Chapter 3
Design Domains and Abstraction Levels for Effective Smart System Simulation

Sara Vinco, Michele Lora, Valerio Guarnieri, Jan Vanhese, Dimitrios Trachanis, and Franco Fummi

3.1 Introduction

Smart systems represent a broad class of systems defined as intelligent, miniaturized devices incorporating functionality like sensing, actuation, and control. In order to support these functions, they must include sophisticated and heterogeneous components and subsystems such as: application-specific sensors and actuators, multiple power sources and storage devices, intelligence in the form of power management, baseband computation, digital signal processing, power actuators, and subsystems for various types of wireless connectivity (as shown in Fig. 3.1).

Smart components and subsystems are developed and produced with very different technologies and materials specific to the corresponding domain and technology. The heterogeneity involves not only the language or framework adopted, but also different levels of abstraction and different communication and synchronization

S. Vinco
Politecnico di Torino, Torino, Italy
e-mail: sara.vinco@polito.it

M. Lora
University of Verona, Verona, Italy
e-mail: michele.lora@univr.it

V. Guarnieri
EDALab s.r.l., Verona, Italy
e-mail: valerio.guarnieri@edalab.it

J. Vanhese • D. Trachanis
Keysight Technologies, Rotselaar, Belgium
e-mail: jan_vanhese@keysight.com; dimitrios.trachanis@keysight.com

F. Fummi (✉)
University of Verona and EDALab s.r.l., Verona, Italy
e-mail: franco.fummi@univr.it; franco.fummi@edalab.it

N. Bombieri et al. (eds.), *Smart Systems Integration and Simulation*,
DOI 10.1007/978-3-319-27392-1_3

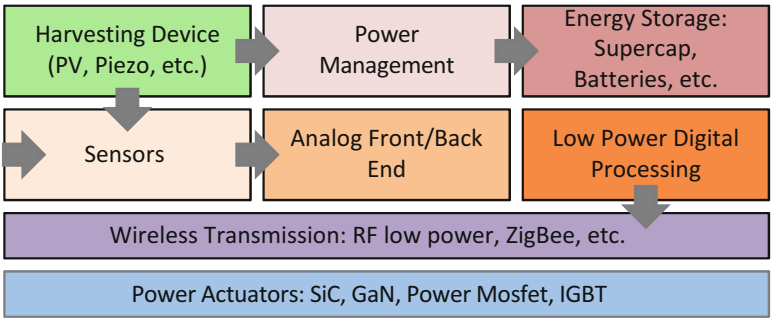

Fig. 3.1 Typical components of a smart system

styles. As of today, no design methodology and tools exist that can master, simultaneously and in a seamless manner, all the challenges that designers of smart micro-systems are confronted with when new products need to be developed. Nevertheless, modeling and design capabilities for heterogeneous components and subsystems are today available at specialized design houses and silicon makers in various forms. On the other hand, system integrators typically have separate tooling to model the environment. As a result, the challenge in the realization of a smart system goes beyond the design of the individual components and subsystems (an already difficult task by itself), but rather consists in accommodating a multitude of functionality, technologies, and materials.

In this context, simulation is a very critical task, as each component domain adopts specific tools and frameworks, that do not cover the whole smart system heterogeneity. On the other hand, simulation is a key phase in the design and verification process of a system, as it heavily impacts time-to-market and the competitiveness of the final product.

The goal of this chapter is to ease simulation and validation of smart systems. In this context, it provides a *taxonomy of abstraction level/design domains*, to highlight challenges and tools available for each domain. This allows to identify a precise role in the design flow for co-simulation and simulation scenarios, and thus to outline the possible strategies for gaining correct simulation of smart system. The two complementary approaches are compared with the goal of showing respective strengths and weaknesses.

The chapter also aims at enhancing *reuse and integration* by showing how state-of-the-art and commercial tools can ease the adoption of homogeneous simulation, with automatic code generation from lower abstraction levels and automatic integration of heterogeneous interfaces.

As a result, the chapter builds a *comprehensive modeling and simulation framework* that supports digital, analogue, and circuit-level descriptions simultaneously. This improves the contemporary smart systems design flow in such a way that a system-level simulation of all the heterogeneous components/subsystems of a smart

system will be possible. This advances state-of-the-art approaches by supporting the development of smart systems, their integration, and efficient simulation.

The chapter is organized as follows. Section 3.2 provides a background on smart system design, by listing available frameworks and formalisms, together with state-of-the-art tools. Section 3.3 identifies the typical abstraction levels and design domains involved in smart system design, with the goal of defining a taxonomy of the most widespread tools and languages. Finally, Sect. 3.4 proposes code conversion and generation approaches to gain homogeneous simulation of the heterogeneous components of a smart system, by working on both language and formalisms. Section 3.5 provides experimental evidence of the proposed solutions, and Sect. 3.6 concludes the chapter with some concluding remarks.

3.2 Background on Smart System Modeling

Smart components (and sub-components) are developed and produced with very different technologies and materials, specific to the corresponding domain. The goal of this section is to provide the necessary background for the proposed methodologies. Section 3.2.1 outlines the most widespread formalisms and frameworks available in the literature for tackling smart systems heterogeneity, while Sect. 3.2.2 deepens the ones adopted in the proposed flow.

3.2.1 Formalisms and Frameworks for Smart System Modeling

The heterogeneity of smart system involves not only the language or framework adopted, but also different levels of abstraction and communication and synchronization styles. An evidence of this are the adopted *description languages*, that only target specific domains, such as digital and software components (SpecC and SystemC) or analogue components (VHDL-AMS, Verilog-AMS, and SystemC-AMS).

In the literature, the main approaches proposed for handling such a heterogeneity are (1) top–down flows, relying either on model-based design (MBD) or on models of computation (MoCs), and (2) co-simulation, which exploits different simulation environments to take care of the heterogeneity of the system [13].

In *MBD approaches*, the system model is at the center of the design process and it is continually refined throughout a strictly top–down development flow [5, 22, 26]. Components following different synchronization mechanisms are put together through data conversion, that must be implemented manually, thus not guaranteeing correct integration.

Several *MoCs* have been proposed to describe different aspects of smart systems. As an example, extended finite state machines (EFSMs) [18] are an enhancement of traditional FSMs suited for describing digital HW components and cycle-accurate protocols, while hybrid automata have been defined to allow the integration of

continuous physical dynamics with discrete behaviors [20]. Unfortunately, every MoC is a stand-alone environment that cannot cover all the domains comprised in smart system development. Forcing communication through manual conversions between MoCs does not provide any guarantee of correctness of the final result.

The complementary approach is to integrate existing components in a bottom–up flow. This is realized with *co-simulation* environments where each component is simulated in its native environment and framework. Different simulators are then connected by defining rules and conversions about time management and event ordering, supported methods of communication, and rules of process activation [10, 17]. However, co-simulation assemblies heterogeneous components without providing a rigorous formal support, and it only moves the problem of integrating heterogeneous components to the problem of integrating different simulators.

3.2.2 Adopted Platforms for Smart System Design

This section gives a very brief overview of the main tools and platforms used in this work for smart system modeling and integration: SystemC and SystemC-AMS as a language supporting a number of abstraction levels (Sect. 3.2.2.1), HIFSuite for automatic conversion of reused code and components (Sect. 3.2.2.2), SystemVue as a simulator and co-simulator (Sect. 3.2.2.3), and UNIVERCM as a MoC spanning across various heterogeneous domains (Sect. 3.2.2.4).

3.2.2.1 SystemC and SystemC-AMS

SystemC is a widely deployed extension to C/C++ for describing HW constructs, ranging from register-transfer level up to transactional level [1]. The underlying simulation kernel is entirely event-based, i.e., a centralized scheduler controls the execution of processes based on events (synchronization, time notifications, or signal value changes). SystemC provides also a methodology for performing abstract modeling, simulation through generalized modeling of communication and synchronization: transaction level modeling (TLM) [3].

The SystemC simulation kernel has not been natively designed to handle the modeling and simulation of analog/continuous time systems. The recent extension SystemC-AMS [2] was designed for overcoming this lack, i.e., for modeling and simulating interacting analog/mixed signal functional subsystems, thus allowing to extend the adoption of a SystemC-based environment also to extra-functional, continuous time domains.

SystemC-AMS provides different abstraction levels to cover a wide variety of domains. *Timed data-flow* (TDF) models discrete time processes, that are scheduled statically by considering their producer–consumer dependencies. *Linear signal flow* (LSF) supports the modeling of continuous time behaviors through a library of predefined primitive modules (e.g., integration, or delay), each associated with

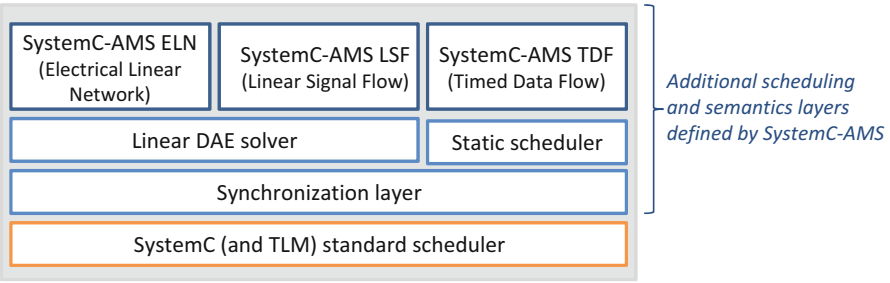

Fig. 3.2 Architecture of the SystemC scheduler as extended with SystemC-AMS support

a linear equation. *Electrical linear network* (ELN) models electrical networks through the instantiation of predefined primitives, e.g., resistors or capacitors, where each primitive is associated with an electrical equation. In case of ELN or LSF descriptions, a SystemC-AMS AD solver analyzes the ELN and LSF components to derive the equations modeling system behavior, that are solved to determine system state at any simulation time.

As highlighted in Fig. 3.2, the key feature of all SystemC extensions is that overall simulation is handled by the sole SystemC simulation kernel, that interacts with its extensions to define, time after time, both the execution queue and the corresponding system evolution.

3.2.2.2 HIFSuite

HIFSuite is a set of tools and application programming interfaces (APIs) that provide support for modeling and verification of HW/SW systems [15]. The core of HIFSuite is the HDL Intermediate Format (HIF) language upon which a set of front-end and back-end tools have been developed to allow the conversion of HDL code into HIF code and vice versa. HIFSuite allows designers to manipulate and integrate heterogeneous components implemented by using different hardware description languages (HDLs). Moreover, HIFSuite includes tools, which rely on HIF APIs, for manipulating HIF descriptions in order to support code abstraction/refinement and post-refinement verification, including A^2T, a tool for abstracting RTL digital components to TLM or C++ [8].

3.2.2.3 SystemVue

SystemVue is an electronic design automation (EDA) environment for electronic system-level (ESL) design, focused on RF and DSP systems [4]. It supports complex RF envelope carriers and dataflow simulations [21]. In SystemVue, a system is described as a schematic of components connected with wires and busses. The

simulation technology is based on a Data-Flow MoC and it is based on the Ptolemy multi-domain, heterogeneous simulation platform [22].

SystemVue is well suited for the integration of heterogeneous systems. It provides numerous libraries with parameterized components and interfaces to diverse modeling formats, ranging from MATLAB to the main HDLs, such as Verilog and VHDL. Furthermore, it allows to create custom components in math language or C++ and to add them to a purely SystemVue system. SystemVue supports multi-domain simulations through links to event-based as well as circuit simulation engines, such as SystemC and ModelSim, may be extended to analogue simulations.

3.2.2.4 UNIVERCM

UNIVERCM is an automaton-based formalism that unifies the modeling of both the analogue (i.e., continuous) and the digital (i.e., discrete) domains, as well as hardware-dependent SW. A formal and complete definition is available in [19].

In each UNIVERCM automaton (depicted in Fig. 3.3), states model the continuous dynamics of the system as a condition that must be satisfied to perform continuous evolution (invariant) and a predicate modeling the evolution of variables over time (flow). Edges between states model the discrete dynamics as evolution of variables and activation of synchronization events, controlled by a boolean predicate on the variable state and by synchronization checks.

UNIVERCM is an important resource in smart system design as it is well suited for the application to heterogeneous domains [19]. Indeed, the computational model allows to cover the heterogeneity that characterizes such systems, ranging from analogue and digital HW up to dedicated SW. Guglielmo et al. [19] presented a comprehensive reuse and design flow based on UNIVERCM, thus showing how it is possible to provide formal rules and automatic tools to convert the heterogeneity to UNIVERCM and to produce a homogeneous simulatable implementation of the generated UNIVERCM system. Thus, UNIVERCM enhances reuse and bottom–up design.

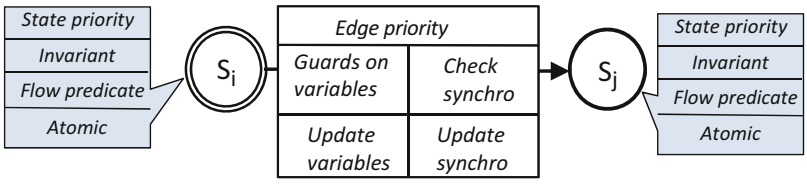

Fig. 3.3 Example of UNIVERCM automaton

3.3 Analysis of Smart System Simulation Solutions

Simulation and design are heavily influenced by the abstraction level of each component and, as a consequence, by the level of heterogeneity that characterizes the system in terms of domains, abstraction levels, and synchronization mechanisms. It is thus necessary to clearly identify the abstraction level involved in smart system design (Sect. 3.3.2) and to associate each domain and simulator to the correct level. For this reason, this section proposes a taxonomy that associates frameworks and design flows to each domain and abstraction level (Sect. 3.3.3). This constitutes a necessary starting point for understanding the impact of abstraction levels and of the heterogeneity/homogeneity trade-off on simulation (Sect. 3.3.4).

3.3.1 Typical Domains of Smart System Design

The typical classes of components of any smart system are identified in terms both of constituting characteristics and of role *w.r.t.* the inner information/energy flows. For this reason, components are sub-divided into six main domains:

- *MEMS, sensors, and actuators*, in charge of communicating with the surrounding environment;
- *Power sources*, necessary to guarantee correct functioning of all other components;
- *Discrete and power devices*, as parts of the energy flow, responsible for energy dispatching and harvesting;
- *Analogue and RF components*, mainly responsible for signal processing, transmission, and reception;
- *Digital HW*, core of the system processing and functionality;
- *Embedded SW*, as system controller and main mean of communication with the end users.

The main simulation problems of smart systems derive from this heterogeneity that requires the use of different design languages and different abstraction levels. Moreover, it is extremely unlikely that a single team has the knowledge to cover all such design domains, thus, we have to assume that a set of design teams must cooperate by using their own favorite design languages. In fact, there is no *Esperanto* able to effectively model all such domains. A variety of design languages has rather been proposed in the past decades to cover specific design domains, and some standards de facto became the reference languages for design teams specialized in each design domain. This challenging scenario will be the focus of the next sections.

3.3.2 Abstraction Levels of Smart System Design

The main factors determining the level of abstraction are: time granularity, interconnection model, state space granularity, and data aggregation. *Time granularity* is an important dimension in a heterogeneous environment. It may be continuous or discrete time, or follow an event-based semantics where time ticks only when the system state changes. The *interconnection model* describes communication and synchronization between components as potential or flow quantities (conservative systems), flow charts, or transactions. The *granularity of state space* details data aggregation for simulation purposes, i.e., variables managed by differential equations, symbolic variables, or objective constructs (i.e., system state describes the possible behavior, e.g., C++). Finally, *data aggregation* states whether the component is modeled by considering the minimum (black box) or maximum (clear box) number of state space variables necessary for a correct representation of the observable behavior.

Given these factors, it is possible to identify five main abstraction levels, typical of smart systems.

- At *transactional* level, simulation is strictly event-based and inter-component communication happens via transactions (that provide a communication protocol to the system). System state is modeled with variables.
- At *functional* level, simulation is event-based but communication relies on the flow chart interconnection style.
- The *structural* level has two main approaches depending on time granularity. Continuous time evolution is modeled with differential equations and by observing conservative laws. Discrete time may adopt both event-based or flow chart synchronization, and finite set variables are adopted.
- At *device* level, simulation can be both continuous or discrete time. The major difference is that at device level all variables are modeled explicitly, while structural level models only those variables that are strictly necessary for simulation purposes.
- The *physical* level adopts continuous time synchronization and the conservative interconnection style. State space is described with continuous fields as differential equations and all variables are modeled in a clear box approach.

3.3.3 Design-Domains/Simulation-Level Taxonomy

Given the variety of abstraction levels and the heterogeneous domains typically present in any smart system, it is possible to build the design-domains/simulation-level taxonomy shown in Fig. 3.4. Such a chart identifies the abstraction level (rows) and the domain (column) of the most widespread tool and languages adopted in the context of smart systems. This allows to correctly differentiate the use of co-simulation and simulation according to the two dimensions. Text in bold shows the typical entrance level and tools for each domain.

	MEMS, sensors and actuators	*Power sources*	*Discrete and power devices*	*Analog and RF*	*Digital HW*	*Embedded SW*	
TRANSACTIONAL	SystemVue	SystemVue	SystemVue	SystemVue	SystemVue, SystemC (TLM/AMS)	SystemVue, QEMU	SIMULATION
FUNCTIONAL	C++	C++	C++	C++, SystemVue	C++, SystemC	**C++, QEMU**	SIMULATION
STRUCTURAL	ADS, Matlab, AMS HDLs, MEMS+	Matlab, Simulink, AMS HDLs	ADS	ADS, Matlab, AMS HDLs	**RTL HDL**	Cycle accurate QEMU	CO-SIMULATION
DEVICE	Matlab, MEMS+, AMS HDLs	FEM, Spice	EMPro, Spectre, Momentum	EMPro, Spectre, Momentum	AMS HDLs	–	CO-SIMULATION
PHYSICAL	**Matlab, MEMS+, FEM**	**FEM, Spice**	**EMPro, Spectre, Momentum**	**EMPro, Spectre, Momentum**	AMS HDLs	–	CO-SIMULATION

Fig. 3.4 Design-domains/simulation-level taxonomy, identifying the abstraction level (*rows*) and the domain (*column*) of the most widespread tool and languages adopted in the context of smart systems. *Text in bold* shows the typical entrance level and tools for each domain

Models belonging the *lowest abstraction levels* (i.e., physical, device, and structural) are represented by different domain-specific design languages. They must thus be simulated by using their own simulator (e.g., Matlab, Modelsim, EMPro). For this reason, a framework covering more than one domain can be implemented only by using *co-simulation techniques* which connect different tools by exchanging simulation data from one tool to another.

Moving to the *functional level*, there is a convergence in the modeling language, as all models belonging to different domains are represented in C++. This would in principle allow a simulation among different domains. However, the MoC implemented into each C++ model can be different from domain to domain. Thus, simulation cannot be simply obtained by linking functional C++ models, but such models must also be coherent *w.r.t.* the same MoC. Thus, either the chosen MoC covers all domains or some data and synchronization conversion is necessary.

At *transaction level*, simulation frameworks enforce a common transaction-based communication protocol to all domains. This allows to seamlessly integrate components belonging to different domains and based on different MoCs and synchronization mechanisms.

3.3.4 Impact of MoCs on Simulation and Co-simulation Performance

The taxonomy in Fig. 3.4 helps in further understanding the impact of MoCs and of heterogeneity on simulation and co-simulation at different abstraction levels.

As mentioned in Sect. 3.3, the heterogeneity of the *lowest abstraction levels* forces to simulate each design domain by using ad-hoc simulators. Co-simulation frameworks are thus built by connecting different simulators, such as shown in [10, 17]. Unfortunately, explicitly modeling the synchronization between

simulators, different for language, formalism, and underlying MoC, heavily impacts simulation performance and effectiveness [19]. Other approaches achieve a lighter impact by compiling separately the different formats and linking them together, such as done by ModelSim to co-simulate SystemC and VHDL. This lighter approach is still affected by the presence of heterogeneous MoCs, as the data sharing mechanism and time synchronization introduce a heavy overhead.

Functional level brings to a convergence in terms of modeling language and framework, thus showing the impact of MoCs to the full. If all C++ components follow the same MoC, then they can easily be integrated with no further overhead. Else, if the adopted MoCs are heterogeneous, it becomes necessary to introduce a communication layer for applying data and synchronization conversion.

Communication and synchronization are further eased at *transaction level*, as transactions and standard interfaces force a single communication protocol to all components. This mitigates the effect of having multiple MoCs, as problems risen by data sharing and time synchronization are moved inside the transactional communication mechanism.

This analysis highlights that the heterogeneity of smart systems impacts simulation performance in many directions. Contributing elements are indeed the adopted languages, the levels of abstraction, and the MoCs followed by the components to be integrated. The weakest approach appears to be co-simulation, mandatory at lowest levels, as it pays the price of all degrees of heterogeneity. Simulation becomes more effective at functional and transactional levels, where heterogeneity is constrained and limited to few synchronization mechanisms. For these reasons, the remainder of this chapter will focus on code generation for effective simulation of smart systems at functional and transactional levels.

3.4 Proposed Methodologies

The analysis of the smart system simulation scenarios proposed in the previous section highlighted that the choices in terms of abstraction level, language, and MoC may heavily affect simulation performance. This section outlines three alternatives, different in terms of implementation choices and covered domains. The solutions are summarized in Fig. 3.5, and they provide different coverage/performance trade-offs, together with techniques and tools for achieving automatic generation of simulatable code. Section 3.4.1 focuses on functional level, and it estimates the impact of MoCs on simulation. On the other hand, Sect. 3.4.2 provides two solutions at transactional level, based on SystemC and on the SystemVue framework.

3.4.1 Smart System Simulation at Functional Level

The functional level brings all domains to a convergence in terms of modeling language, usually C++. This easies the achievement of simultaneous simulation of components belonging to different domains. At the same time, an effort may

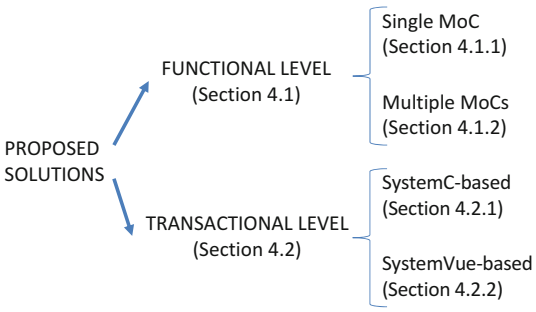

Fig. 3.5 Proposed solutions for homogeneous simulation of heterogeneous smart systems

be necessary whenever the C++ representations of components follow different MoCs, i.e., different synchronization management rules. This section provides an example for both flows, with the goal of showing the impact of MoCs to the full.

3.4.1.1 Simulation Based on a Single MoC

The UNIVERCM MoC, presented in Sect. 3.2.2.4, was designed to reconcile heterogeneous domains to a unique formalism. It supports a full bottom–up approach where already existing heterogeneous descriptions can be automatically converted and integrated into UNIVERCM automata for being, subsequently, re-mapped to a single simulatable model. This section details both the flows, with a focus on the major conversion issues and solutions.

Mapping from Heterogeneity to UNIVERCM

The strategy to map any component to UNIVERCM strictly depends on the domain and abstraction level of the starting description [14].

Mapping *digital HW descriptions* in UNIVERCM requires to reproduce the simulation semantics of HDLs, both in terms of scheduling and of synchronization.

HDL processes are represented as automata. All edges of an automaton are guarded by the activation of synchronization labels, reproducing a value change of any of the signals in the sensitivity list. This activates an automaton in response to changes in its sensitivity list. Note that the propagation of synchronization events is straightforward, as labels are instantaneously visible from any automaton.

The typical HDL scheduling routine is in charge of generating and propagating events and advancing simulation time. This mechanism must be represented in UNIVERCM so that events are processed in the same order and simulation semantics is preserved. The main feature that must be preserved is thus the fact that simulation time is advanced only when there is no event to be processed in the system nor any signal to be updated. The scheduling routine is represented with an additional automaton, that advances a continuous variable representing time only when there is no active label in the system. This allows to process events in the same order as in the original HDL and to preserve the original simulation semantics.

HW-dependent SW (HdS) is SW that controls and abstracts HW functionality, to allow easy and standard access to HW devices and the deployment of more abstracted SW. HdS is thus in charge of managing communication with HW and it needs to be reactive to signals and interrupts risen by HW devices. Each HdS function is mapped to a UNIVERCM automaton, evolving among a certain set of states via transitions (note that continuous time evolution is not supported for this domain). Each function is provided with two special labels: an activation label (representing function invocation and activated by automata willing to execute the function) and a return label (used to communicate to the caller that the function has finished its execution). This allows inter-function communication. Automata representing HdS functions can be also sensitive to events coming from HW automata, representing HW interrupts. This, together with data sharing for modeling MMIO mechanisms, allows to reproduce the basics of HW–SW communication. An example of HW–SW communication, and of mapping to UNIVERCM of the corresponding components, is provided in Fig. 3.6.

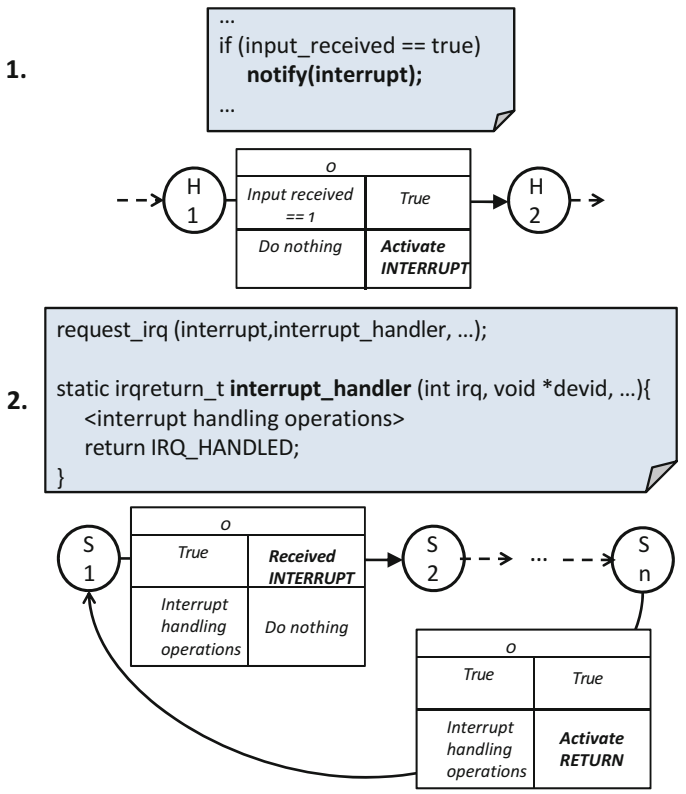

Fig. 3.6 Mapping to UNIVERCM of a digital HW component firing an interrupt (*1*) and of the corresponding interrupt service routine (*2*)

UNIVERCM can be easily adopted to model also *analogue models* described with differential equations as hybrid automata [16, 27]. The mapping is straightforward, even if some transformations are necessary to reproduce the synchronization semantics and to remove hierarchy from the automata.

Once that all starting descriptions have been converted to UNIVERCM, automata evolve simultaneously through data sharing (i.e., by accessing the same variables) and by synchronizing via labels. Thus, no additional communication or scheduling mechanism is necessary.

Mapping from UNIVERCM to C++

The conversion flow from UNIVERCM to C++ is defined in general for any automata, with no concern regarding the language of the original description converter to UNIVERCM.

Each UNIVERCM automaton is mapped to a C++ function, representing the whole automaton evolution, as depicted in Fig. 3.7. A state variable is used to store the current state of the automaton. The function body is built as a `switch` statement, where each case represents one of the automaton states. Each state case lists the implementation of all the outgoing edges and of the delay transition provided for the state.

Each *edge* is implemented as an `if` or `else if` statement, whose guard is a logic `and` of the enabling condition on the edge and of the activation condition on synchronization events. The body executed when the guard is satisfied includes the update of variables and the activation of synchronization events. Furthermore, the state variable is updated to the destination state of the edge.

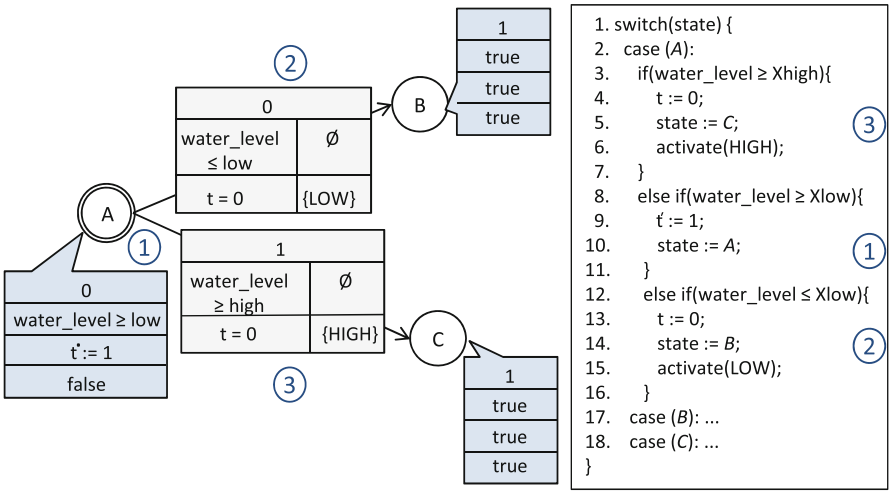

Fig. 3.7 UNIVERCM automaton to be converted to C++ (*left*) and corresponding generated code (*right*)

Continuous evolution is implemented as an `if` or `else if` statement whose guard is the invariant condition that allows to remain in the state. The body executed when the guard is true implements a discretized implementation of the flow predicate, by adopting the *Euler numerical integration algorithm* with time discretization step chosen by the designer [11]. It is important to note that the Euler method can be replaced with one of the many available algorithms for the approximation of solutions of ordinary differential equations.

Code generated from UNIVERCM automata is ruled by a *management function*, in charge of activating automata and of managing the status of the overall system and parallel composition of automata. The result of this approach is that all code generated from UNIVERCM automata is controlled by a single function, and it is thus provided with a simple interface.

If the system is made of more UNIVERCM automata, the management function is not enough to grant correct composition. Indeed, the starting components come from heterogeneous domains, and thus the communication means may differ. On the other hand, communication between UNIVERCM automata happens via variable sharing and through synchronization events. Thus, any two automata can be easily composed by checking the correspondence between variables and synchronization events of the two. Mapping the one in the other must be identified by the designer. This allows to extend the management function to all operations necessary to propagate updated values.

Finally, UNIVERCM variables and events are mapped to native C++ constructs. Variables are mapped to a couple of C++ variables, representing the current value and the future value, respectively, in order to respect the UNIVERCM semantics. Value update is performed by the management function, as previously anticipated. The type of each variable is determined by the variable alphabet for discrete variables, while continuous variables are mapped to doubles. Support type libraries may be used, for simulation purposes or to enhance simulation speed [9]. Synchronization events are represented with boolean values, where `true` states that the label is active. In detail, labels are mapped to a couple of boolean values, representing the current value and the next simulation value, respectively. At the end of each simulation step, the management function will set the new current value to the future one, and reset the future value to `false`.

Integration Strategies and Challenges

Simulation based on a single MoC poses no challenges regarding integration. All starting components, despite of their heterogeneity, are converted to UNIVERCM automata, by mapping the starting semantics to UNIVERCM native constructs. This allows to abstract the characteristics of the starting descriptions, and to represent the system as a number of automata that interact through no conversion mechanism. This is a winning approach, as no manual intervention is necessary to allow integration. This reduces by far communication overheads, and it speeds up simulation.

3.4.1.2 Simulation Based on Multiple MoCs

UNIVERCM is a very powerful MoC, as it covers a wide number of domains. However, its representation of digital HW may lead to an explosion of the modeled automata, both in terms of states and of synchronization labels. Furthermore, no methodology has been defined yet for mapping circuit-based descriptions, as electrical behaviors and conservation laws are difficult to reproduce in an automata based approach. For this reason, it may be necessary to integrate code generated via UNIVERCM with C++ code generated with other strategies. This section outlines two additional strategies, necessary to cover all smart system domains efficiently. The section ends by presenting the integration strategies and challenges, to allow overall smart system simulation even in presence of different MoCs.

HIFSuite for Efficient Conversion of Digital HW to C++

HIFSuite (introduced in Sect. 3.2.2.2) is a closely integrated set of tools and APIs for reusing already developed components and for verifying their integration into new designs [15].

HIFSuite was first designed for allowing system designers to convert HW/SW design descriptions from a HDL to a different HDL and to manipulate them in a uniform and efficient way. For this reason, the underlying HIF core language is made of a set of objects corresponding to traditional HDL constructs like, for example, processes, variable/signal declarations, sequential and concurrent statements, and so forth [6]. Each HIF construct is mapped to a C++ class that describes specific properties and attributes of the corresponding HDL construct. Such objects can then be manipulated through powerful C++ APIs which allow to explore, manipulate, and extract information from HIF descriptions.

All such characteristics make HIFSuite a very convenient infrastructure to define conversion tools working on digital HW descriptions. The typical conversion flow from digital HW to C++ is outlined in Fig. 3.8, and it leaves the underlying MoC of the starting description unchanged.

Any digital HW description, implemented in a HDL language, is converted to its HIF representation via the *HIFSuite front-end tools*, performing a straightforward mapping from HDL constructs to the corresponding HIF objects. The abstraction of the HIF description is then carried out by two manipulation tools from HIFSuite, DDT and A^2T. DDT replaces the original HDL data types from the starting HW description with C++ built-in data types in order to greatly improve simulation performance. Then, A^2T implements the methodology in [7] to convert the HDL processes to functions and the HDL scheduling semantics to a management function. Additionally, A^2T can be guided to generate more performing C++ code by providing it with profiling information of the starting HDL implementation. If the repeated execution of asynchronous processes dominates execution time, A^2T may replace the standard dynamic HDL simulation semantics with a static scheduling approach. Such an approach creates a sequence of processes to be repeated at every

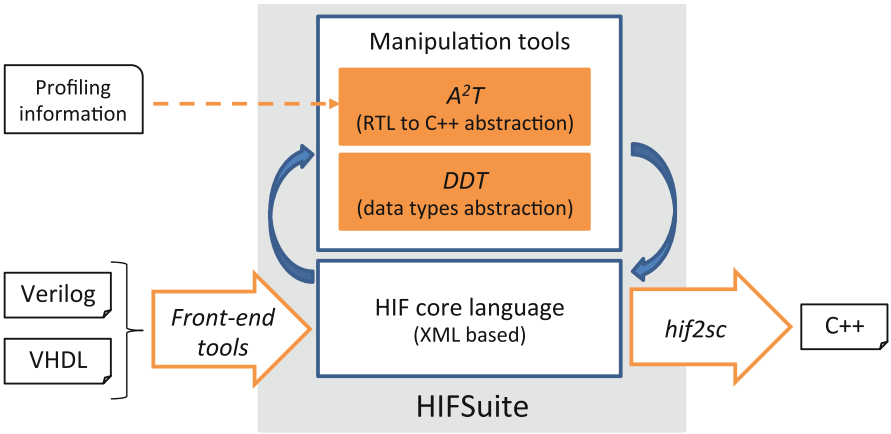

Fig. 3.8 HIFSuite-based flow for automatic conversion of digital HW descriptions to C++

simulation cycle, thus avoiding the overhead of event management. This allows to *further abstract the starting HDL description*, to customize the generated code with the goal of optimizing simulation performance. The obtained HIF description is finally converted to C++ through the back-end tool hif2sc.

The winning aspect of this strategy *w.r.t.* the UNIVERCM-based conversion flow presented in Sect. 3.4.1.1 lies in the efficiency of the generated code. HIF natively preserves the HDL semantics, thus not introducing additional constructs, e.g., for scheduling or synchronization management. This results in a more compact C++ implementation of the starting digital HW.

Conversion of Analogue and Mixed Signal Descriptions to C++

Analogue components can be seen as a set of algebraic and differential equations, expressing the functionality. These equations can be expressed in different ways: they can be explicitly listed or they can be hidden by expressing them as interconnections of primitives, as for block diagrams. Thus, when aiming at reproducing the behavior of an analogue device, it is fundamental to extract the correct set of equations from the original description. To accomplish this task, HIFSuite analysis features come in handy, and they are exploited into a framework of front-end, manipulation and back-end tools. The resulting flow is depicted in Fig. 3.9.

To read analogue descriptions, the Verilog parser of HIFSuite is extended to support Verilog-AMS. The tool takes care of parsing analogue descriptions, based on dipole equations, and to map constructs into HIF. The HIF representation is then used to analyze and manipulate the information expressed by the design. Analysis and manipulation are performed by OCCAM (*Ordinary C++ Code for Analogue Models*), a tool developed on top of HIFSuite that implements an analysis and manipulation algorithm composed by the following five steps:

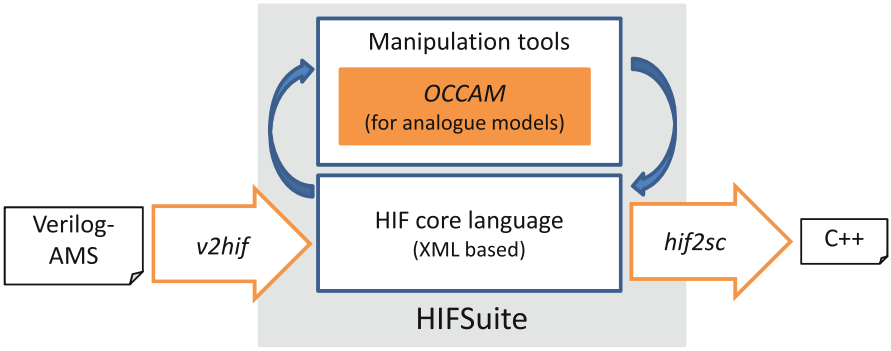

Fig. 3.9 HIFSuite-based flow for automatic conversion of analogue and mixed signal descriptions to C++

- *Acquisition*: Starting from the set of dipole equations acquired by the Verilog front-end tool, a hash table is created. For each electric branch of the circuit represented by the original description, current and voltage are labeled and then, every equation is stored in the hash table, using the left value label as key. Then, also the inverse equations are computed, stored in the table, and marked as "linearly linked" to the original equation.
- *Enrichment*: The system of equations can be partially specified, and some relations may thus be left implicit. It is necessary to apply Kirchhoff's current and voltage laws to retrieve the entire set of equations composing the system. This is done by employing a modified nodal analysis algorithm on the set of equations extracted during the acquisition step. The implicit equations, retrieved by the modified nodal analysis, are inserted into the hash table and marked as "linearly linked."
- *Assemble*: In order to abstract the system, the outputs of interest are fixed by the designer. For every output of interest, its label is used to fetch an equation from the hash table. Then, all the terms of the fetched equations are used as label to fetch other equations, recursively, in order to retrieve all the terms influencing the chosen output. A tree structure representing these dependencies is built for every output of interest.
- *Preparation*: The tree built at the assemble step is visited, and the dependencies are mapped into a sequence of assignments and function calls, to represent algebraic and differential operators.
- *Dismantle*: The sequence of instructions created after the previous steps are inserted into a function. Since the produced models aim at simulating continuous time evolution, they have to be repeatedly executed. Thus, the simulation scheduler will provide to call and execute the function wrapping the behavior, periodically during the simulation.

Finally, the behavioral representation produced by OCCAM and modeled in HIF has to be translated into C++. To do this, the HIFSuite back-end tools have been extended in order to support this kind of representation, to produce C++ code for the simulation.

Integration Strategies and Challenges

The integration of C++ code generated with the presented techniques introduces major challenges. Indeed, this section clearly highlighted that at functional level different domains and techniques share a common language, but not the MoC and the synchronization mechanisms. As an example, an event fired by a component generated through UNIVERCM may be difficult to detect by code generated through HIFSuite or through a complex abstraction process, necessary to handle analogue and mixed signal descriptions.

If execution inside components may be self-sufficient and correct, problems arise whenever interaction between components is necessary. Due to the complexity of the task and to the complex configurations that may show up, this task can be handled only manually, by carefully considering the characteristics of the specific components into play.

Whenever integrating heterogeneous C++ code, the designer shall consider:

- *Functionality activation*: Each MoC introduces different scheduling strategies in the C++ code, ranging from the reconstruction of HDL scheduling up to simple activation of all automata for UNIVERCM-based code. The designer shall implement a global scheduling routine, that activates the single domains by respecting timing and causality relationships;
- *Time evolution*: Each MoC advances time with specific solutions, that are affected by the presence of runnable activities. Local scheduling strategies must thus agree on a shared notion of time, so that events are propagated in the correct order and that digital synchronous signals such as clocks are coherent *w.r.t.* the remainder of the system;
- *event propagation*: Each local scheduler must be able to detect synchronization events fired by the other domains. For this reason, the global scheduler must convert events from one formalism to the other, without introducing delays or timing misalignments;
- *Data sharing*: Different components must be able to share data despite of the implementation differences. The global scheduling routine shall propagate value changes, thus converting data from one format (or data type) to the other.

This highlights that, even if the single conversion techniques are correct, interaction of heterogeneous MoC introduces heavy management overheads and it may leave space for synchronization misalignments.

3.4.2 Smart System Simulation at Transactional Level

The transactional layer brings all domains to a convergence in terms of modeling language and of underlying framework. The differences in terms of MoC or abstraction level are not reduced by means of conversion methodologies, but they are rather preserved to ease the integration process. Ad-hoc interfaces or simulation

strategies mask this heterogeneity with a transaction-based mechanism, where a global scheduler satisfies activation requests and performs all conversions and synchronization with no intervention from the user. This section provides two examples of this strategy, the one relying on the standard language SystemC (Sect. 3.4.2.1) and the other based on the commercial tool SystemVue (Sect. 3.4.2.2). This will highlight the characteristics of the transactional level to the full.

3.4.2.1 SystemC-Based Simulation

SystemC, together with its extensions, is a well-established language for the modeling of smart systems. Its strength, as anticipated in Sect. 3.2.2.1, is the presence of a single simulation kernel, mastering requests coming from any of the supported MoCs and libraries.

SystemC can be considered transactional as any of the supported MoCs defines a precise interface to the simulation kernel, thus wrapping different levels of abstraction of the instantiated constructs. Each solver communicates with the simulation kernel through transactions, i.e., activation requests that are satisfied by the kernel through synchronization with the remainder of the system and through data sharing and conversion. This section shows how effective SystemC can be at supporting the heterogeneity of smart systems, ranging from analogue and mixed signal conservative descriptions up to digital HW components.

Mapping from UNIVERCM to SystemC

Mapping of UNIVERCM to SystemC traces the approach for C++ code generation proposed in Sect. 3.4.1. However, the presence of a simulation kernel allows to delegate some management tasks, and to reproduce automata behavior through native SystemC constructs. Note that this is crucial to ease and enhance the interaction with SystemC code generated through different design flows.

The main effect of the adoption of SystemC is on the management routine. UNIVERCM automata are indeed mapped to processes, rather than functions. This allows to delegate automata activation to the SystemC scheduler, by making each process sensitive to its input variables. Automata activation is removed from the management function, that still updates the status of variables and events at any simulation cycle. The management function itself is declared as a process, activated with a custom event after all automata have performed one simulation step.

The mapping of synchronization events is left unchanged, despite of the presence of native SystemC events, i.e., `sc_events`. Indeed, SystemC events cannot be used into conditions, while this is a feature necessary to fully support UNIVERCM transition semantics.

The mapping of UNIVERCM variables changes slightly. Variables shared by two or more automata are mapped to SystemC signals, to allow data sharing between processes and ensure correct simulation and process activation. UNIVERCM

variables used by a single automata are still mapped to a couple of C++ variables, i.e., current value and future value, that are updated and handled by the management function.

Mapping of Digital HW to SystemC and SystemC TLM Through HIFSuite

As previously stated in Sects. 3.2.2.2 and 3.4.1.2, HIFSuite is an ideal framework to convert digital HW descriptions into corresponding SystemC and SystemC-TLM descriptions. The flow to automatically convert digital HW descriptions to SystemC at RTL is depicted in Fig. 3.10. The input HW description, written in VHDL or Verilog, is firstly converted to its HIF representation by the *HIFSuite front-end tools*. This step is achieved by parsing the input description and mapping HDL constructs to corresponding HIF objects. Then, the HIF description is converted to the corresponding SystemC RTL code by the back-end tool hif2sc. A number of manipulations on the HIF description are required during this step to account for the lack of expressiveness of SystemC *w.r.t.* VHDL and Verilog. In fact, some VHDL and Verilog constructs do not have a direct mapping to a corresponding SystemC construct. As such, they must be translated by resorting to an equivalent implementation through other SystemC constructs.

HIFSuite also features a flow to automatically abstract digital HW descriptions to SystemC TLM for faster simulation speed. The resulting flow is illustrated in Fig. 3.11. The first step consists again of converting the input HW description to its corresponding HIF representation by the *HIFSuite front-end tools*. If the target is to generate a TLM description optimized for simulation performance, the following step consists of invoking DDT from HIFSuite on the generated HIF description in order to improve simulation performance by replacing the original HDL data types with C++ built-in data types. This step is however completely optional. In case it is bypassed, the output TLM description at the end of the flow will feature SystemC data types. The abstraction of the HIF description from RTL to TLM is carried out by the manipulation of A^2T from HIFSuite. A^2T produces code compliant with the TLM-2.0 standard. The user can select which TLM protocol will be generated by adopting one of the two TLM-2.0 coding styles, namely *loosely timed* (LT) and *approximately timed* (AT). If the LT coding style is adopted, the

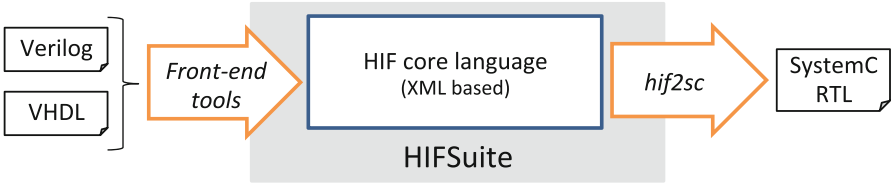

Fig. 3.10 HIFSuite-based flow for automatic conversion of digital HW descriptions to SystemC RTL

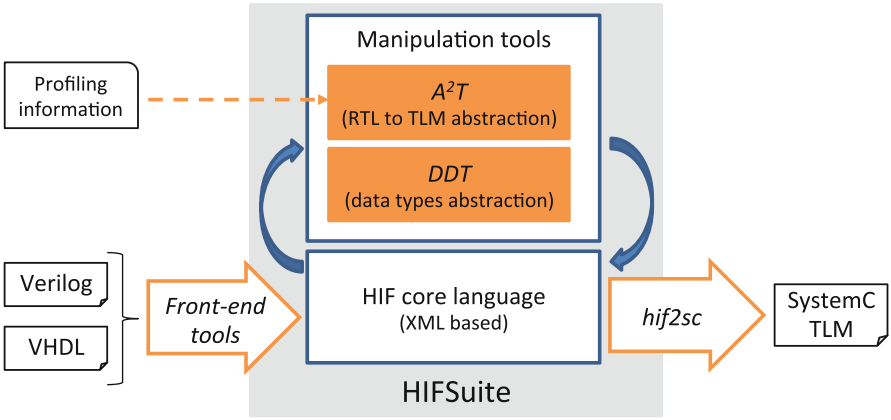

Fig. 3.11 HIFSuite-based flow for automatic conversion of digital HW descriptions to SystemC TLM

abstracted design will implement the blocking transport interface, and blocking transport primitives will be used to achieve communication. Conversely, if the AT coding style is adopted, the abstracted design will implement the non-blocking transport interface, and non-blocking transport primitives will be used to achieve communication. The abstraction process generates C++ functionality code from RTL processes, and replaces the RTL cycle-accurate communication protocol with the transaction-based TLM communication protocol. As reported in Sect. 3.4.1.2, profiling information on the starting HW description can be provided to A^{2}T in order to generate more efficient C++ code for the design functionality. Finally, the abstracted HIF description is converted to SystemC TLM through the back-end tool hif2sc.

Mapping of Analogue Conservative Descriptions to SystemC-AMS

Smart systems often feature heterogeneous components that do not match the traditional digital design flow. A typical example is MEMS components, often used as means of sensing and actuation, thus having a crucial role in the interaction of the system with the surrounding environment. The main complexity introduced by this kind of descriptions is that they are *both behavioral and conservative*, i.e., they feature a certain level of abstraction *w.r.t.* the actual component realization, but at the same time they obey physical laws, such as energy conservation laws [12, 25].

The limitations of traditional flows and tools at handling such components are highlighted by the characteristics of SystemC-AMS that, though being the reference language for smart system simulation, does not support descriptions that are both behavioral and conservative (as described in Sect. 3.2.2.1). The limited flexibility

of SystemC-AMS forces designers to adopt other HDLs (e.g., Verilog-AMS), that cannot be easily integrated with the frameworks and flows presented in this chapter.

For these reasons, this section shows how SystemC-AMS can be extended to support behavioral and conservative descriptions. Instead of adding a new abstraction level (with corresponding libraries and classes), the adopted approach uses SystemC-AMS existing primitives in a novel way [28]. Note that, due to the limitations of SystemC-AMS, supported models are strictly linear and time-invariant.

The starting point of the methodology is a Verilog-AMS behavioral description. In Verilog-AMS, a circuit is modeled as an abstract graph of nodes connected by branches [24]. System state is defined in terms of voltages (`V()`) and currents (`I()`) associated with nodes and branches. Relationships between nodes are modeled with algebraic and differential equations, called *simultaneous statements*.

Since SystemC-AMS is less expressive than Verilog-AMS, any Verilog-AMS simultaneous statement is reproduced by connecting a number of ELN elements. Given a Verilog-AMS description, each simultaneous statement is divided into basic contributions by finding the largest sub-equation that can be represented by a single ELN object. In linear and time-invariant descriptions, this corresponds to breaking the equation into the single addends.

Each addend is then mapped to the most suitable ELN primitive. As an example, an instance of the `sca_vsource` primitive is used to reproduce independent voltage sources, e.g., `V(a) <+ +8.01`. On the other hand, an instance of the `sca_vccs` primitive reproduces voltage controlled current sources, e.g., `I(a) <+ +4.02 V(b)`. ELN primitives must then be connected to reproduce the relationship expressed by the starting simultaneous statement. If the term on the left-hand side of the simultaneous statement is a current, SystemC-AMS instances are connected in parallel. Else, if the term is a voltage, instances are connected in series, by adding intermediate components. Figure 3.12 exemplifies these concepts on a simultaneous statement including a voltage controlled current source, a current controlled current source, and an independent current source.

Differential contributions require a more complex approach, as they model a derivative (or integrative) relationship between the current or voltage of two separate circuit nodes. SystemC-AMS, on the other hand, restricts differential behaviors to dependencies on single network nodes, through the adoption of capacitors (`sca_c` ELN module) or inductors (`sca_l`). To overcome this limitation, it is necessary to introduce an intermediate node that has no physical correspondence in the circuit, but that is rather used for describing the differential dependence. The node is connected to an inductor in case of a derivative construct (e.g., `I(a) <+ ddt(+4.02 V(b))`) and to a capacitor in case of an integrative construct (e.g., `I(a) <+ idt(+4.02 V(b))`). Suitable ELN primitives are then used to bind the evolution of the intermediate node to the nodes involved in the starting differential contribution.

As the application of the proposed approach may be tedious and error-prone, and thus prevent the application to industrial-size case studies, the whole methodology has been automated on top of the HIFSuite framework.

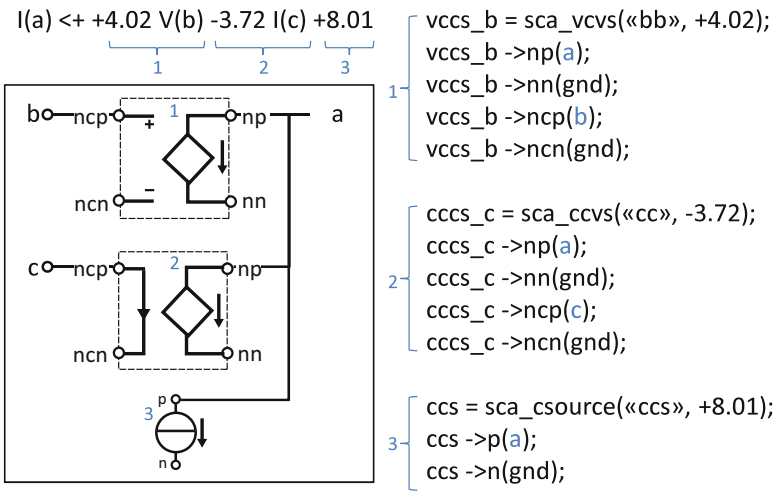

Fig. 3.12 Example of mapping of a Verilog-AMS simultaneous statement to SystemC-AMS. The simultaneous statement includes a voltage controlled current source (term 1, mapped to an instance of `sca_vccs`), a current controlled current source (term 2, mapped to a `sca_cccs`), and an independent current source (term 3, mapped to a `sca_csource`). Since the *left-hand side* of the simultaneous statement is a current construct, all ELN instances are connected in parallel. Non-connected terminals are connected to ground

Integration Strategies and Challenges

The code generation solutions presented in this section tackle the heterogeneity of smart systems by adopting a common language (i.e., SystemC and its extension), still preserving the heterogeneity in terms of MoC. However, interaction between different MoCs does not rely on manual, error-prone synchronization approaches, as for the functional level (Sect. 3.4.1). All synchronization is indeed transferred to the simulation kernel, that satisfies requests from all MoCs and abstraction levels.

Synchronization correctness is thus guaranteed by the underlying SystemC simulation kernel, that natively masters heterogeneous requests and takes care of synchronization issues between its extensions and MoCs. Furthermore, native converters allow to perform data conversion and to propagate events from one MoC to the other, without any manual intervention from the user. Still, the heterogeneity in terms of MoCs affects simulation performance, as data and synchronization conversion imply a computation overhead. Thus, the simplicity of integration comes at a price of simulation performance.

3.4.2.2 SystemVue-Based Simulation

SystemVue is an environment designed for easing the integration process. Its execution semantics is based on the synchronous dataflow MoC. As such, system behaviors are described by interconnecting basic blocks, expressing a functionality. The strength of SystemVue is that it provides predefined blocks as well as a C++

API to create libraries of custom components that can be included in a system simulation together with components shipped with SystemVue. This allows to easily integrate any C++ code, including manually designed code and code generated with the methodologies proposed in Sect. 3.4.1.

The first step to integrate a C++ external component in SystemVue is to specify its interface as names and data types of all the inputs, outputs, and parameters. The *interface of a SystemVue node* implemented in C++ is composed by a set of variables that are then specified to belong to the interface using the macros: `DEFINE_MODEL_INTERFACE`, `ADD_MODEL_OUTPUT`, and `ADD_MODEL_INPUT`. The data types of these variables, in order to be accepted by the macros, have to belong to a well-defined subset of the available C data types. Some data types, such as circular buffers, are implemented in the SystemVue support library. The other available data types are a subset of the C/C++ data types, that does not include the standard `unsigned integers`. This can be an issue, as normal `unsigned int` data types do not ensure that the span of data representation is the same on different architectures. For this reason, in order to assure the predictability of the number of bits used to represent data on the interface, every variable is declared as `double`. Then, before any computation step, the data read from the interface is assigned to a data structure using standard `Integer` and `Boolean` variables for computation. After the computation, the variables of the data structure are copied into the output variables. Figure 3.13 gives a sketch of the C++ code generated by HIFSuite for SystemVue. The left-hand side of the figure focuses on

```
class component : public
    SystemVueModelBuilder::DFModel
{
    unsigned int cycles;
    ...
    double input_1;
    double input_2;
    double output;
    ...
    struct component_iostruct{
        uint32_t input_1;
        bool input_2;
        uint32_t output;
    } io_exchange;
    ...
    DEFINE_MODEL_INTERFACE(component)
    {
        ADD_MODEL_INPUT(input_1);
        ADD_MODEL_INPUT(input_2);
        ADD_MODEL_OUTPUT(output);
        return true;
    }
    ...
```

```
    ...
    bool Run() {
        component_iostruct io_exchange(
            OUL,
            false,
            OUL
        );
        io_exchange.input_1 = input_1;
        io_exchange.input_2 = input_2;
        simulate(&io_exchange, cycles);
        output = io_exchange.output;
        return true;
    }
    ...
    void simulate(
        component_iostruct * io,
        unsigned int & cycles )
    {
        // A2tool generated implementation
        ...
    }
    ...
};
```

Fig. 3.13 Overview of the SystemVue-compliant C++ generated by HIFSuite

the interface and it shows the declaration of the interface variables, the input/output data structure, and the interface declaration. On the right-hand side of the figure the `Run` method exemplifies the usage of input/output variables and data structure.

In SystemVue, functionality is implemented in terms of four functions:

- `Setup()` is used to specify the rate of each port, in particular when using circular buffers, in the node interface. The default value is uni-rate, and it is not mandatory to implement this function.
- `Initialize()` is executed during the initialization of the dataflow, thus should be used to run all the initialization code necessary to the node functionality.
- `Run()` is the main method, as it contains the functionality that has to be executed at every simulation step. Its execution is scheduled by SystemVue, according to the dataflow structure, and the rate of the input/output ports of the node.
- `Finalize()` performs any post-simulation coding that the model needs to perform, such as closing file or de-allocating memory.

In order to respect this interface, C++ code generation techniques must be customized and extended to ensure SystemVue support. As an example, the code generated by HIFSuite uses the `Initialize()` method to reset all variables and data structures of the component. The `Run()` function, as depicted on the left part of Fig. 3.13, handles the input/output as discussed above and it calls the code generated by A^2T (i.e., simulate) to emulate component evolution, passing the input/output structure as parameter. When the simulate function returns, the output variables are written according to computed component evolution.

A final integration issue arises whenever components adopt different MoCs. In SystemVue, synchronization and communication among different nodes is based in SDF, that forces the insertion of a delay in every loop among different components. Thus, it is necessary to insert delays to break the loops between connected components, for instance, between a bus and a CPU or between bus and peripherals. However, the generalized insertion of such delays can produce synchronization problems due to the modification of simulation delays that usually guarantee the correct behavior of a digital system. For this reason, digital components in loop are automatically merged by HIFSuite in a single component and abstracted with A^2T as a single component.

By following these guidelines, SystemVue easies the integration of existing code, as the designer must simply match the APIs for the designed components, while the synergy with HIFSuite automatically translates pre-design digital and analogue components and all synchronization issues are left to the simulation kernel.

3.5 Experimental Validation of Proposed Methodologies

The goal of this section is to support the proposed analysis and methodologies with experimental evidence. To this extent, the proposed examples focus on single code generation techniques and on the simulation of a complex smart system case study achieved through SystemVue.

Table 3.1 Abstraction alternatives of digital components for functional and transactional simulation

	Modelsim (VHDL/ Verilog)	SystemC RTL, SystemC data types	Abstract C++, HDTLib data types types (SystemC top)		Abstract C++, C++ native data (SystemC top)		Abstract C++, C++ native data types (pure C++ top)	
Design	T (s)	T (s)	T (s)	S (x)	T (s)	S (x)	T (s)	S (x)
AES	72.3	850.9	332.5	2.6	8.0	106.4	7.1	119.8
Camellia	1823.7	25,433.3	9022.6	2.8	8.0	3179.2	3.3	7707.1
DES56	707.5	7608.5	1941.1	3.9	8.5	895.1	4.6	1654.0
SHA512	1758.9	6302.1	2452.4	2.5	12.6	371.2	3.4	1377.2
XTEA	171.8	975.2	260.9	3.7	18.0	54.2	3.4	286.8

3.5.1 *Validation of HIFSuite-Based Language Conversion Techniques*

The automatic abstraction of digital components to SystemC/TLM and to C++ plays a key role in the simulation of a smart platform at both the functional and the transactional levels. Thus, its effectiveness must be evaluated in depth.

Table 3.1 reports simulation time (`T(s)`) for some VHDL and Verilog digital components, together with the speedup achieved through the automatic abstraction by A^{2}T with the support of HDTLib or DDT for data type abstraction. The reference simulation time is generated by Modelsim (column *Modelsim*). The generated code may be managed through either a SystemC top-level module (columns labeled with *SystemC top*) or a C++ main simulation file (*pure C++ top*). This distinction allows to analyze all the scenarios outlined in Fig. 3.5, thus covering both the functional abstraction level (single/multiple MoC) and the transactional abstraction level (through the adoption of SystemC or SystemVue for component aggregation).

Results clearly conclude that the automatic abstraction of digital components is extremely efficient (up to three orders of magnitude in speedup) in the case of RTL modules converted to C++ for single MoC functional simulation or for SystemVue-based transactional simulation. In the other cases, the effectiveness of the abstraction process is limited on single components, but it still produces a simulation advantage whenever the platform model must be built by aggregating different components.

3.5.2 *Validation of the Mapping of Analogue Conservative Descriptions to SystemC-AMS*

Mapping of analogue conservative descriptions to SystemC-AMS proved to be a complex step, due to the requirements in terms of construct coverage and of application of energy conservation laws. In order to prove the effectiveness of the overall methodology, we applied the overall approach to a complex industrial case

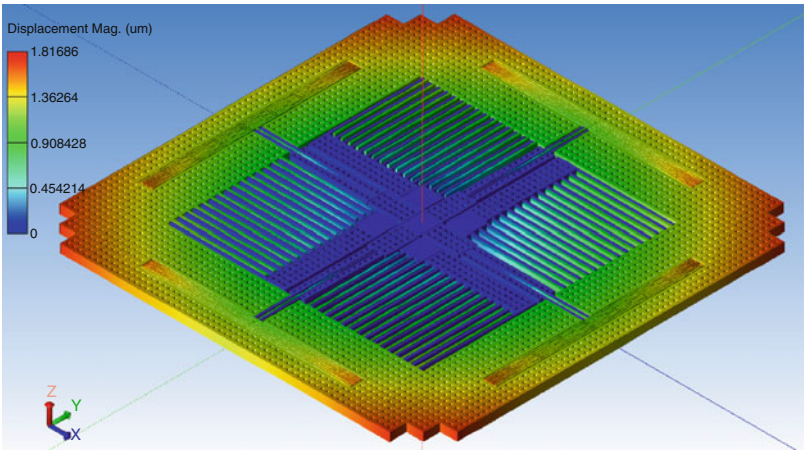

Fig. 3.14 Three-dimensional model of the accelerometer in the MEMS+ design simulator

Table 3.2 Characteristics of the original Verilog-AMS MEMS design

Lines of code		89
Equations	Voltage sources	10
	Current sources	15
Node declarations	Interface	14
	Internal	14
Contributions	Independent	4
	Voltage	59
	Current	0
	Derivative	12
	Integrative	0

study, developed in the context of the SMAC project. Application to this industrial case studies was eased though the implementation of an automatic tool, called ABACuS (Analogue BehAvioural Conservative SystemC-AMS), that leverages HIFSuite to ease the conversion process.

The adopted case study is a *two-dimensional MEMS accelerometer* implemented in Verilog-AMS by means of the MEMS design platform MEMS+, that supports automatic Verilog-AMS code generation [12], starting from three-dimensional physical models as the one depicted in Fig. 3.14. Table 3.2 reports the main characteristics of the MEMS design, both in terms of simultaneous statements and of types of contributions. The MEMS design features most of types of supported contributions, thus showing the application and validation of a significant part of the methodology on a single case study.

Table 3.3 shows the results of the application of ABACuS to the MEMS design. The table shows the number of lines of code of the resulting SystemC-AMS implementation, the number of added nodes and of instances of SystemC-AMS primitives. The number of lines of codes is increased tenfold (precisely, 11.12x),

Table 3.3 Characteristics of the generated SystemC-AMS MEMS design

Lines of code		1474
Added node declarations		12
SystemC-AMS primitive instantiations	`sca_r`	93
	`sca_vsource`	4
	`sca_vcvs`	32
	`sca_ccvs`	0
	`sca_csource`	0
	`sca_vccs`	48
	`sca_cccs`	0
	`sca_l`	12
	`sca_c`	0

Table 3.4 Characteristics of the execution of ABACuS on the MEMS design

Overall		17.48 s
HIFSuite tools	Conversion to HIF	1.86 s
	Conversion to SystemC-AMS	7.81 s
ABACuS	Node management	0.94 s
	Division into contributions	0.29 s
	ELN component instantiations	6.58 s

as the SystemC-AMS generated by the methodology is more verbose than Verilog-AMS. Each contribution requires the instantiation of the ELN primitive, plus the corresponding explicit port binding. Furthermore, the number of ELN primitives is higher than the number of Verilog-AMS contributions. This is due to the presence of 12 derivative contributions in the original Verilog-AMS code. Each such contribution determines the instantiation of three ELN primitives (as explained in Sect. 3.4.2.1). As a result, of the 188 resulting SystemC-AMS ELN instances:

- 93 correspond to resistors added to connect each SystemC-AMS node to ground;
- 59 correspond to voltage source contributions;
- 36 are generated by the 12 derivative constructs, that determine also the declaration of 12 additional internal nodes.

Fast code generation is a major advantage of the proposed approach. Table 3.4 highlights that code generation is almost instantaneous (17.48 s overall), and that most of the effort is spent in the HIFSuite conversions (55 %). The most costly step of ABACuS lies in the mapping from Verilog-AMS contributions to ELN primitives and in their instantiation (37 %). On the other hand, node management and the separation of Verilog-AMS equations into single contributions are almost immediate.

The generated code was validated by comparing its execution *w.r.t.* the original Verilog-AMS code, run by using the Questa simulator [23]. SystemC-AMS simulation was run by adopting the same input stimula of the Verilog-AMS implementation, and with a 1us timestep. SystemC-AMS proved to be slightly faster than the Verilog-AMS execution (28.02 s and 33.72 s, respectively). At the same

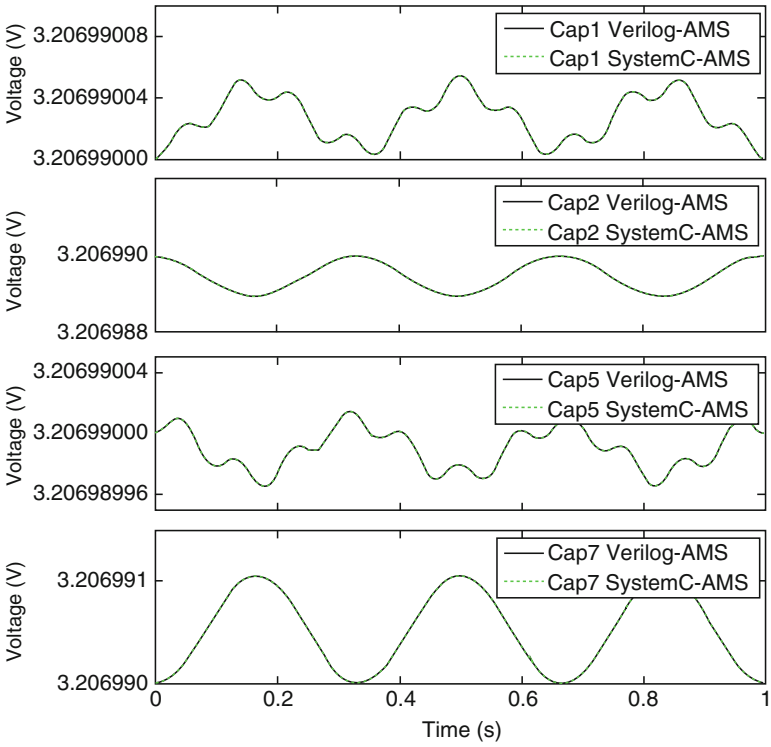

Fig. 3.15 Evolution of the MEMS outputs for Verilog-AMS (*solid*) and SystemC-AMS (*dashed*)

time, the average error in the computation of the MEMS outputs is 0.02 %. This confirms the visual accuracy evident from Fig. 3.15, where the Verilog-AMS and SystemC-AMS curves are almost totally overlapping. The small error is due to the different management of time in the two simulators: SystemC-AMS adopts a fixed timestep, while Verilog-AMS can adapt the length of the timestep over time, thus reaching a higher accuracy. The low error rate highlights the effectiveness of the generated code, both in terms of accuracy and of simulation speed.

3.5.3 Adoption of SystemVue for a Heterogeneous Case Study

The final example collects all previous results to show a transactional level simulation of a smart system based on SystemVue integrating a number of heterogeneous components. The starting point is complex heterogeneous smart system, developed with the goal of representing a generic smart system. The system, called *open source test case* (OSTC), includes eight modules covering digital HW, embedded SW, RF-transceiver, network elements, and a MEMS sensor (i.e., the accelerometer).

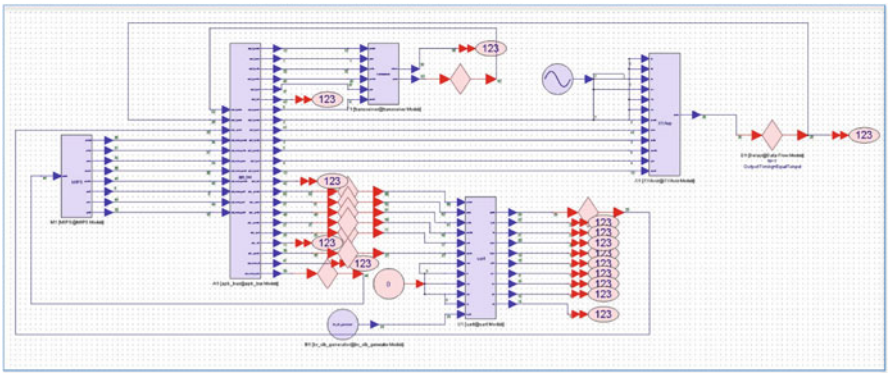

Fig. 3.16 SystemVue schematic of the OSTC. The left-most component represents the sub-system memory and CPU. This is connected to the component implementing the bus. Components on the right-hand side implement the peripherals. Red rhombuses are delays introduced to break dataflow loops. Red circles represent sinks collecting the outputs of the OSTC

Table 3.5 Simulation time for the three different simulation scenarios in SystemVue

Scenario	Simulation time (s)	Speed-up
Co-simulation of all digital HW	278.59	–
Co-simulation of one digital HW	153.23	1.8×
C++-based simulation	36.32	7.7×

Such modules are extremely heterogenous in terms of language, as they are described in SystemC, VHDL, Verilog, Verilog-AMS and C++. An exhaustive description of the OSTC will be the focus.

Figure 3.16 shows the SystemVue representation of the OSTC. Each module has been imported in SystemVue after its abstraction to C++, performed by using HIFSuite. SystemVue supports co-simulation, thus allowing the comparison of the following scenarios:

- Co-simulation of all digital HW components;
- Co-simulation of one digital HW component;
- Homogeneous C++-based simulation.

The simulation scenario used for all the models simulates 100 ms of system execution, with a timestep of 100 ns. The inputs of the accelerometer are sinusoidal stimula, and the software application is pre-loaded in the memory. The software takes care of system boot and peripheral initialization. Then, the application repeatedly reads data from the accelerometer, computes the data, and sends the results to the digital hardware and the network interface.

Table 3.5 shows the time needed to simulate the three different scenarios. What appears clear from these results is that the number of simulators instantiated, hence the number of co-simulation interfaces employed, heavily impacts performance. In particular, it worth notice that, in this case, every co-simulation interface (two

in the case of the first entry of the table, one in the second), seems to introduce around 120 s overhead *w.r.t.* the simulation without co-simulation interface, thus introducing an overhead of about 80 %. As a result, the impact of interfaces and conversion layers between different tools seems highly relevant and strictly dependent on the number of used interfaces and external tools. The limited speed-up is mainly affected by the low abstraction capability of the two main digital components of the OSTC. Such components are indeed described at gate level rather than at RTL, thus the abstraction to C++ is not extremely effective. Higher speedups can be obtained by using real RTL components, such as the ones reported in Table 3.1.

3.6 Concluding Remarks

This chapter provided a formalization of the abstraction levels and design domains of a smart system. This taxonomy allows to identify a precise role in the design flow for co-simulation and simulation scenarios, and to examine the impact of heterogeneous or homogeneous MoCs. Moreover, a methodology has been proposed to move from the co-simulated heterogeneity to a simulatable homogeneous representation of the entire smart system at two level of abstraction: functional level and transactional level. At functional level, all components are implemented in C++, with the goal of understanding the role of the underlying synchronization and simulation semantics and their overhead on simulation performance. At transactional level, two widespread simulation frameworks, i.e., SystemC and SystemVue, have been adopted to ease code integration, even in presence of very heterogeneous design flows.

References

1. Accellera Systems Initiative, SystemC (2015), accellera.org/downloads/standards/systemc
2. Accellera Systems Initiative, SystemC-AMS 2.0 Standard (2015), accellera.org/downloads/standards/systemc
3. Accellera Systems Initiative, SystemC TLM (Transaction-Level Modeling) (2015), accellera.org/activities/working- groups/systemc-tlm
4. Agilent Technologies, SystemVue Electronic System-Level (ESL) Design Software (2015), www.home.agilent.com/en/pc-1297131/systemvue
5. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passarone, A. Sangiovanni-Vincentelli, Metropolis: an integrated electronic system design environment. Computer **36**(1), 45–52 (2003)
6. N. Bombieri, G.D. Guglielmo, M. Ferrari et al., HIFSuite: tools for HDL code conversion and manipulation. EURASIP J. Embed. Syst., 1–20 (2010)
7. N. Bombieri, F. Fummi, G. Pravadelli, Abstraction of RTL IPs into embedded software, in *ACM/IEEE Design Automation Conference* (2010), pp. 24–29

8. N. Bombieri, F. Fummi, G. Pravadelli, Automatic abstraction of RTL IPs into equivalent TLM descriptions. IEEE Trans. Comput. **60**(12), 1730–1743 (2011)
9. N. Bombieri, F. Fummi, V. Guarnieri, F. Stefanni, S. Vinco, HDTLib: an efficient implementation of SystemC data types for fast simulation at different abstraction levels. Des. Autom. Embed. Syst. **16**(2), 115–135 (2012)
10. F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, E. Aboulhamid, A SystemC/Simulink co-simulation framework for continuous/discrete-events simulation, in *Proceedings of the IEEE International Behavioral Modeling and Simulation Conference* (2007), pp. 1–6
11. J.C. Butcher, *Numerical Methods for Ordinary Differential Equation* (Wiley, New York, 2003)
12. Coventor, Inc., MEMS+: MEMS Simulation Software (2015), www.coventor.com/mems-solutions/products/mems
13. G. De Micheli, R. Ernst, W. Wolf, *Readings in Hardware/Software Co-Design* (Morgan Kaufmann, San Francisco, 2001)
14. L. Di Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, S. Vinco, A formal support for homogeneous simulation of heterogeneous embedded systems, in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems* (2012), pp. 211–219
15. EDALab s.r.l., HIFSuite (2015), www.hifsuite.com
16. G. Frehse, PHAVer: algorithmic verification of hybrid systems past HyTech, in *Hybrid Systems: Computation and Control*. Lecture Notes in Computer Science, vol. 3414 (Springer, Berlin/Heidelberg, 2005), pp. 258–273
17. F. Fummi, M. Loghi, M. Poncino, G. Pravadelli, A cosimulation methodology for HW/SW validation and performance estimation. ACM Trans. Des. Autom. Electron. Syst. **14**, 23:1–23:32 (2009)
18. D.D. Gajski, N.D. Dutt, A.C.-H. Wu, S.Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design* (Kluwer Academic Publishers, Norwell, 1992)
19. L.D. Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, S. Vinco, UNIVERCM: the UNIversal VERsatile computational model for heterogeneous system integration. IEEE Trans. Comput. **62**(2), 225–241 (2013)
20. T. Henzinger, The theory of hybrid automata, in *IEEE Symposium on Logic in Computer Science (LICS)* (1996), pp. 278–292
21. C.-J. Hsu, J. Pino, F.-J. Hu, A mixed-mode vector-based dataflow approach for modeling and simulating LTE physical layer, in *Proceedings of the ACM/IEEE Design Automation Conference* (2008), pp. 18–23
22. E.A. Lee, Overview of the Ptolemy project (2001), ptolemy.eecs.berkeley.edu
23. Mentor Graphics, Questa Advanced Simulator (2015), www.mentor.com/products/fv/questa
24. S. Mijalkovic, Advanced circuit and device modeling with Verilog-A, in *Proceeding of the IEEE Microelectronics* (2006), pp. 439–442
25. P. Schneider, C. Bayer, K. Einwich, A. Kohler, System level simulation - a core method for efficient design of MEMS and mechatronic systems, in *Proceedings of the IEEE Systems, Signals and Devices* (2012), pp. 1–6
26. The MathWorks Inc., Stateflow: Design and Simulate State Machines and Control Logic (2007), www.mathworks.com/products/stateflow/
27. B. van Beek, The Compositional Interchange Format: Introduction (2015), se.wtb.tue.nl/sewiki/media/vanbeek/cifintro.pdf
28. F.F.S. Vinco, M. Lora, Conservative behavioural modelling in SystemC-AMS, in *IEEE/ECSI FDL* (2015), pp. 1–8