checking essential safety properties of the application software.

The translation of the application software into BIP is by embedding as explained in 5. The development of embedding tools focuses on the coordination mechanisms of the source language and the definition of adequate interfaces for atomic components. It encapsulates and reuses the application software's data structures and functions. BIP model generators are available for DSLs such as Lustre, Simulink, nesC, and

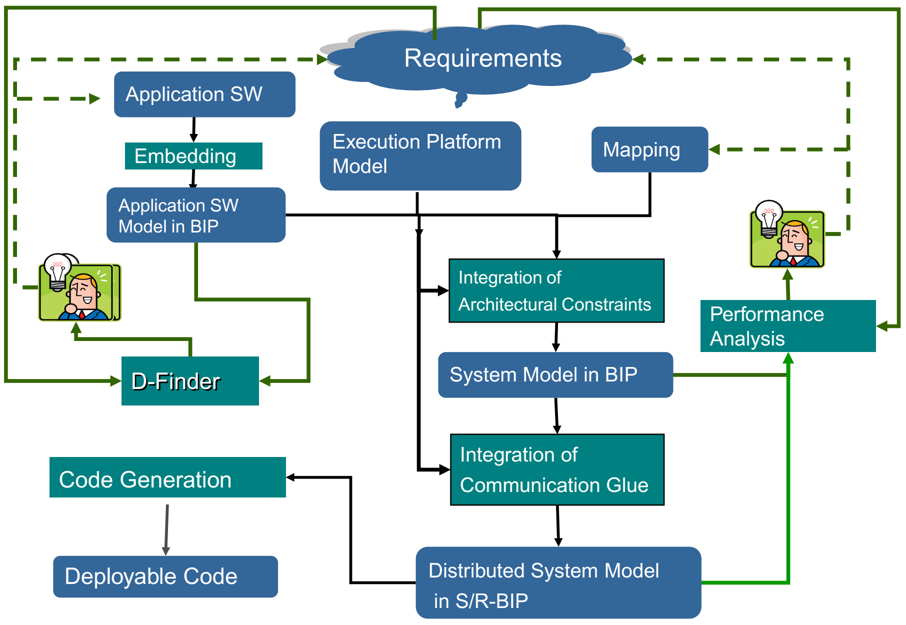


Fig. 5.6 BIP design flow. An implementation — that is, deployable code — is generated from the application software, a model of the execution platform, and a mapping.

DOL. The generated BIP models preserve the structure of the initial programs, their size is linear with respect to the initial program size, and they are easy for the system developers to understand.

Functional correctness of the application software model can be checked using the D-Finder tool. D-Finder applies symbolic compositional verification heuristic techniques by using invariants. It computes increasingly stronger invariants for composite components as conjunctions of invariants of atomic components' and interaction invariants. The former are computed by application of static analysis techniques to atomic components. The latter are computed from abstractions of the composite component to be verified. They characterize the way glue operators restrict the product space of the composed atomic components.

We recently improved this method to take advantage of the incremental system design process, which proceeds by adding new interactions to a component under construction. Each time a new interaction is added, it is possible to verify whether the resulting component violates a given property and so discover design errors as they appear. The incremental verification technique uses sufficient conditions to ensure the preservation of invariants when new interactions are added during the component construction process. If these conditions are not satisfied, D-Finder generates new invariants by reusing invariants of the constituent components. Reusing invariants considerably reduces the verification effort.

Experimental results on standard benchmarks show that D-Finder can run exponentially faster than existing monolithic verification tools, such as NuSMV.

To generate system models and implementations from the application software, we use an extensible toolset including source-to-source transformers and compilers as depicted in Figure 5.7. The BIP toolset offers several compilation chains, targeting different execution platforms. To implement BIP on single-core platforms we use engines — dedicated middleware for the execution of the C++ code generated

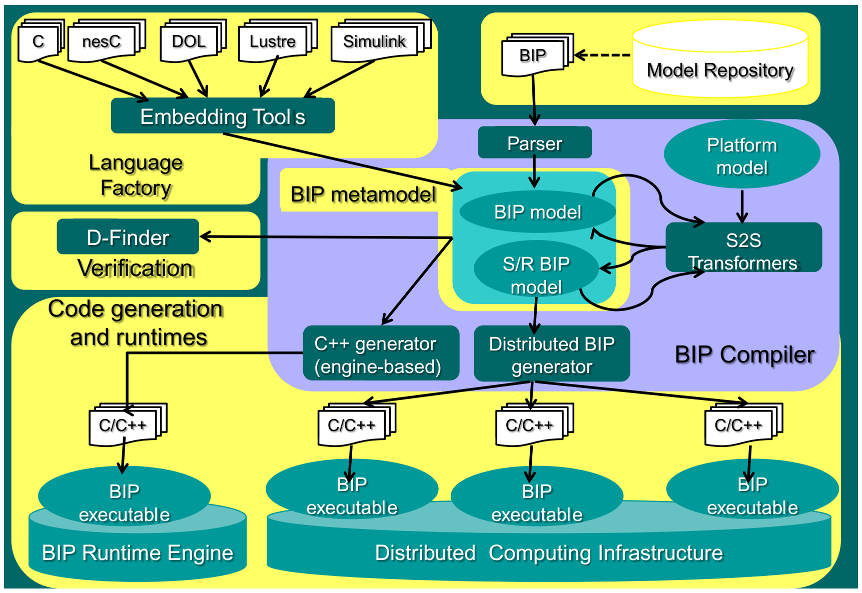


Fig. 5.7 BIP toolset. It includes translators from various programming models, verification tools, source-to-source transformers, and C/C++ code generators for BIP models.

from BIP descriptions. The BIP toolset currently provides two engines: one for real-time single-thread and one for multi-thread execution. For multi-thread execution, each atomic component is assigned to a thread, with the engine itself being a thread. Communication occurs only between atomic components and the engine — never directly between different atomic components.

Source-to-source transformations in BIP are intended to derive from the application software model, correct system models taking into account features of the execution platform. They combine hardware-driven and distribution-driven transformations.

Hardware-driven transformations allow the generation of a system model from hardware architecture and a mapping associating components of the source model with processing elements of the platform and data of the source model with memory of the hardware platform. The generation process is parameterized by choices regarding possible scheduling and arbitration policies. It also uses a library of hardware-dependent components providing models of physical and middleware components [8].

Distribution-driven transformations generate S/R-BIP models, a subclass of models in which protocols using Send/Receive primitives replace multiparty interactions [7].

We explain below the principle of distribution-driven transformations. These are applied to BIP models with a user-defined partition of their interactions. The number of blocks of the partition determines the degree of parallelism between interactions. The initial model is transformed into an S/R-BIP model structured in three hierarchically structured layers. Each layer is obtained by a corresponding transformation:

1. The component layer consists of the original model's atomic components in which each port involved in strong interactions is replaced by a pair consisting of a send and a receive port.
2. The interaction protocol layer consists of a set of components, each of which manages a block of the interactions' partition. Each component detects whether the associated interactions

are enabled and executes them after resolving conflicts either locally or with assistance from the third layer.

3. The conflict resolution protocol layer implements a distributed algorithm for resolving conflicts as requested by the interaction protocol layer. It basically solves a committee coordination problem, that can be solved by using either a fully centralized arbiter or a distributed one e.g., token-ring or dining philosophers algorithm.

These transformations have been proven correct by construction. They are based on the generic and modular reuse of protocols described as architectures in BIP. The degree of parallelism of the distributed model depends on the choice of both the interactions' partition and the conflict resolution protocol.

Given the three-layer S/R-BIP model and a mapping of its atomic components on processors, we can generate either an MPI program or a set of plain C/C++ programs that use TCP/IP communication. This generation process statically composes atomic components running on the same processor to obtain a single observationally equivalent component, and reduce coordination overhead at runtime.

6

A System-Centric Vision for Computing

In this section, we discuss four issues raised by a system-centric vision for computing.

- How can computing systems engineering be linked to other systems engineering theories and practices? Establishing links can mutually enrich and cross-fertilize engineering disciplines. Furthermore, this is essential for matching the needs for increasing immersion of the cyber-world in human and physical environments.
- Is design central to computing? Today, large computing systems are developed in an ad hoc manner without caring so much about disciplined and rigorous design. Currently, empiricism is gaining ground and becoming the dominant doctrine in large system development. In our opinion, sooner or later, it will hit the wall of trustworthy and cost-effective systems integration.
- What are the limits of understanding and mastering the Cyber-world? Awareness of current limitations should allow the finding of avenues for overcoming them as much as possible or mitigating their effects.

- What type of theory is the most adequate for system design? Can mathematical elegance and practical relevance be reconciled?

6.1 Linking Computing to Other Disciplines

The increasing immersion and interaction of computing systems with both physical and societal systems inevitably poses the problem of the very nature of computing and its relationship with other scientific disciplines. How can the interplay between different types of systems (physical, computing, biological) be understood and mastered? To what extent can multi-disciplinary approaches enrich computing with new paradigms and concepts?

Computing is a scientific discipline in its own right with its own concepts and paradigms. It deals with problems related to the representation, transformation, and transmission of information. Information is an entity distinct from matter and energy. It is defined as a relationship involving the syntax and the semantics of a given language. By its nature, it is immaterial but needs media for its representation. The concept of information should not be confused with "syntactic information" which is a quantity characterizing the minimal amount of resources needed for a representation (e.g., number of bits or the complexity of computational resources).

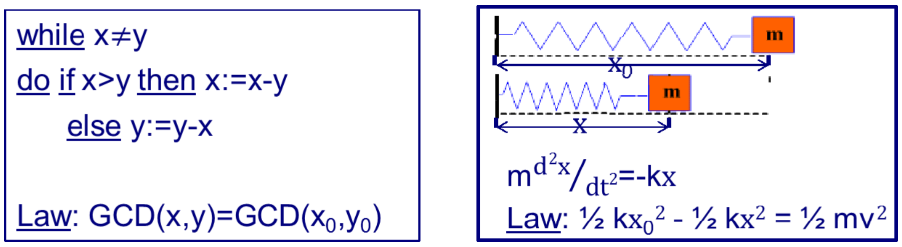
Computing is not merely a branch of mathematics. Just as any other scientific discipline, it seeks validation of its theories on mathematical grounds. But mainly, and most importantly, it develops specific theory intended to explain and predict properties of systems that can be tested experimentally.

The advent of embedded systems brings computing closer to physics. Linking physical systems and computing systems requires a better understanding of differences and points of contact between them. Is it possible to define models of computation encompassing quantities such as physical time, physical memory, and energy? Significant differences exist in the approaches and paradigms adopted by the two disciplines.

Classical physics is *primarily* based on continuous mathematics while computing is rooted in discrete non-invertible mathematics. It focuses mainly on the discovery of laws governing the physical world as it is, while computing is rooted in a priori concepts and deals with building artifacts. Physical laws are declarative by their nature. Physical systems are specified by differential equations involving relations between physical quantities. The essence of basic physical phenomena can be captured by simple linear laws. They are, to a large extent, deterministic and predictable. Synthesis is the dominant paradigm in physical systems engineering. We know how to build artifacts meeting given requirements (e.g., bridges or circuits), by solving equations describing their behavior. By contrast, state equations of very simple computing systems, such as an RS flip-flop, do not admit linear representations in any finite field. Computing systems are described in executable formalisms such as programs and machines. Their behavior is intrinsically non-deterministic. Non-decidability of their essential properties implies poor predictability.

Despite these differences, both disciplines share a common objective which is the study of dynamic systems. We attempt below a comparison for a simplified notion of dynamic system. A dynamic system can be described by a set of equations of the form $X' = f(X, Y)$ where X' is a "next state" variable, X is the current state and Y is the current input of the system. For physical systems the variables are usually real-valued functions of a single real-valued time parameter while for computing systems the variables range over discrete domains. The next state variable X' is typically $\mathrm{d}X/\mathrm{dt}$ for physical systems, while for computing systems it denotes the state of the system in the next computation step.

Figure 6.1 shows a program computing the GCD of two integer variables and a mass–spring system. The operational semantics of the programming language associate with the program a next-state function, while the solution of the differential equation describes the movement of the mass. The reachable states of the program are characterized by the invariant $\mathrm{GCD}(x, y) = \mathrm{GCD}(x_0, y_0)$ where x_0, y_0 are the initial values of x and y, respectively. This invariant can be used to prove that the program is correct if it terminates. In exactly the same manner, the



```
while x≠y
do if x>y then x:=x-y
     else y:=y-x

Law: GCD(x,y)=GCD(x0,y0)
```

$m\,{}^{d^2x}/_{dt^2}=-kx$

Law: $½\,kx_0^2 - ½\,kx^2 = ½\,mv^2$

Fig. 6.1 Dynamic systems and laws characterizing a GCD program and a spring–mass system.

law of conservation of energy $1/2kx_0^2 - 1/2kx^2 = 1/2\ kv^2$ characterizes the movement of the mass as a function of its distance from the origin, its initial position, and its speed.

This example illustrates remarkable similarities and also highlights some significant differences. Computing systems can be certainly considered as scientific theories. However, they are subject to specific laws that are not easy to discover. Computing program invariants is a well-known non-tractable problem. On the contrary, all physical systems, and electromechanical systems in particular, are subject to uniform laws governing their behavior. Another important difference is that for physical systems, variables are all functions of a single time parameter and this drastically simplifies their specification and analysis. For instance, operations on variables are defined on streams of values while operations of computing system variables are on single values. From this point of view, physical systems are closer to synchronous computing systems. However, models of computation do not have a built-in notion of time. The latter can be, of course, represented as a state variable (clock). Nonetheless, clock synchronization can be achieved only at some degree of precision and is algorithmically expensive. As already discussed in Section 5.2.2, this computational notion of time as a state variable explicitly handled by a system, significantly differs from physical time modeled as an ever-increasing time parameter.

Computing enriches our knowledge with theory and models enabling a deeper understanding of discrete dynamic systems. It proposes a constructive and operational view of the world which complements the classic declarative approach adopted by physics.

Living organisms intimately combine physical and computational processes that have a deep impact on their development and evolution. They share several characteristics with computing systems such as the use of memory, the distinction between hardware and software, and the use of languages. However, some essential differences exist. Computation in living organisms is robust, has built-in mechanisms for adaptivity and, most importantly, it allows the emergence of abstractions and concepts.

I believe that these differences delimit a gap that is hard to be filled by actual models of computing systems. At the same time, they determine challenges for research in computing and can inspire groundbreaking paradigm shifts that could liberate computing from its current limitations.

6.2 Rigorous Design versus Controlled Experiments

The need for rigorous design is sometimes directly or indirectly questioned by developers of large-scale systems (e.g., web-based systems). These systems of overwhelming complexity have been built incrementally in an ad hoc manner. Their behavior can be studied only empirically by testing and through controlled experiments. The key issue is to determine trade-offs between performance and cost by iterative tuning of parameters. Currently, a good deal of research on web-based systems privileges an analytic approach that aims to find laws that generate or explain observed phenomena rather than to investigate design principles for achieving a desired behavior. It is reported in [17] that "On line companies .... don't anguish over how to design their Web sites. Instead they conduct controlled experiments by showing different versions to different groups of users until they have iterated to an optimal solution" .

This trend calls for two remarks.

- In contrast to physical sciences, computing is predominantly synthetic. Its main goal is to develop theory, methods, and tools for building trustworthy and optimized systems. Considering the cyber-universe as a "given reality" driven by its own laws and privileging analytic approaches for their discovery and study, is epistemologically absurd and can have only limited scientific impact. The physical world is the result of a long and well-orchestrated evolution. It is governed by simple laws. To quote Einstein, "*the most incomprehensible thing about the world is that it is at all comprehensible*". The trajectory of a projectile under gravity is a parabola which is a very simple and easy to understand law. There is nothing similar about the traces of computing systems.

- Ad hoc and experimental approaches can be useful only for optimization purposes. Trustworthiness is a qualitative property and by its nature, it cannot be achieved by the fine tuning of parameters. Small changes can have a dramatic impact on safety and security properties.

6.3 The Limits of Understanding and Mastering the Cyber-world

As pointed out in the Introduction, proceduralization of declarative specifications is intractable. This seriously limits our ability to transform requirements into provably correct programs. An interesting question is finding domain-specific declarative languages whose expressiveness does not completely compromise tractability of synthesis.

Abstraction hierarchies are a methodological simplification of the real world to cope with its inherent complexity and better figure our relevant properties at different levels of observation. They are used in all scientific disciplines to determine successive levels of granularity of observation at which system properties can be studied (Figure 6.2). Theory should allow the prediction of how properties at some level are reflected upstream or downstream in the hierarchy. When we move to a higher abstraction level, new properties may emerge which are intrinsic to this level. Emerging properties at some level cannot be inferred only from properties of lower levels. Mutual exclusion on a set of tasks cannot be inferred from individual properties of the tasks for the same reason as the properties of a molecule of water cannot be solely deduced from properties of hydrogen and oxygen atoms.

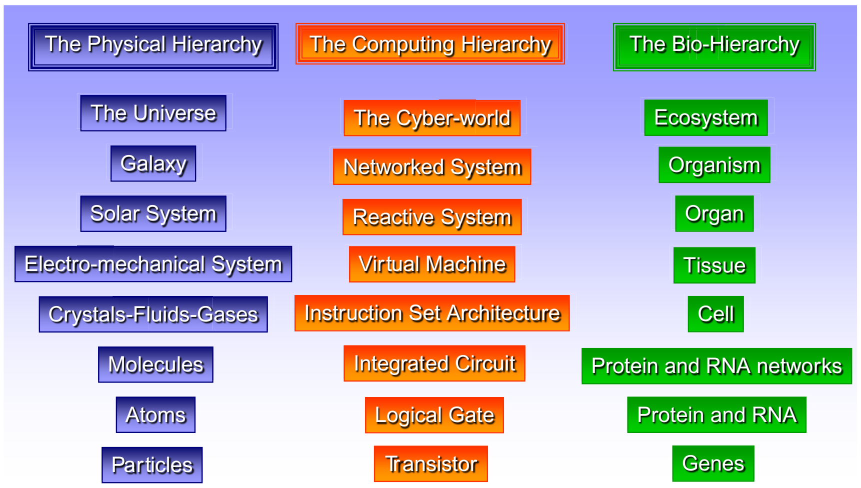


Fig. 6.2 Abstraction hierarchies for physical, computing, and biological systems.

Within the computing systems globe, it is essential to develop theory methods and tools for climbing up and down the cyber-hierarchy. How can energy-efficiency influence the way we are programming? Which models most adequately feature system behavior at each abstraction level? How can models and their properties, at different abstraction levels, be related through well-founded abstraction relations? These problems remain unsolved and will probably remain open for quite a long time. Their answers will largely determine our ability to master the cyber-physical world.

Naturally, discreteness of computation and uncertainty seriously compromise our ability to guarantee correctness. Traditional engineering amply relies on robust system behavior: small changes of parameters within an interval of values have commensurable effects. Owing to the discreteness of computation, qualitative properties are not robust. Safety or security properties may be jeopardized by the slightest hardware or software modification. Even quantitative properties such as performance are not robust because of non-determinism and uncertainty (e.g., timing anomalies).

We need theory and methods for enhancing robustness of computing systems. For trustworthiness properties, a mitigation of failures can be achieved either by using redundancy techniques or monitoring at runtime. For quantitative properties, we need a deeper understanding of the interplay between their predictability and uncertainty.

6.4 The Quest for Mathematically Tractable and Practically Relevant Theory

The proper goal of theory in any field is to make models that accurately describe real systems. Models can be used to explain phenomena and predict system behavior. They should help system builders do their jobs better.

There is currently a harmful separation between theoretical and applied research in computing.

Theoretical research has a predilection for mathematically clean theoretical frameworks, no matter how relevant they can be. Many theoretical frameworks and results are "low-level" and have no point of contact with real computing. They are mainly based on transition systems which are structure-agnostic and cannot account for phenomena such as coordination and communication. They can be badly lifted from semantic to syntactic level. They certainly can provide a deep insight into hard problems raised by correct system design but they fail to provide a basis for practicable and scalable techniques. We believe that theoretical research should be refocused to address system design challenges at the right level of abstraction by eventually sacrificing mathematical elegance for practicality.

A quite different attitude is adopted by practically oriented research. Existing frameworks for programming or modeling real systems are constructed in an ad hoc manner. They are obtained by putting together a large number of semantically unrelated constructs and primitives. It is practically impossible to get any rigorous formalization and build any useful theory for such frameworks. It is also problematic to assimilate and master their concepts by reading manuals of hundreds of pages. System development remains by far an art owing to unharnessed expressiveness and the fuzzy semantics of existing frameworks. Lack of rigorousness is routinely compensated by tricks, hacks, and other magics that are beyond any scientific explanation and analysis.

We need theoretical frameworks that are expressive, make use of a minimal set of high-level concepts and primitives for system description, and that are amenable to formalization and analysis.

Is it possible to find a mathematically elegant and still practicable theoretical framework for computing systems? As explained, we cannot expect to have theoretical settings as beautiful and as powerful as those for physical systems. One profound reason is that computing systems are human artifacts while the physical systems are the result of a very long evolution.

In contrast to physical sciences which focus mainly on the discovery of laws, computing should focus mainly on developing theory for system "constructivity" and predictability. Design is central to the discipline. Awareness of its centrality is a chance to reinvigorate research, and build new scientific foundations matching the needs for increasing system integration and new applications.

There already exists a large body of constructivity results such as algorithms, architectures, and protocols. Their application allows correctness for (almost) free. How can global properties of a composite system be effectively inferred from the properties of its constituents? This remains an old open problem that urgently needs answers. Failure to bring satisfactory solutions will be a limiting factor for system integration. It would also mean that computing is definitely relegated to second-class status with respect to other scientific disciplines.

Acknowledgments

The presented vision presented amply relies on theory, and experimental results developed by the BIP team. I would like to thank in particular Saddek Bensalem, Marius Bozga, Jacques Combaz, and Simon Bliudze for their contribution to the development of the BIP framework.

The paper has benefited from constructive comments and criticism by Peter Denning and two anonymous reviewers.

References

[1] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," *EMSOFT*, pp. 229–238, 2010.

[2] K. J. Astrom and B. Wittenmark, *Adaptive Control.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2nd ed., 1994.

[3] Beck, Kent; et al., *Manifesto for Agile Software Development.* Agile Alliance, 2001.

[4] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, "Incremental component-based construction and verification using invariants," in *FMCAD*, pp. 257–256, Lugano, Switzerland, October 20–23 2010.

[5] S. Bliudze and J. Sifakis, "A Notion of Glue Expressiveness for Component-Based Systems," *Lecturer Notes in Computer Science*, vol. 5201, pp. 508–522, 2008.

[6] P. Bogdan and R. Marculescu, "Towards a science of cyber-physical systems design," *Proceeding ICCPS '11 Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pp. 99–108.

[7] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," *EMSOFT*, pp. 209–218, 2010.

[8] P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, and K. Huang, "Rigorous system level modeling and analysis of mixed HW/SW systems," *MEMOCODE*, pp. 11–20, 2011.

[9] M. Butler, M. Leuschel, S. L. Presti, and P. Turner, "The use of formal methods in the analysis of trust (Position Paper)," *Lecture Notes in Computer Science*, vol. 2995/2004, pp. 333–339, 2004.

[10] G. Buttazzo, *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications.* Real-Time Systems Series, Springer, vol. 24, 2001.

[11] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *CACM*, vol. 52, no. 11, November 2009.

[12] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, 2008.

[13] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE* (special issue on CPS), vol. 100, no. 1, pp. 13–28, January 2012.

[14] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 97)*, pp. 169–183, IBM Press, 1997.

[15] D. Q. Goldin, S. A. Smolka, P. C. Attie, and E. L. Sonderegger, "Turing machines, transition systems, and interaction," *Information and Computation*, vol. 194, no. 2, pp. 101–128, November 2004.

[16] N. Halbwachs, *Synchronous Programming of Reactive Systems.* Kluwer Academic Pub., 1993.

[17] T. Hannay, "The controlled experiment," in *This Will Make You Smarter*, (J. Brockman, ed.), Happer Perennial.

[18] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Communications of the ACM*, vol. 55, no. 7, July 2012.

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

[20] H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, February 2012.

[21] International Council on Systems Engineering (INCOSE), *Systems Engineering Handbook* Version 3.1. August 2007.

[22] H. Kopetz, "The rationale for time-triggered ethernet," *Proceedings of the 29th IEEE Real-Time Systems Symposium.*

[23] H. Kopetz, R. Obermaisser, C. E. Salloum, and B. Huber, "Automotive software development for a multi-core system-on-a-chip," *Fourth International Workshop on Software Engineering for Automotive Systems (SEAS'07)*, 2007.

[24] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[25] J. Magee and J. Kramer, "Dynamic structure in software architectures," in *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT 96)*, pp. 3–14, ACM Press, 1996.

[26] S. Maoz, D. Harel, and A. Kleinbort, "A compiler for multimodal scenarios: Transforming LSCs into AspectJ, September 2011," *Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4.

[27] D. H. Mcknight and N. L. Chervany, "The meanings of trust," *Trust in Cyber-Societies-LNAI*, pp. 27–54, 2001.

[28] R. A. D. Millo, R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," *CACM*, vol. 22, no. 5, May 1979.

[29] D. K. Mulligany and F. B. Schneider, “Doctrine for Cybersecurity,” Technical Report, Cornell University, May 2011.

[30] Rigorous Design of Component-Based Systems — The BIP Component Framework: http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.

[31] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, “A definition and classification of timing anomalies,” in *Sixth International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, July 4 2006.

[32] J. Sifakis, “A framework for component-based construction,” in *IEEE International Conference on Software Engineering and Formal Methods (SEFM05)*, pp. 293–300, Koblenz, September 7–9 2005.

[33] SOFTWARE 2015: A national software strategy to ensure U.S. security and competitiveness report of the 2nd national software summit, April 29, 2005.

[34] R. J. van Glabbeek, “Ursula Goltz: Refinement of actions and equivalence notions for concurrent systems,” *Acta Information*, vol. 37, no. 4/5, pp. 229–327, 2001.

[35] J. van Leeuwen and J. Wiedermann, “The turing machine paradigm in contemporary computing,” in *Mathematics Unlimited — 2001 and Beyond*, (B. Enquist and W. Schmidt, eds.), LNCS, Springer-Verlag, 2000.

[36] D. A. Watt, B. A. Wichmann, and W. Findlay, “Ada: Language and Methodology,” 1987.