Methods and Tools for Component-based System Design

Joseph Sifakis
Verimag

Extended Abstract

Traditional engineering disciplines such as civil or mechanical engineering are based on solid theory for building artefacts with predictable behavior over their life-time. In contrast, we lack similar constructivity results for computing systems engineering: computer science provides only partial answers to particular system design problems. With few exceptions, predictability is impossible to guarantee at design time and therefore, *a posteriori* verification remains the only means for ensuring their correct operation.

System design is empirical due to our inability to predict the behavior of application software running on a given platform. Usually, systems are built by reusing and assembling components that are simpler sub-systems. This is the only way to master complexity and to ensure correctness of the overall design, while maintaining or increasing productivity. However, system level integration becomes extremely hard because components are usually highly heterogeneous: they have different characteristics, are often developed using different technologies, and highlight different features from different viewpoints. Other difficulties stem from current design approaches, based on expertise and experience of design teams. Naturally, designers attempt to solve new problems by reusing, extending and improving existing solutions proven to be efficient and robust. This favors component reuse and avoids re-inventing designs. Nevertheless, on a longer term perspective, this may also be counter-productive: designers are not always able to adapt to new requirements. Moreover, they exclude in advance better solutions simply because they do not fit their know-how and expertise.

System design is the process leading to a mixed software/hardware system meeting given requirements. It involves the development of application software taking into account features of an execution platform. The latter is defined by its architecture involving a set of processors equipped with hardware-dependent software such as operating systems as well as primitives for coordinating computation and interaction with the external environment.

System design radically differs from pure software design in that it must take into account not only functional but also extra-functional requirements regarding the use of resources of the execution platform such as time, memory and energy. Meeting such requirements is essential, in particular for embedded systems. It requires evaluation of the impact of design choices on the overall system behavior. This also implies a deep understanding of the interaction between application software and the underlying execution platform. We currently lack approaches for modelling mixed hardware/software systems. There are no rigorous techniques for deriving global models of a given system from models of its application software and its execution platform [1]

A system design flow consists of steps starting from requirements and leading to an implementation. It involves the use of methods and tools for progressively deriving an implementation by making adequate design choices. It must meet the following essential properties:

1. Correctness: This means that the designed system meets its requirements. Correctness with respect to qualitative requirements can be achieved by application of verification techniques. Quantitative requirements such as performance and dependability are usually checked by analysis techniques. Ensuring correctness requires that the design flow relies on models with well-defined semantics. The models should consistently encompass system description at different levels of abstraction from application software to implementation. For checking quantitative requirements, we need models where resources are first class concepts. Unfortunately, existing formalisms fail to jointly encompass qualitative and quantitative aspects. System designers use different semantically unrelated formalisms e.g. for programming, hardware description, performance evaluation, fault analysis. This mitigates efficiency of the validation process as it hard to relate facts established for different models of the same system.

978-3-9810801-7-9/DATE11/©2011 EDAA

2. Productivity: This can be achieved by system design flows
    - providing high level domain-specific languages for ease of expression;
    - allowing reuse of components and the development of component-based solutions;
    - integrating tools for programming, validation and code generation.

3. Parsimony: The design flow should not enforce any particular programming or execution model. Very often system designers privilege specific programming models or implementation principles that exclude in advance efficient solutions. They program in low level languages that do not help discover parallelism or non determinism and enforce strictly sequential execution. For instance, programming multimedia applications in plain C may lead to designs obscuring the inherent functional parallelism and involving built-in scheduling mechanisms that are not optimal. Most system design flows privilege a unique programming model together with an associated compilation chain adapted for a given execution model. For example, synchronous system design relies on synchronous programming models and usually targets hardware or sequential implementations on single processors [2]. Alternatively, real-time programming based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms [3]. It is essential that designers use adequate programming models. Design choices should be driven only by system requirements to obtain the best possible implementation.

We call rigorous a design flow which allows guaranteeing validities of its requirements. A rigorous design flow should be:

1. Model-based, that is all software and system descriptions used along the flow should be based on a single semantic model. This is essential for maintaining the overall coherency by guaranteeing that a description at step n meets essential properties of a description at step n + 1. This means in particular that the semantic model is expressive enough to directly encompass various types of component heterogeneity arising along the design flow [4]:
    - Heterogeneity of computation: The semantic model should encompass both synchronous and asynchronous computation by using adequate coordination mechanisms. This should allow in particular, modeling mixed hardware/software systems.
    - Heterogeneity of interaction: The semantic model should enable natural and direct description of various mechanisms used to coordinate execution of components including semaphores, rendezvous, broadcast, method call, etc.
    - Heterogeneity of abstraction: The semantic model should support the description of a system at different abstraction levels from its application software to its implementation. In particular, it should be possible to establish a clear correspondence between the description of an untimed platform-independent behavior and the corresponding timed and platform-dependent implementation.
2. Component-based, that is it provides primitives for building composite components as the composition of simpler components. 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. For instance, if the composition is by strong synchronization (rendezvous), modeling broadcast requires an extra component to choose amongst the possible strong synchronizations a maximal one. We need frameworks providing families of composition operators for natural and direct description of coordination mechanisms such as protocols, schedulers and buses.
3. Correct-by-construction that is it should rely on tractable theory for guaranteeing incrementally correctness to avoid as much as possible monolithic verification. Such a theory is based on two types of rules [1]:
    - Compositionality rules for inferring global properties of composite components from the properties of composed components e.g. if a set of components are deadlock-free then for a certain type of composition the obtained composite components is deadlock-free too. A special and very useful case of compositionality is when a behavioral equivalence relation between components is a congruence [5]. In that case, substituting a component in a system model by a behaviorally equivalent component leads to an equivalent model. Today we badly lack compositionality theory for progress properties as well as extra-functional properties.
    - Composability rules ensuring that essential properties of a component are preserved when it is used to build composite components. Consider for instance, the following two components. One is obtained as the composition of a set of tasks with a

shared resource accessed in mutual exclusion. The other is obtained as the composition of the same set of tasks by applying a policy meeting given scheduling constraints. Is it possible to obtain a single component integrating this set of tasks and such that both mutual exclusion and the scheduling constraints hold? This kind of problem is non trivial. It is faced by system designers every day. They know solutions to specific problems and they need theory for combining them without jeopardizing their essential properties. Feature interaction in telecommunication systems, interference among web services, interference in aspect programming are all manifestations of lack of composability.

We present a rigorous system design flow and supporting tools meeting the above requirements. It is based on the BIP (Behavior, Interaction, Priority) component framework [6]. It uses BIP as a single unifying semantic model to ensure consistency between design steps. It extensively uses compositionality and composability to ensure correctness by avoiding complexity of monolithic verification. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction that is they preserve observational equivalence and consequently safety properties. Functional verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom. To avoid inherent complexity limitations, the verification method applies compositionality techniques implemented in the D-Finder tool [7].

BIP models are described in the BIP language, which allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described as a Petri net extended with data and functions described in C. The transitions of the Petri nets are labelled with guards (conditions on the state of a component and its environment) as well as functions that describe computations on local data. The description of coordination between components is layered. The first layer describes interactions between components. The second layer describes dynamic priorities between the interactions of the layer underneath and is used to express scheduling policies. Interactions are specified by using hierarchically structured connectors of two types: 1) connectors which describe synchronization by rendezvous of the interacting components; 2) connectors which describe broadcast of a message to all the enabled interacting components.

The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [8]. BIP has clean operational semantics that describe the behaviour of a composite component as the composition of the behavior of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

BIP can model in a natural and direct manner various types of synchronization. Using less expressive frameworks e.g. based on a single composition operator, often leads to intractable models. For instance, BIP directly encompasses multiparty interaction between components. Modeling multiparty interaction in frameworks supporting only point-to-point interaction e.g. function call, requires the use of protocols. This leads to overly complex models with complicated coordination structure.

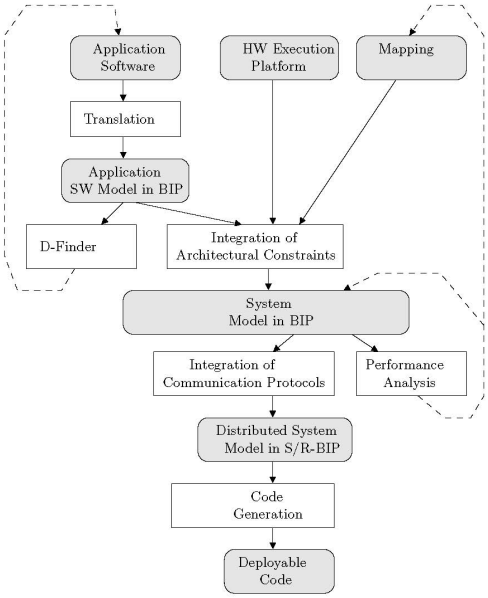

Figure 1

The design flow depicted in Figure 1, involves 4 distinct steps:

1. The translation of the application software into a BIP model. This allows its representation in a rigorous semantic framework. There exist translations of several programming models into BIP including synchronous, data-flow and event driven models [9,10].

2. The generation of a system model in BIP from 1) the BIP model representing the application software; 2) a model of the target hardware platform; 3) a mapping of the atomic components of the application software model into processing elements of the platform. The generated model takes into account hardware architecture constraints and execution times of atomic actions. Architecture constraints include mutual exclusion induced from sharing physical resources such as buses, memories and processors as well

as scheduling policies seeking optimal use of these resources.

3. The generation of a distributed system model obtained from the previous model by expressing high level coordination mechanisms e.g., interactions and priorities, in terms of primitives of the execution platform. This transformation consists in replacing atomic multiparty interactions by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring overall coherency e.g. non interference of protocols implementing different interactions.

4. The generation of monolithic C/C++ or MPI code from sets of interacting components executed on the same processor. This allows efficient implementation by avoiding overhead due to coordination between components.

The design flow is entirely supported by a toolset, which includes translators from various programming models, verification tools, source-to-source transformers and C/C++-code generators for BIP models (http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html)

References

[1] Joseph Sifakis. A Framework for Component-based Construction, 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM05), Keynote talk, September 7-9, 2005, Koblenz, pp. 293-300.

[2] N. Halbwachs. Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, 1993.

[3] A. Burns and A. Welling. Real-Time Systems and Programming Languages. Addison-Wesley, 2001. 3rd edition.

[4] T.A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design Computer, October 2007, pp. 32-40.

[5] R. Milner. A Calculus of Communication Systems, volume 92 of LNCS. Springer, 1980.

[6] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In Software Engineering and Formal Methods SEFM'06 Proceedings, pages 3-12. IEEE Computer Society Press, 2006.

[7] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In Computer Aided Verification CAV'09 Proceedings, volume 5643 of LNCS. Springer, 2009.

[8] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In Concurrency Theory CONCUR'08 Proceedings, volume 5201 of LNCS, pages 508-522. Springer, 2008.

[9] Marius Bozga, Vassiliki Sfyrla, Joseph Sifakis: Modeling synchronous systems in BIP. EMSOFT 2009: pages 77-86.

[10] A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis. Using BIP for Modeling and Verification of Networked Systems - A Case Study on TinyOS-based Networks. In Network Computing and Applications NCA'07 Proceedings, pages 257-260. IEEE, 2007.