Rigorous Component-based System Design

Saddek Bensalem[1,2], Ananda Basu[1], Marius Bozga[1], Paraskevas Bourgos[1], and Joseph Sifakis[1]

[1]VERIMAG Laboratory, Université Joseph Fourier Grenoble, CNRS
[2]CEA-Leti, MINATEC Campus, Grenoble France

Abstract. Rigorous system design requires the use of a single powerful component framework allowing the representation of the designed system at different levels of detail, from application software to its implementation. This is essential for ensuring the overall coherency and correctness. The paper introduces a rigorous design flow based on the BIP (Behavior, Interaction, Priority) component framework [1]. This design flow relies on several, tool-supported, source-to-source transformations allowing to progressively and correctly transform high level application software towards efficient implementations for specific platforms.

1 Introduction

Traditional engineering disciplines such as civil or mechanical engineering are based on solid theory for building artefacts with predictable behaviour over their life-time. These follow laws established by simple Newtonian physics and recognized building codes and regulations. Their complexity is limited by these physical and normative factors.

In contrast, for systems engineering, we do not have an equivalent theoretical basis allowing to infer system properties from the properties of its components. Computer science provides only partial answers to particular system design problems. With few exceptions, in this domain predictability is impossible to guarantee at design time and therefore, a posteriori validation remains the only means for ensuring their correct operation over time.

The complexity of systems currently being built, the fast pace of technological innovation, and the harsh market conditions to which they are subjected, including in particular time-to-market, create many difficulties for system design. These difficulties can be traced in large part to our inability to predict the behaviour of an application s oftware running on a given platform. Usually, such systems are built by reusing and assembling components: simpler sub-systems. his is the only way to master the complexity and to ensure the correctness of the overall design, while maintaining or increasing productivity. However, system-level integration becomes extremely hard because components are usually highly heterogeneous: and have different characteristics, are often developed using different technologies, and highlight different features from different viewpoints. Other difficulties stem from current design approaches, often empirical and based on expertise and experience of design teams. Naturally, designers attempt to solve new problems

by reusing, extending and improving past solutions proven to be efficient and robust. This favors component reuse and avoids re-inventing and re-discovering design solutions every time. Nevertheless, on a l onger term perspective, it may also be counter-productive: people are not always able to adapt in a satisfactory manner to new requirements and moreover, they tend to reject better solutions simply because they do not fit their design know-how.

2 System design

System design is facing several difficulties, mainly due to our inability to predict the behavior of an application software running on a given platform. Usually, systems are built by reusing and assembling components that are, simpler subsystems. 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, often empirical and 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 and re-discovering designs. Nevertheless, on a longer term perspective, this may also be counter-productive: designers are not always able to adapt in a satisfactory manner to new requirements. Moreover, they a priori exclude better solutions simply because they do not fit their know-how.

System design is the process leading to a mixed software/hardware system meeting given specifications. 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 coordination of the computation and interaction with the external environment.

System design radically differs from pure software design in that it should take into account not only functional but also extra-functional specifications regarding the use of resources of the execution platform such as time, memory and energy. Meeting extra-functional specifications is essential for the design of embedded systems. It requires evaluation of the impact of design choices on the overall behavior of the system. It 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.

A system design flow consists of steps starting from specifications and leading to an implementation on a given execution platform. It involves the use of methods and tools for progressively deriving the implementation by making adequate design choices.

We consider that a system design flow must meet the following essential requirements:

– *Correctness*: This means that the designed system meets its specifications. 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 its implementation. Correctness can be achieved by application of verification techniques. It is desirable that if some specifications are met at some step of the design flow, they are preserved in all the subsequent steps.
– *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

– *Performance*: The design flow must allow the satisfaction of extra-functional properties regarding optimal resource management. This means that resources such as memory, time and energy are first class concepts encompassed by formal models. Moreover, it should be possible to analyze and evaluate efficiency in using resources as early as possible along the design flow. Unfortunately, most of the widely used modeling formalisms offer only syntactic sugar for expressing timing constraints and scheduling policies. Lack of adequate semantic models does not allow consistency checking for timing requirements, or meaningful composition of features.
– *Parcimony*: The design flow should not enforce any particular programming or execution model. Very often system designers privilege specific programming models or implementation principles that a priori exclude 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. It is essential that designers use adequate programming models. Furthermore, design choices should be driven only by system specifications to obtain the best possible implementation.

3 Rigorous design flow

We call *rigorous* a design flow which allows guaranteeing essential properties of the specifications. Most of the rigorous 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].

A rigorous design flow should be characterized by the following:

– It should be *model-based*, that is all the software and system descriptions used along the design flow should be based on a single semantic model. This is essential for maintaining the overall coherency of the flow 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 encompasses 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. This makes possible the definition of a clear correspondence between the description of an untimed platform-independent behavior and the corresponding timed and platform-dependent implementation.

– It should be *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) modelling 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.

– It should rely on tractable theory for guaranteeing *correctness by construction* to avoid as much as possible monolithic a posteriori verification. Such a theory is based on two types of rules:

- 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.
- Composability rules ensuring that essential properties of a component are preserved when it is used to build composite components.

4 The BIP Design Flow

BIP [1] (Behavior, Interaction, Priority) is a general framework encompassing rigorous design. It uses the BIP language and an associated toolset supporting the design flow. The BIP language is a notation which allows building complex systems by coordinating the behaviour 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 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 the interactions between components. The second layer describes dynamic priorities between the interactions and is used to express scheduling policies. 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 [6]. BIP has clean operational semantics that describe the behaviour of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

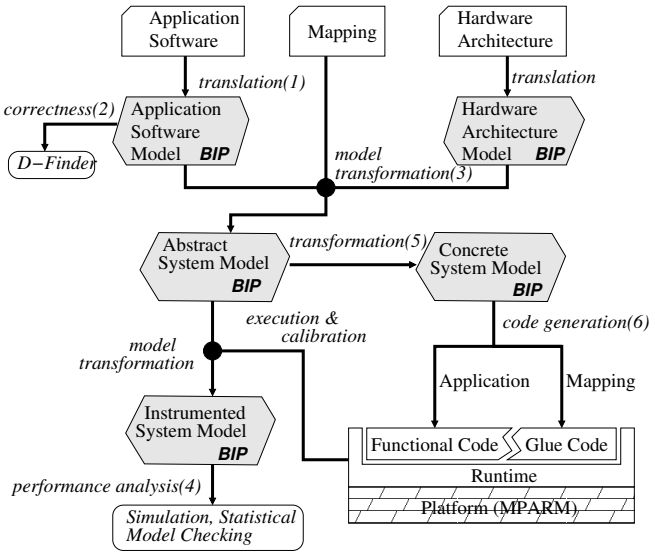

Fig. 1. BIP Design Flow for Manycore

The BIP design flow uses a single language to ensure consistency between the different design steps. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction, that means, they preserve observational equivalence and

consequently essential safety properties. The design flow involves several distinct steps, as illustrated in figure 1:

1. The translation of the application software into a BIP model. This allows its representation in a rigorous semantic framework. Translations for several programming models (including synchronous, data-flow and event-driven) into BIP are already implemented.
2. Correctness checking of the functional aspects of the application software. Functional verification needs to be done only at high level models since safety properties and deadlock-freedom are preserved by different transformations applied along the design flow. To avoid inherent complexity limitations, the verification method rely on compositionality and incremental techniques.
3. The generation of an abstract system model from the BIP model representing the application software, a model of the target execution platform as well as a mapping of the atomic components of the application software model into processing elements of the platform. The obtained 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.
4. Performance analysis on the system model using simulation-based models combined with statistical model checking.
5. The generation of a concrete system model obtained from the abstract model by expressing high level coordination mechanisms e.g., interactions and priorities by using primitives of the execution platform. This transformation involves the replacement of atomic multiparty interactions and/or dynamic priorities by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring. These transformations are proved correct-by-construction as well.
6. The generation of platform dependent code, inluding both functional and glue code needed to deploy and run the application on the target multi-core. In particular, components mapped on the same core can be statically composed thus avoiding extra overhead for (local) coordination at runtime.

References

1. 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.
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. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods FM'06 Proceedings*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.
5. R. Milner. *A Calculus of Communication Systems*, volume 92 of *LNCS*. Springer, 1980.
6. S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *CONCUR'08*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.