Semiformal Assertion-Based Verification of Hardware/Software Systems in a Model-Driven Design Framework

22

Graziano Pravadelli, Davide Quaglia, Sara Vinco, and Franco Fummi

Abstract

Since the mid-1990s, Model-Driven Design (MDD) methodologies (Selic, IEEE Softw 20(5):19–25, 2003) have aimed at raising the level of abstraction through an extensive use of generic models in all the phases of the development of embedded systems. MDD describes the system under development in terms of abstract characterization, attempting to be generic not only in the choice of implementation platforms but even in the choice of execution and interaction semantics. Thus, MDD has emerged as the most suitable solution to develop complex systems and has been supported by academic (Ferrari et al., From conception to implementation: a model based design approach. In: Proceedings of IFAC symposium on advances in automotive control, 2004) and industrial tools (3S Software CoDeSys, 2012. http://www.3s-software.com; Atego ARTiSAN, 2012. http://www.atego.com/products/artisan-studio; Gentleware Poseidon for UML embedded edition, 2012. http://www.gentleware.com/uml-software-embedded-edition.html; IAR Systems IAR visualSTATE, 2012. http://www.iar.com/Products/IAR-visualSTATE/; rhapsodyIBM Rational Rhapsody, 2012. http://www.ibm.com/software/awdtools/rhapsody; entarchSparx Systems Enterprise architet, 2012. http://www.sparxsystems.com.au; Aerospace Valley TOPCASED project, 2012. http://www.topcased.org). The gain offered by the adoption of an MDD approach is the capability of generating the source code implementing the target design in a systematic way, i.e., it avoids the need of manual writing. However, even if MDD simplifies the design implementation, it does not prevent the designers from wrongly defining the design behavior. Therefore, MDD gives full benefits if it also integrates *functional verification*.

G. Pravadelli (✉) • D. Quaglia • F. Fummi
Università di Verona, Verona, Italy
e-mail: graziano.pravadelli@univr.it; davide.quaglia@univr.it; franco.fummi@univr.it

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

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_23

In this context, Assertion-Based Verification (ABV) has emerged as one of the most powerful solutions for capturing a designer's intent and checking their compliance with the design implementation. In ABV, specifications are expressed by means of formal properties. These overcome the ambiguity of natural languages and are verified by means of either static (e.g., model checking) or, more frequently, dynamic (e.g., simulation) techniques. Therefore ABV provides a proof of correctness for the outcome of the MDD flow. Consequently, the MDD and ABV approaches have been combined to create efficient and effective design and verification frameworks that accompany designers and verification engineers throughout the system-level design flow of complex embedded systems, both for the Hardware (HW) and the Software (SW) parts (STM Products radCHECK, 2012. http://www.verificationsuite.com; Seger, Integrating design and verification – from simple idea to practical system. In: Proceedings of ACM/IEEE MEMOCODE, pp 161–162, 2006). It is, indeed, worth noting that to achieve a high degree of confidence, such frameworks require to be supported by *functional qualification* methodologies, which evaluate the quality of both the properties (Di Guglielmo et al. The role of mutation analysis for property qualification. In: 7th IEEE/ACM international conference on formal methods and models for co-design, MEMOCODE'09, pp 28–35, 2009. DOI 10.1109/MEMCOD.2009.5185375) and the testbenches which are adopted during the overall flow (Bombieri et al. Functional qualification of TLM verification. In: Design, automation test in Europe conference exhibition, DATE'09, pp 190–195, 2009. DOI 10.1109/DATE.2009.5090656). In this context, the goal of the chapter consists of providing, first, a general introduction to MDD and ABV concepts and related formalisms and then a more detailed view on the main challenges concerning the realization of an effective semiformal ABV environment through functional qualification.

Acronyms

ABV	Assertion-Based Verification
CTL	Computation Tree Logic
DUV	Design Under Verification
EFSM	Extended Finite-State Machine
ES	Embedded System
ESL	Electronic System Level
FSM	Finite-State Machine
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
LLVM	Low-Level Virtual Machine
LTL	Linear Time Logic
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MDA	Model-Driven Architecture

MDD	Model-Driven Design
MLBJ	Multi-Level Back Jumping
MMIO	Memory-Mapped I/O
MoC	Model of Computation
OMG	Object Management Group
OSCI	Open SystemC Initiative
OVL	Open Verification Library
PIM	Platform Independent Model
PSL	Property Specification Language
PSM	Platform Specific Model
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expression
SoC	System-on-Chip
SVA	System Verilog Assertions
SW	Software
TLM	Transaction-Level Model
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Contents

22.1 Introduction to Model-Driven Design

The focus of MDD is to elevate the system development to a higher level of abstraction than that provided by HW description languages (e.g., VHDL and Verilog) for Hardware (HW) aspects and by third-generation programming languages for Software (SW) aspects [92]. The development is based on *models*, which are abstract characterizations of requirements, behavior, and structure of the embedded system without anticipating the implementation technology.

Due to the noticeable effort of the Object Management Group (OMG) [84], the Unified Modeling Language (UML) [85] was originally adopted as the reference modeling language for describing software, and then it was also applied to the description of embedded hardware. UML provides general-purpose graphic elements to create visual models of systems and attempts to be generic in both the integration and the execution semantics. Due to such a general-purpose semantics, more specific *UML profiles* have been introduced for dealing with specific domains or concerns. They extend subsets of the UML meta-model with new standard elements, and they refine the core UML semantics to cope with particular hardware/software problems. For example, the SysML [97] profile supports the specification, the analysis, and the design of complex systems, which may include physical components. The Gaspard2 [57] profile, instead, supports the modeling of System-on-Chip (SoC). The synchronous reactive [35] profile provides a restrictive set of activity diagrams and sequence diagrams with a clear and semantically sound way of generating valid execution sequences. The Modeling and Analysis of Real-Time Embedded Systems (MARTE) [83] profile adds capabilities to UML for modeling and analysis of real-time and embedded systems. Its modeling concepts provide support for representing time and time-related mechanisms, the use of concurrent resources and other embedded systems characteristics (such as memory capacity and power consumption). The analysis concepts, instead, provide model annotations for dealing with system property analysis, such as schedulability analysis and performance analysis. It is worth noticing that other UML profiles exist for hardware-related aspects such as system-level modeling and simulation [81,90]. Other hardware-oriented profiles and a comparison of them are clearly described in [18]. Model-Driven Design (MDD) has been also used for modeling embedded systems that interact together through communication channels to build distributed applications. In this context, the MDD approach consists in using a *UML deployment diagram* to capture the structural representation of the whole distributed application. MARTE stereotypes (e.g., the MARTE-GQAM sub-profile and MARTE nonfunctional properties) can be used to represent attributes such as throughput, price, and power consumption. Furthermore some aspects (e.g., node mobility) require the definition of an ad-hoc UML network profile [46].

Besides the standard UML supported by OMG, some proprietary variants of the UML notations also exist. The most famous ones are the MathWorks' Stateflow and Simulink [98] formalisms. They use finite-state, machine-like and functional-block, diagram-like models, respectively, for specifying behavior and structure of reactive hardware/software systems with the aim of rapid embedded SW prototyping and engineering analysis. Several MDD tools on the market support UML and Model-Driven Architecture (MDA). The underlying idea of MDA is the definition of models at different levels of abstraction which are linked together to form an implementation. MDA distinguishes the conceptual aspects of an application from their representation on specific implementation technologies. For this reason, the MDA design approach uses Platform Independent Models (PIMs) to specify what an application does, and Platform Specific Models (PSMs) to specify how the application is implemented and executed on the target technology. The key element of an MDA approach is the capability to automatically transform models: transformation of PIM into PSM enables realizations, whereas transformations between PIMs enable integration features.

22.2 Introduction to Assertion-Based Verification

Assertion-based verification aims at providing verification engineers with a way to formally capture the intended specifications and checking their compliance with the implemented embedded system. Specifications are expressed by means of formal properties defined according to temporal logics, e.g., Linear Time Logic (LTL) or Computation Tree Logic (CTL), and expressed by means of assertion languages, like Property Specification Language (PSL) [64].

Approaches based on ABV are traditionally classified in two main categories: *static* (i.e., *formal*) and *dynamic* (i.e., *simulation based*). In static Assertion-Based Verification (ABV), formal properties, representing design specifications, are exhaustively checked against a formal model of the design by exploiting, for example, a model checker. Such an exhaustive reasoning provides verification engineers with high confidence in system reliability. However, the well-known state space explosion problem limits the applicability of static ABV to small/medium-size, high-budget, and safety-critical projects [67]. On the contrary, thanks to the scalability provided by simulation-based techniques, dynamic ABV approaches are nowadays preferred for verifying large designs, which have both reliability requirements and stringent development cost/time-to-market constraints. In particular, in the hardware domain, dynamic ABV is affirming as a leading strategy in industry to guarantee fast and high-quality verification of hardware components [24, 80], and several verification approaches have been proposed [50, 51]. In dynamic ABV, properties are compiled into *checkers* [17], i.e., modules that capture the behavior of the corresponding properties and monitor if they hold with respect to the design, when the latter is simulated by using a set of (automatically generated) stimuli.

22.3 Integrating MDD and ABV

Even if MDD simplifies software implementation, it does not prevent the designer from wrongly defining system behavior. Certain aspects concerning the verification of the code generated by MDD flows are automated, as, for example, the structural analysis of code, but specification conformance, i.e., functional verification, is still a human-based process [47,55]. Indeed, the de facto approach to guarantee the correct behavior of the design implementation is monitoring system simulation: company verification teams are responsible for putting the system into appropriate states by generating the required stimuli, judging when stimuli should be executed, manually simulating environment and user interactions, and analyzing the results to identify

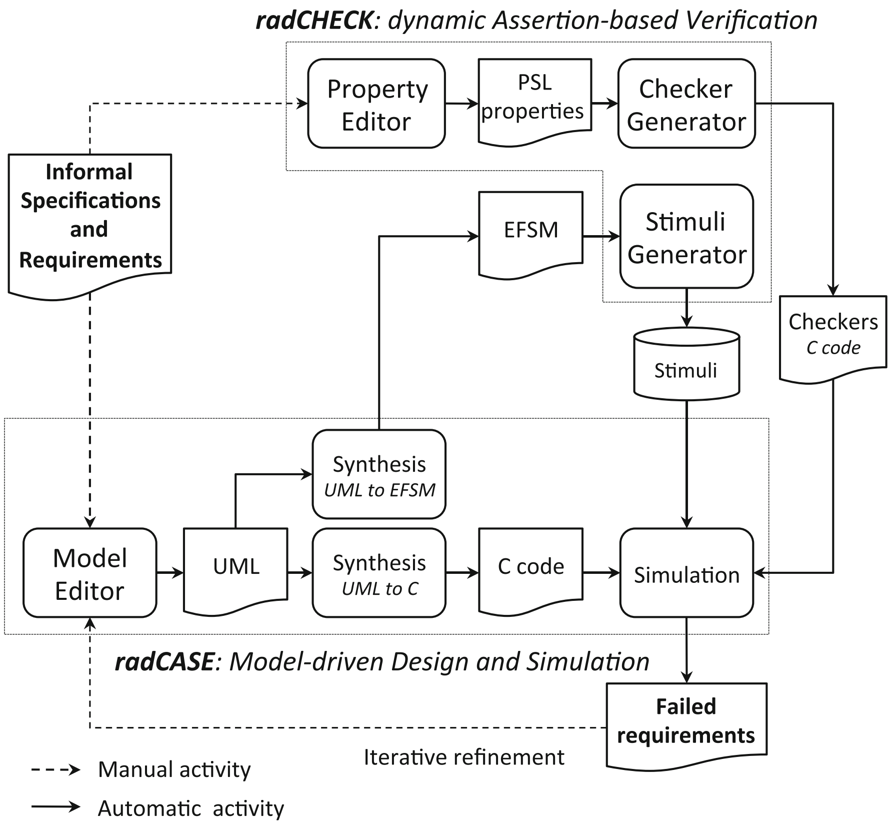


Fig. 22.1 The combined model-driven and verification framework implemented in the RadSuite

unexpected behaviors. MDD and dynamic ABV, if combined in a comprehensive framework, enable automatic functional verification of both embedded SW and HW components. The approach generally relies on a design and verification framework composed of two environments: a UML-like modeling and development environment - supporting model-driven design - and a dynamic ABV environment that supports assertion definition and automatic checkers and stimuli generation.

An example of this design and verification flow, which integrates MDD and ABV, is represented by the commercial RadSuite, composed by radCASE and radCHECK [38] (Fig. 22.1). Starting from informal specifications and requirements, the designer, with the model editor of radCASE, defines the system model by using a UML-based approach. Concurrently, with the property editor of radCHECK, he/she defines a set of PSL assertions that the system must fulfill. Then, radCASE automatically translates the UML specifications in the C-code implementation, and it automatically extracts an Extended Finite-State Machine (EFSM) model (see Sect. 22.4.1.1 for this formalism) to support automatic verification. At the same time, radCHECK can be used to automatically generate executable checkers from the defined PSL assertions. The dynamic ABV is guided by stimuli automatically generated by a corner-case-oriented concolic stimuli generator that exploits the EFSM model to explore the system state space (see Sect. 22.7). Checkers execute concurrently with the Design Under Verification (DUV) and monitor if it causes any failure of the corresponding properties. The designer uses the resulting information, i.e., failed requirements, for refining the UML specifications incrementally and in an iterative fashion.

22.4 Models and Flows for Verification

The key ingredient underpinning an effective design and verification framework based on MDD and ABV is represented by the possibility of defining a model of the desired system and then automatically deriving the corresponding simulatable description to be used for virtual prototyping. Selecting the formalism to represent the model is far from being a trivial choice, as the increasing complexity and heterogeneity of embedded systems generated, over time, a plethora of languages and different representations, each focusing on a specific subset of the Embedded System (ES) and on a specific domain [44]. Examples are EFSMs, dedicated to digital HW components and cycle-accurate protocols, hybrid automata for continuous physical dynamics, high-level UML diagrams for high-level modeling of hardware, software, and network models. This heterogeneity reflects on the difficulty of standardizing automatic approaches that allow the conversion of the high-level models in executable specifications (e.g., SystemC/C code). Such an automatic conversion is indeed fundamental to reduce verification costs, particularly in the context of virtual prototyping of complex systems that generally integrate heterogeneous components through both bottom-up and top-down composition flows. In this direction, automata-based formalisms represent the most suitable solutions to enable a precise mapping of the model into simulatable descriptions. Thus, the following discussion in this section is intended to summarize the main automata-based formalisms available as state of the art, together with bottom-up and top-down flows, whose combined adoption allows the generation of a homogeneous simulatable description of the overall system.

22.4.1 Automata-Based Formalisms

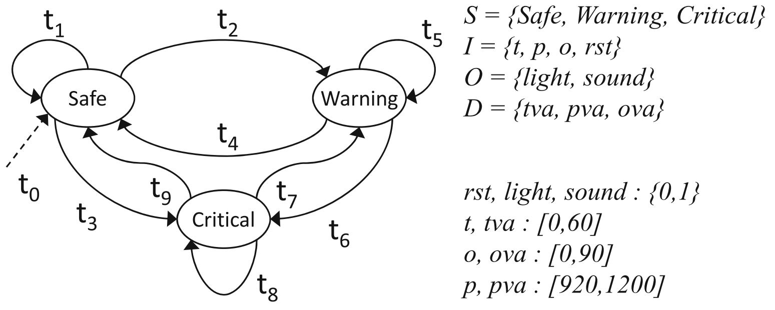
The most widespread models for representing the behavior of a component or a system are based on automata, i.e., models that rely on the notions of states and of transitions between states. The simplest automata-based model, i.e., the finite-state machine, has proved to be too strict to allow a flexible and effective view of modern components. This originated a number of extensions, each targeting specific aspects and domains, as summarized in the following of this section.

22.4.1.1 Extended Finite-State Machines

EFSMs extend standard Finite-State Machines (FSMs) to allow a more compact representation of the system that reduces the risk of state explosion for complex designs [4]. An EFSM differs from the traditional FSM, since transitions between states are not labeled in the classical form input/output values, but they take care of the values of internal variables too. Practically, transitions are labeled with an *enabling function*, which represents the guard of the transition, and an *update function*, which specifies how the values of variables and outputs evolve when the transition is fired upon the satisfaction of its guard.

To exemplify the concept, Fig. 22.2 reports the EFSM of a simplified in-flight safety system. The states of the EFSM are $S = \{Safe, Warning, Critical\}$, where $Safe$ is the reset state. The input variables are $I = \{t, p, o, rst\}$ and represent the corresponding temperature, pressure, and oxygen variables, whereas rst represents the EFSM reset signal. $O = \{light, sound\}$ is the set of output variables corresponding to light and sound controls. Finally, $D = \{tva, pva, ova\}$ is the set of internal variables of the EFSM. For each transition, the enabling function and update function are reported in the table under the figure. For readability, only a reset transition t_0 is depicted with a dotted arrow as a representative of each of the reset transitions outgoing from the states of the EFSM and entering in *Safe*.

Unfortunately, the semantics of EFSMs is strictly discrete, and it does not support continuous-time physical models. Thus it cannot, for instance, represent an ES with its continuous-time environment.



TRANSITION	ENABLING FUNCTION	UPDATE FUNCTION
t_0	$rst = 1$	$tva = 0;\ ova = 0;\ pva = 0;$ $light = 0;\ sound = 0;$
t_1	$t < 42 \wedge p \geq 980$	$-$
t_2	$t \geq 42$	$tva = t;\ pva = p;\ light = 1;$
t_3	$t < 42 \wedge p > 1020$	$tva = t;\ pva = p;\ sound = 1;$
t_4	$t < 42 \wedge p \leq 1020$	$light = 0;\ sound = 0;$
t_5	$t \geq 42 \wedge p > 1020 \wedge$ $t \leq tva \wedge o \geq 18$	$tva = t;$
t_6	$t \geq tva \wedge o < 18$	$ova = o;\ sound = 1;$
t_7	$o > ova \wedge p < pva$	$sound = 0;$
t_8	$o \leq ova \wedge p \geq pva \wedge$ $t \geq 42 \wedge p > 1020$	$-$
t_9	$t < 42 \wedge p \leq 1020$	$light = 0;\ sound = 0;$

Fig. 22.2 An EFSM specification of a simplified in-flight safety system

22.4.1.2 Hybrid Automata

A hybrid automaton is modeled as a set of states and transitions, but, as opposed to EFSMs, it supports both discrete time and continuous time dynamics [58]. The discrete time dynamics coincides with FSM semantics, and it is implemented through transitions between states that respond to system evolution and to synchronization events. The continuous-time dynamics is implemented by the states, which are associated with two predicates: the *flow* predicate that constrains the evolution of continuous variables into the state, and the *invariant* predicate, that specifies whether it is possible to remain into the state or not, depending on a set of conditions on variables. Variables can be assigned continuous values, as opposed to EFSMs.

Figure 22.3 depicts an example of a simple hybrid automaton. If compared with Fig. 22.2, the automaton now associates each state with an invariant condition (e.g., $x \geq 18$, for the state $S0$) and with a flow predicate which shows the rate of change of the variable x with time (e.g., $x' = -0.1x$, for the state $S0$, where x' represents the first derivative of variable x). Furthermore, a synchronization event, $close$, is used to force the transition to $S1$, irrespective of the current state of variable x.

A special class of automata, namely, *timed automata*, introduces the notion of time [6]. Time evolution is modeled with dense variables, called clocks, whose evolution is constrained by predicates and used in transition guards. Furthermore, events and variables are associated with a time stamp. As a result, automaton evolution depends also on time. This is extremely useful in the modeling of real-time systems with time constraints.

Hybrid automata are especially suited for modeling control scenarios, modeling a tight interaction between a controller and a continuous time environment. However, this formalism is suited for modeling only high-level systems. Unfortunately, describing cycle-accurate hardware behaviors as well as software functionalities (such as interrupt handling) would exponentially increase the complexity of the related model, and it would lead to state space explosion [56].

22.4.1.3 UML Diagrams

UML is a standardized general-purpose modeling language, specially suited for modeling SW intensive systems, but often adopted also for modeling HW components and networked systems [85]. UML includes a set of graphic notation techniques for clearly representing different aspects of a system, i.e., its structure (*structural diagrams*) or its behavior (*behavioral diagrams*).

The most useful class from the point of view of MDD approaches for HW/SW systems is represented by *behavioral diagrams*, which model what happens in the system, either in terms of internal behavior or from the communication perspective.

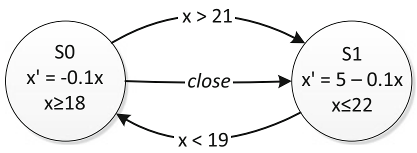


Fig. 22.3 Example of a hybrid automaton

Behavioral diagrams can be further classified into different classes, among which the most relevant for modeling HW/SW systems are:

- *Activity diagrams*, which represent the data and control flow between activities
- *Interaction diagrams*, which represent the interaction between collaborating parts of a system, in terms of message exchange (*sequence diagram* and *communication diagrams*), and of state evolution depending on timed events (*timing diagrams*)
- *State machine diagrams*, which are automata-based model representing the state transitions and actions performed by the system in response to events

Besides behavioral diagrams, *deployment diagrams*, belonging to the category of structural diagrams, have been also used in the context of MDD to capture the structural representation of network aspects, in conjunction with MARTE stereotypes [46].

In general, UML diagrams have been specialized to fit many different domains through the definition of profiles. However, their diagrams are too high level to represent cycle accurate models and physical models with a sufficient accuracy, without incurring in the state explosion problem or degenerating into standard FSMs.

Figure 22.4 shows a sequence diagram example. Sequence diagrams show how processes operate and their interactions, represented as exchanged messages. For this reason, they are the most common diagrams to specify system functionality, communication, and timing constraints. Lifelines (the vertical dashed lines) are the objects constituting the system. The rectangles in a lifeline represent the execution of a unit of behavior or of an action, and they are called execution specification. Execution specifications may be associated with timing constraints that represent either a time value (③) or a time range (④). Finally, messages, written with horizontal arrows, display interaction. Solid arrowheads represent synchronous calls, open arrowheads represent asynchronous messages, and dashed lines represent reply messages.

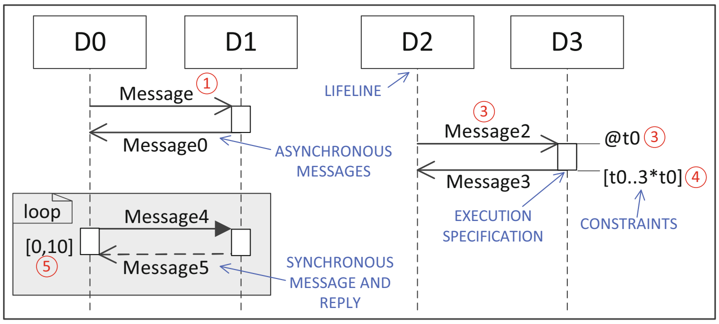


Fig. 22.4 Example of UML sequence diagram

22.4.1.4 The UNIVERCM Model of Computation (MoC)

UNIVERCM is an automaton-based MoC that unifies the modeling of both the analog (i.e., continuous) and the digital (i.e., discrete) domains, as well as hardware-dependent SW. A formal and complete definition is available in [44, 56].

In each UNIVERCM automaton (depicted in Fig. 22.5), states reproduce the continuous-time dynamics of hybrid automata, while transitions reproduce the discrete-time semantics of EFSMs. As a consequence, UNIVERCM automata can be reduced to either EFSMs or hybrid automata, depending on the enabled features. An automaton can be transformed in an *equivalent EFSM* by transforming its continuous time features into discrete transitions (e.g., by discretizing the flow predicate). A UNIVERCM automaton can also be transformed into an equivalent hybrid automaton by reducing discrete transitions to conditions that allow to change state and by moving the corresponding actions to the flow predicate of the destination state. This correspondence of UNIVERCM automata to well-established formal models allows to apply well-known design or verification techniques, originally defined for EFSMs or hybrid automata, also to the context of UNIVERCM. This makes UNIVERCM an important resource in the design of HW/SW systems, as it covers the heterogeneity of ES, ranging from analogue and digital HW to dedicated SW, in order to build reuse and redesign flows [44].

Note that UNIVERCM states and transitions are provided with two additional tags, i.e., atomic and priority. The *atomic* tag is used to define atomic regions that are considered to all intent a single transition when performing parallel composition with other automata. The *priority* tag is used to handle nondeterministic behaviors that may be present in a system: in case two or more transitions can be performed at the same time, the automaton activates the one with lower value of the priority tag.

UNIVERCM variables fall back in three main classes: discrete variables, wire variables, and continuous variables. Wire variables extend discrete variables with an event that is activated whenever the corresponding variable changes value. This mechanism is used to mimic the event-driven semantics of Hardware Description Languages (HDLs). Continuous variables are dense variables that can be either assigned an explicit value (e.g., in discrete transitions) or constrained through the flow predicate. They thus resemble the clock variables of timed automata [6].

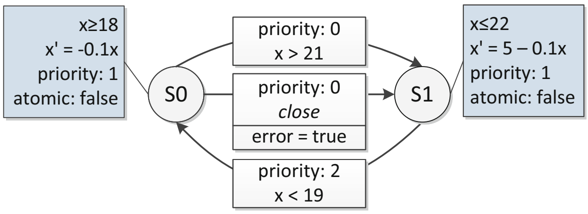


Fig. 22.5 Example of UNIVERCM automaton

22.4.2 Top-Down and Bottom-Up Flows for System Verification

UNIVERCM bridges the characteristics of the major automata-based formalisms, and it thus allows to reduce both top-down and bottom-up flows to a single framework, to build a homogeneous simulatable description of the overall ES. Such a description can then be the focus of redesign and validation flows, targeting the homogeneous simulation of the overall system [43]. This section presents the main flows that can be reduced to UNIVERCM, as summarized in Fig. 22.6.

22.4.2.1 Bottom-Up: Mapping Digital HW to UNIVERCM

The mapping of digital HW to UNIVERCM can be defined focusing on the semantics of HDLs, i.e., of the languages used for reproducing and designing HW execution. In HDLs, digital HW is designed as a number of concurrent processes that are activated by events and by variations of input signals, constituting the process sensitivity list. Process activation is managed by an internal scheduler that repeatedly builds the queue of runnable processes and advances simulation time.

When mapping to UNIVERCM, each HDL process is mapped to an automaton, whose transitions are activated by variations in the value of the signals in the sensitivity list. In the example in Fig. 22.7, the automaton is activated by events on the read signal `b` that fires the transaction from state `H0` to state `H1`. This mechanism, together with the sharing of variables and signals, ensures that process communication and interaction are correctly preserved. Note that, since digital HW does not foresee continuous-time evolution, the automaton is restricted to the discrete-time dynamics (i.e., to an EFSM). The existence of predefined synchronization points (e.g., wait primitives) is ensured in UNIVERCM with an ad hoc predicate, called *atomic*, that allows to consider a number of transitions and states as a single transition (e.g., transitions from `H0` to `H4` in Fig. 22.7). This guarantees that the original execution flow is preserved.

The mapping of the sensitivity list to events, and the parallel semantics of UNIVERCM automata, builds an automatic scheduling mechanism that avoids the need for an explicit scheduling routine. The advancement of simulation time is explicitly modeled with an additional automaton [43].

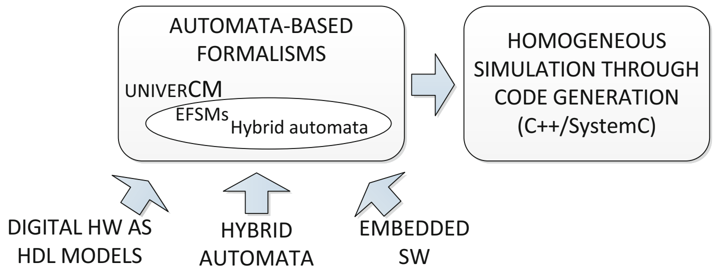

AUTOMATA-BASED FORMALISMS
UNIVERCM
EFSMs
Hybrid automata
HOMOGENEOUS SIMULATION THROUGH CODE GENERATION (C++/SystemC)
DIGITAL HW AS HDL MODELS
HYBRID AUTOMATA
EMBEDDED SW

Fig. 22.6 Top-down and bottom-up flows proposed in the following of this section

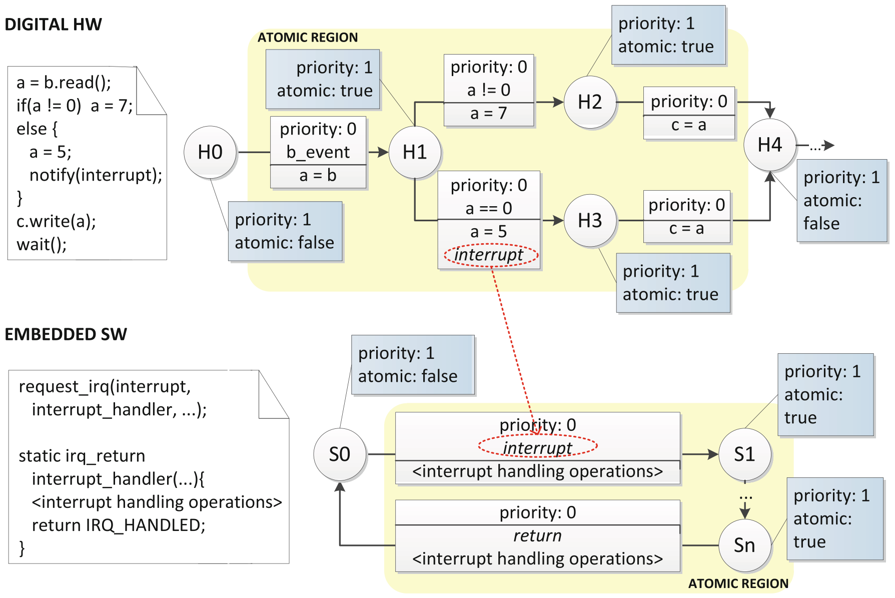


Fig. 22.7 Mapping of a digital HW component and of the communicating embedded SW to UNIVERCM automata

22.4.2.2 Bottom-Up: Mapping Embedded SW to UNIVERCM

Embedded SW is typically structured into a number of functions that can be easily represented as UNIVERCM automata evolving among a certain set of states via transitions. Since SW does not allow continuous evolution, each automaton is restricted to its discrete-time dynamics (i.e., to an EFSM).

Each function is provided with an activation event (for representing function invocation, event *interrupt* in Fig. 22.7) and a return label, which is used to communicate to the caller that the function has finished its execution (event *return* in Fig. 22.7).

Note that the atomic predicate can be used also in case of SW to avoid race conditions and unpredictable behaviors due to concurrency, e.g., in Fig. 22.7 all transitions are encapsulated in an atomic region, to guarantee that the execution of function `interrupt_handler` is non-interruptible.

Communication with HW devices based on the Memory-Mapped I/O (MMIO) approach is easily implemented in UNIVERCM by representing MMIO locations as variables shared with the HW automata. HW interrupts are mapped to synchronization events. The interrupt handling routine is mapped to an automaton, just like any other function. The activation event of the automaton is the interrupt fired by the HW device. The automaton remains suspended until it receives the interrupt event and, on receipt of the event, it executes the necessary interrupt handling operations. In the example of Fig. 22.7, the activation event of the function (event *interrupt*) is the interrupt fired by the HW automaton (in the transition from `H1` to `H3`, as highlighted by the red arrow).

22.4.2.3 Bottom-Up: Mapping Hybrid Automata to UNIVERCM

Since UNIVERCM is a superset of hybrid automata, this mapping is quite straightforward. Care must be taken in the mapping of synchronization events, as hybrid automata may activate an event only if all its recipients may perform a transition in response to the event. To reproduce this semantics in UNIVERCM, each recipient automaton is provided with a flag variable that is false by default and that is set to true only if the current state has an outgoing transition fired by the synchronization event. The event may be fired only after checking that all the corresponding flag variables are true.

Hybrid automata may be hierarchical, for simplifying the design of analog components. Mapping a hierarchical hybrid automaton to UNIVERCM requires to remove the hierarchy by recursively flattening the description.

22.4.2.4 Top-Down: Mapping UML Diagrams to UNIVERCM

The mapping of UML diagrams is defined to EFSMs, that are the reference automata-based model for MDD approaches. Given that EFSMs can be considered the discrete-time subset of UNIVERCM, this is equivalent to mapping the diagrams to UNIVERCM automata. For this reason, in the following, the terms EFSM and UNIVERCM automaton are interchangeable.

The mapping is defined for *UML sequence diagrams*. These are the most common diagrams to specify system functionality, communication, and timing constraints. However, a similar approach can be applied also to other classes of UML diagrams.

The mapping of sequence diagrams to UNIVERCM is exemplified in Fig. 22.8. Each diagram is turned into one automaton, whose states are defined one per lifeline (*D0*, *D1*, *D2*, and *D3*) [45]. Message receipt forces transition from one state to the next (e.g., from *D0* to *D1* at ①). Messages are enumerated, to define enabling conditions that preserve the execution order imposed by the sequence diagram (②). This allows to reproduce also control constructs, such as the loop construct in Fig. 22.8, that iterates a message transfer ten times (⑤). Timing constraints are used to perform check constraints through an additional state *W* (③ and ④) that may raise timing errors.

22.4.2.5 Top-Down: Mapping UNIVERCM Automata to C++/SystemC

UNIVERCM has been specifically defined to ease the conversion of UNIVERCM automata toward C++ and SystemC descriptions [43, 44].

Each UNIVERCM automaton is mapped in a straightforward manner to a C++ function containing 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. When an edge is traversed, variables are updated according to the update functions, and the continuous flow predicates and synchronization events are raised, where required.

The 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.

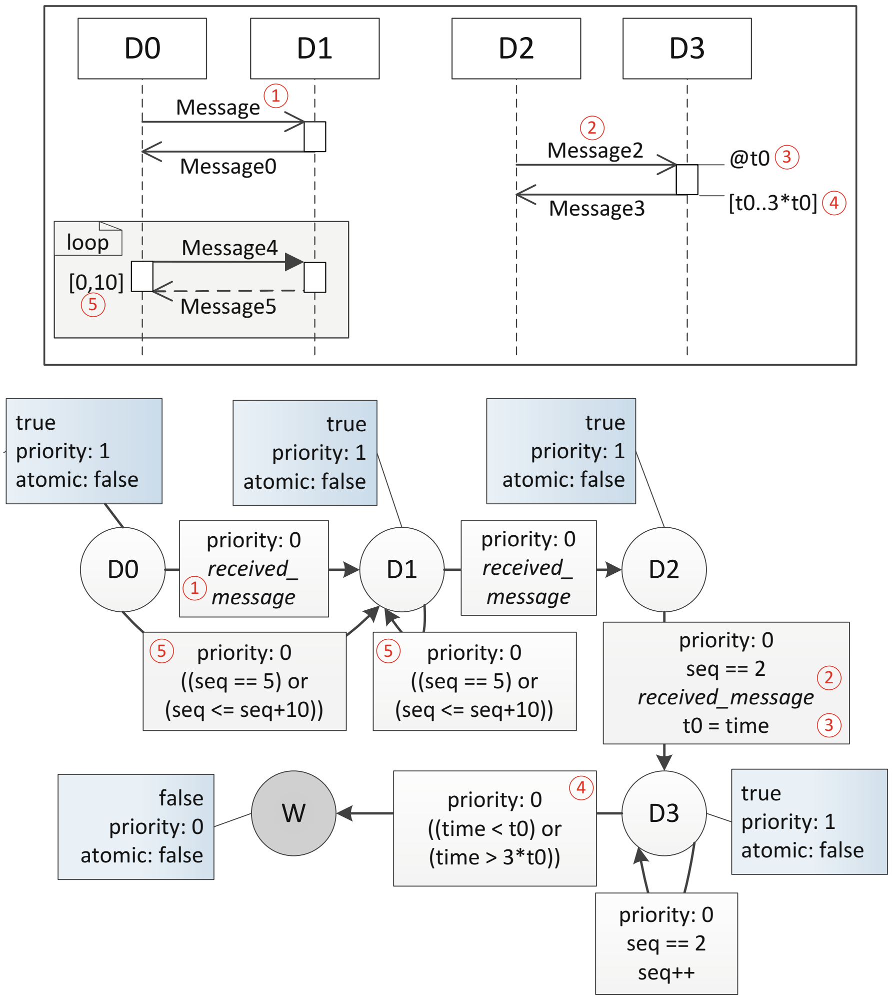


Fig. 22.8 Mapping of a UML sequence diagram to an EFSM (and thus to the discrete-time restriction of UNIVERCM)

In case of conversion toward SystemC descriptions, the presence of a simulation kernel allows to delegate some management tasks and to reproduce automata behavior through native SystemC constructs. Thus, UNIVERCM automata are 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.

22.5 Assertion Definition and Checker Generation

In software verification, software designers widely use *executable assertions* [59] for specifying conditions that apply to some states of a computation, e.g., "pre-conditions" and "post-conditions" of a procedural code block. A runtime error is released whenever execution reaches the location at which the executable assertion occurs and the related condition does not hold any more. This kind of executable assertions is limited to Boolean expressions, which are totally unaware of temporal aspects. However, if the designers aim to check more complex requirements in which Boolean expressions are used for defining relations spanning over the time, they have to (i) define assertions in a formal language and (ii) synthesize them as executable modules, i.e., *checkers*. Checkers, integrated into the simulation environment, monitor the software execution for identifying violations of the intended requirement.

In hardware verification, several solutions have already been proposed. These approaches can be classified in (i) library based or (ii) language based.

Library-based approaches rely on libraries of predefined checkers, e.g., Open Verification Library (OVL) [52], which can be instantiated into the simulation environment for simplifying the checking of specific temporal behaviors. Unfortunately, due to their inflexibility for checking general situations, the predefined checkers limit the completeness of the verification.

Language-based approaches, instead, use declarative languages, such as PSL [64] and System Verilog Assertions (SVA) [94], for formalizing the temporal behaviors into well-defined mathematical formulas (i.e., assertions) that can be synthesized into executable checkers by using automatic tools named checker generators [2, 16, 17, 34]. These tools may generate checkers implementations at different levels of abstraction, from the Register Transfer Level (RTL), e.g., MBAC [17] and FoCs [2], to the C-based Electronic System Level (ESL) [33], e.g., FoCs.

However, a large part of an ES is software, which must also be verified. Some attempts have been tried to extend hardware ABV to embedded software and a comprehensive work is in [38].

In [26], the authors present a Microsoft's proprietary approach for binding C language with PSL. They define a subset of PSL and use a simulator as an execution platform. In this case, only a relatively small set of temporal assertions can be defined, since only the equality operator is supported for Boolean expressions, and the simulator limits the type of embedded software applications.

Another extension of PSL is proposed in [101], where the authors unify assertion definition for hardware and software by translating their semantics to a common formal semantic basis. In [100], the authors use temporal expressions of the *e* hardware verification language to define checkers. In both these cases, temporal expressions are similar but not compatible with PSL standards.

Finally, in [74] the authors propose two approaches based on SystemC checkers. In the first case, embedded software is executed on top of an emulated SystemC processor, and, at every clock cycle, the checkers monitor the variables and functions stored in the memory model. In the second approach, embedded software is translated to SystemC modules which run against checkers. In this case, timing reference is imposed by introducing an event notified after each statement, and the SystemC process is suspended on additional `wait()` statements. In both cases, there are several limitations. First, the approaches are not general enough to support real-life embedded software: the SystemC processor cannot reasonably emulate real embedded system processors. In addition, the translation of embedded software applications in SystemC may be not flexible enough. Secondly, in both cases the SystemC cosimulation and the chosen timing references introduce significant overhead. In particular, clock cycle or statement step may be an excessively fine granularity for efficiently evaluating a sufficient number of temporal assertions. Moreover, on the one hand, defining assertions which consider absolute time may generate significantly large checkers to address the high number of intermediate steps; on the other, it is difficult to define temporal assertions at source code level, i.e., C applications, in terms of clock cycles.

22.5.1 Template-Based Assertion Design

Previous sections show that assertion definitions can be unified for both hardware and software, that is, they can be applied to an entire ES. An effective tool in this case is DDPSL [41]: a template library and a tool which simplify the definition of formal properties. It combines the advantages of both PSL and OVL, i.e., expressiveness and simplicity.

The template library is composed of *DDTemplates*, i.e., PSL-based templates accompanied with an *interface semantics*. The adoption of a PSL-based property implementation guarantees the same expressiveness of LTL and CTL temporal logics, a wide compatibility with HDL (e.g., VHDL, Verilog and SystemVerilog, SystemC) and programming languages (e.g., C++), and enables a large reuse of already available verification tools previously described. Moreover, like the OVL approach, the use of an interface semantics allows a clean separation between property implementation and property semantics. Such an interface notably simplifies property definition: the user needs only to understand the semantics of the interface and replace parameters with the intended expression, and a correct-by-construction PSL code is automatically generated.

DDTemplates are characterized by (i) a parametric interface, (ii) a formal PSL implementation, and (iii) a detailed semantics (i.e., interface semantics) that specifies how to use the corresponding interface for defining properties.

The interface consists of a synthetic description that gives an intuitive idea of the semantics of the property. Such a description is characterized by parameters that are placeholders inside the property. These parameters are strongly typed and distinguished into Boolean, arithmetic, temporal, and Sequential Extended Regular Expression (SERE) parameters. The interaction with such placeholders guides property definition: the user can replace parameters only with compatible elements according to a predefined semantics check.

Figure 22.9 shows an example of the "conditional events bounded by time" DDTemplate. In particular, it reports the synthetic description adopted as interface showing the three parameters (i.e., $\$P$, $\$Q$, and $\$i$) that the user can substitute.

Although the interface is explanatory, major details of the interface semantics are described by means of online documentation provided by the DDEditor. Such documentation reports information related to the parameters type and their meaning, the temporal behavior the template aims to check and possible warnings. For example, the online documentation corresponding to the DDTemplate shown in Fig. 22.9 reports:

- *Semantics of the parameters*
  - **$P** is a *Boolean expression* that represents a configuration, an event or an input/output for the system/program.
  - **$i** is an *integer* that specifies the instant in the future within which **$Q** must hold.
  - **$Q** is a *Boolean expression* that represents a new configuration, an event or an input/output for the system/program.
- *Semantics of the template*
  - The template specifies that if the system takes the configuration $\$P$ (or the event $\$P$ happens) in the cycle t_0, then the new configuration or the event $\$Q$ must occur within the cycle $t_0 + \$i$.
- *Warnings*
  - Notice that $\$Q$ may occur in many cycles within $t_0 + \$i$, but it is not possible that $\$Q$ never happens within the cycle $t_0 + \$i$.

The PSL implementation, instead, consists of a formal PSL definition (Fig. 22.10).

More than 60 templates have been defined and organized into five libraries (e.g., a selection of templates is reported in Table 22.1); each one focuses on a specific category of patterns: universality, existence, absence, responsiveness, and precedence. The *universality* library describes behaviors that must hold continuously during the software execution (e.g., a condition that must be preserved for the whole execution, or that has to hold continuously after that the software reaches a particular configuration). The *existence* library describes behaviors in which the occurrence of particular conditions is mandatory for the software execution (e.g., a condition must be observed at least once during the whole

Every time that **$P** then, within **$i** cycle(s), **$Q**

Fig. 22.9 Example of a DDTemplate interface

Fig. 22.10 Example of formal PSL definition

always (**%P** → next_e[0..**%i**](**%Q**))

Table 22.1 Selection of assertion templates

Library	Parametric interface	Parametric PSL definition
Universality	P holds continuously after Q	`always ($Q -> next (always $P))`
	P holds continuously since Q up to R	`always (($Q & !$R) -> ($P until_ $R))`
Existence	P holds at least once since Q	`next_event!($Q)(eventually! $P)`
	P holds at least once since Q up to R	`always (($Q & !$R) -> (!$R until! $P))`
Absence	P is continuously false after Q until R	`always (($Q & !$R) -> next (!$P until! $R))`
	P is continually false before R	`!$P until! $R`
Responsiveness	P causes S to happen	`always ($P -> eventually! ($S))`
	P causes S to happen, but after Q	`always ($P -> (($Q before! $S) & eventually! ($S)))`
Precedence	P precedes R globally	`always ($P -> ($P before $R))`
	P precedes R before S	`always ($P ->(($P before $R)&($R before! $S)& eventually! ($S)))`

execution or after that a particular configuration is reached, etc.). The *absence* library describes behaviors that must not occur during the software execution or under certain conditions. The *responsiveness* library, instead, describes behaviors that specify cause-effect relations (e.g., a particular condition implies a particular configuration of the software variables). Finally, the *precedence* library describes behaviors that require a precise ordering between conditions during the software execution (e.g., a variable has to assume specific values in an exact order).

Notice that the user can ignore the exact PSL formalization. He/she can define properties by simply dragging and dropping expressions onto placeholders contained into the interface by exploiting the DDEditor tool [41].

22.6 Mutant-Based Quality Evaluation

Assertion-based verification can hypothetically provide an exhaustive answer to the problem of design correctness, but from the practical point of view, this is possible only if (1) the DUV is stimulated with testbenches that generate the set of all possible input stimuli and (2) a complete set of formal properties is defined that totally captures the designer's intents. Unfortunately, these conditions represent two of the most challenging aspects of dynamic verification, since the set of input stimuli for sequential circuits is generally infinite, and the answer to the question "have I written enough properties?" is generally based on the expertise of the verification engineers. For these reasons, several metrics and approaches have been defined to address the **functional qualification** of dynamic verification methodologies and frameworks, i.e., the evaluation of the effectiveness of testbenches and properties adopted to check the correctness of a design through semiformal simulation-based techniques.

Among existing approaches, mutation analysis and mutation testing [37], originally adopted in the field of software testing, have definitely gained consensus, during the last decades, as being important techniques for the functional qualification of complex systems both in their software [61] and hardware [15] components.

Mutation analysis [86] relies on the perturbation of the DUV by introducing syntactically correct functional changes that affect the DUV statements in small ways. As a consequence, many versions of the model are created, each containing one mutation and representing a *mutant* of the original DUV. The purpose of such mutants consists in perturbing the behavior of the DUV to see if the test suite (including testbenches and, in case, also properties) is able to detect the difference between the original model and the mutated versions. When the effect of a mutant is not observed on the outputs of the DUV, it is said to be undetected. The presence of undetected mutants points out inadequacies either in the testbenches, which, for example, are not able to effectively activate and propagate the effect of the mutant, or in the DUV model, which could include redundant code that can never be executed. Thus, mutation analysis has been primarily used for helping the verification engineers in developing effective testbenches to activate all DUV behaviors and discovering design errors. More recently, it has been used also to measure the quality of formal properties that are defined in the context of assertion-based verification. The next sections will summarize some of the most recent approaches based on mutation analysis for the functional qualification of testbenches and properties.

22.6.1 Testbench Qualification

Nowadays, (i) the close integration between HW and SW parts in modern embedded systems, (ii) the development of high-level languages suited for modeling both HW and SW (like SystemC with the Transaction-Level Model (TLM) library), and (iii) the need of developing verification strategies to be applied early in the design flow require the definition of simulation frameworks that work at the system level. Consequently, mutation analysis-based strategies for the qualification of testbenches need to be defined at system level too, possibly before HW and SW functionalities are partitioned. In this context, the mutation analysis techniques proposed for over 30 years in the SW testing community can be reused for perturbing the internal functionality of the DUV, which is indeed implemented like a SW program, often by means of C/C++ behavioral algorithms. In particular, several approaches [5, 12, 13, 20, 88, 89], empirical studies [75], and frameworks [19, 31, 36, 76] have been presented in the literature for mutation analysis of SW program. Different aspects concerning software implementation are analyzed in all these works, in which the approaches are mainly suited for perturbing Java or C constructs. However, all these proposals are suited to target basic constructs and low-level synchronization primitives rather than high-level primitives typically used for modeling TLM communication protocols. Other approaches present mutation operators targeting formal abstract models, independently from specific programming languages [9, 12, 88, 89].

These approaches are valuable to be applied at TLM levels. However, the authors do not show a strict relation between the modeled mutants and the typical design errors introduced during the modeling steps. To overcome this issue, a native TLM mutation model to evaluate the quality of TLM testbenches has been proposed in [14, 15]. It exploits traditional SW testing framework for perturbing the DUV functional part, but it presents a new mutation model for addressing the system-level communication protocol typical of TLM descriptions. The approach is summarized in the rest of this section.

22.6.1.1 Mutant-Based Qualification of TLM Testbenches

The approach assumes that the functionality of the TLM model is a procedural style of code in one or more SystemC processes. Therefore, the mutation model for the functionality is derived from the work in [36] that defined mutation operators for the C language. Selective mutation (suggested in [78] and evaluated in [87]) is applied to ensure the number of mutations grows linearly with the code size. On the contrary, the communication part of the DUV is mutated by considering the effect of perturbations injected on the EFSM models representing the Open SystemC Initiative (OSCI) SystemC TLM-2.0 standard primitives adopted for implementing blocking and non-blocking transaction-level interfaces.

In TLM-2.0, communication is generally accomplished by exchanging packets containing data and control values, through a channel (e.g., a socket) between an initiator module (master) and a target module (slave). For the sake of simplicity and lack of space, we report in Fig. 22.11 the EFSMs representing the primitives and the proposed mutations of the most relevant interfaces (i.e., blocking and non-blocking interfaces).

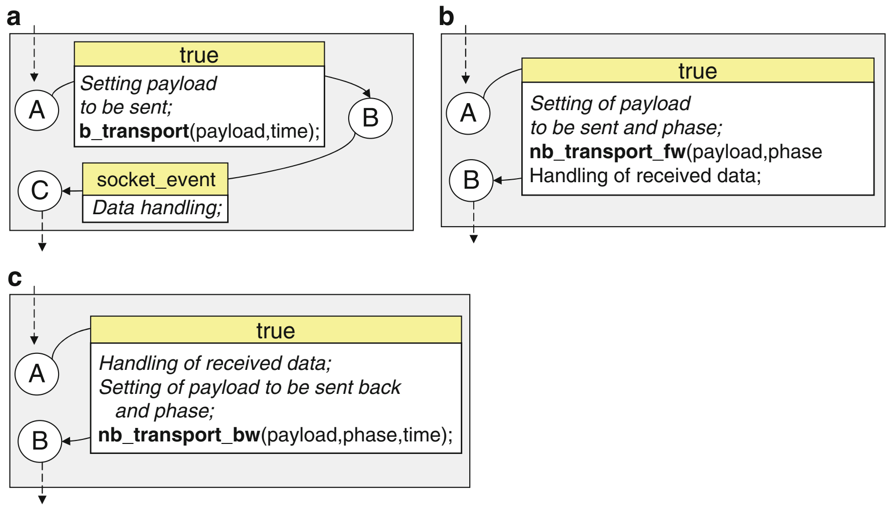

a
A
true
Setting payload
to be sent;
b_transport(payload,time);
B
C
socket_event
Data handling;
b
A
true
Setting of payload
to be sent and phase;
nb_transport_fw(payload,phase
Handling of received data;
B
c
A
true
Handling of received data;
Setting of payload to be sent back
and phase;
nb_transport_bw(payload,phase,time);
B

Fig. 22.11 EFSM models of some SystemC TLM-2.0 primitives

- *Blocking interface.* It allows a simplified coding style for models that complete a transaction in a single-function call, by exploiting the blocking primitive `b_transport()`. The EFSM model of primitive `b_transport()` is composed of three states (see Fig. 22.11a). Once the initiator has called `b_transport()`, the EFSM moves from state *A* (initial state) to state *B*, and it asks the socket channel to provide a payload packet to the target. Then, the primitive suspends in state *B* waiting for an event from the socket channel (*socket_event*) indicating that the packet can be retrieved. Finally, the retrieved data is handled by executing the operations included in the update function, moving from *B* to the final state *C*. Timing annotation is performed by exploiting the *time* parameter in the primitives and managing the time information in the handling code of the received data for implementing, for example, the *loosely timed* models.
- *Non-blocking interface.* Figure 22.11b, c show the EFSM models of the non-blocking primitives, which are composed of two states only. Primitives such as `nb_transport_fw()` and `nb_transport_bw()` perform the required operation as soon as they are called, and they immediately reach the final state in the corresponding EFSM. The caller process is informed if the non-blocking primitive succeeded by looking at its return value. Timing annotation is still performed by exploiting the *time* parameter in the primitives, while parameter *phase* is exploited for implementing more accurate communication protocols, such as the four phases *approximately timed*.

Several TLM communication protocols can be modeled by using the TLM primitives previously described, and their EFSM models can be represented by sequentially composing the EFSMs of the involved primitives. Starting from the EFSM models, the mutation model for the communication protocols is defined by following the next steps:

1. Identify a set of design errors typically introduced during the design of TLM communication protocols.
2. Identify a fault model to introduce faults (i.e., mutations) in the EFSM representations of the TLM-2.0 primitives.
3. Identify the subset of faults corresponding to the design errors identified at step 1.
4. Define mutant versions of the TLM-2.0 communication primitives implementing the faults identified at step 3.

Based on the expertise gained about typical errors made by designers during the creation of a TLM description, the following classes of design errors have been identified:

1. Deadlock in the communication phase
2. Forgetting to use communication primitives (e.g., the TLM communication primitive `nb_transport_bw()` for completing transaction phases, before initiating a new transaction is not called)

3. Misapplication of TLM operations (e.g., setting a *write* command for reading data instead of *read*)
4. Misapplication of blocking/non-blocking primitives
5. Misapplication of timed/untimed primitives
6. Erroneous handling of the generic payload (e.g., failing to set or read the packet fields)
7. Erroneous polling mechanism (e.g., infinite loop)

Other design errors could be added to the previous list to expand the proposed mutation model without altering the methodology. Each of the previous error classes has been associated with at least a mutation of the EFSM models representing TLM primitives, as described in the next paragraphs.

According to the classification of errors that may affect the specification of finite-state machine, proposed by Chow [32], different fault models have been defined for perturbing FSMs [25, 89]. They target, generally, Boolean functions labeling the transitions and/or transition's destination states. Mutated versions of an EFSM can be generated in a similar way, by modifying the behavior of enabling and update functions and/or changing the destination state of transitions.

Hereafter, we present an example of how the EFSM of Fig. 22.11a can be perturbed to generate mutant versions of the TLM primitive according to the design errors previously summarized. Figure 22.12 shows how such kinds of mutations are used to affect the behavior of primitive `b_transport()`. Numbers reported in the bottom right part of each EFSM identify the kind of design errors modeled by the mutation with respect to the previous classification.

Mutations on destination states. Changing the destination state of a transition allows us to model design errors #2, #4, and #7. For example, let us consider Fig. 22.12. Cases (a)–(d) show mutated versions of the EFSM that affect the destination state of the transition. Mutation (a) models the fact that the designer forgets to call `b_transport()` (design error #2), while (b) models the misapplication of a non-blocking primitive instead of a blocking one, since the wait on channel event is bypassed (design error #4). Cases (c) and (d) model two different incorrect uses of the polling mechanism (design error #7).

Mutations on enabling functions. Mutation on the truth value of enabling functions model design errors of type 1 and 4. For example, Fig. 22.12e shows a mutated version of the EFSM corresponding to primitive `b_transport()`, where the transition from *A* to *B* is never fired and *B* is never reached. Such a mutation corresponds to a deadlock in the communication protocol (design error #1), for example, due to a wrong synchronization among modules with the socket channel. The primitive can also be mutated as shown in case (f), which corresponds to using a non-blocking instead of a blocking primitive, since the wait in `B` for the channel event is prevented by an always true enabling function (design error #4).

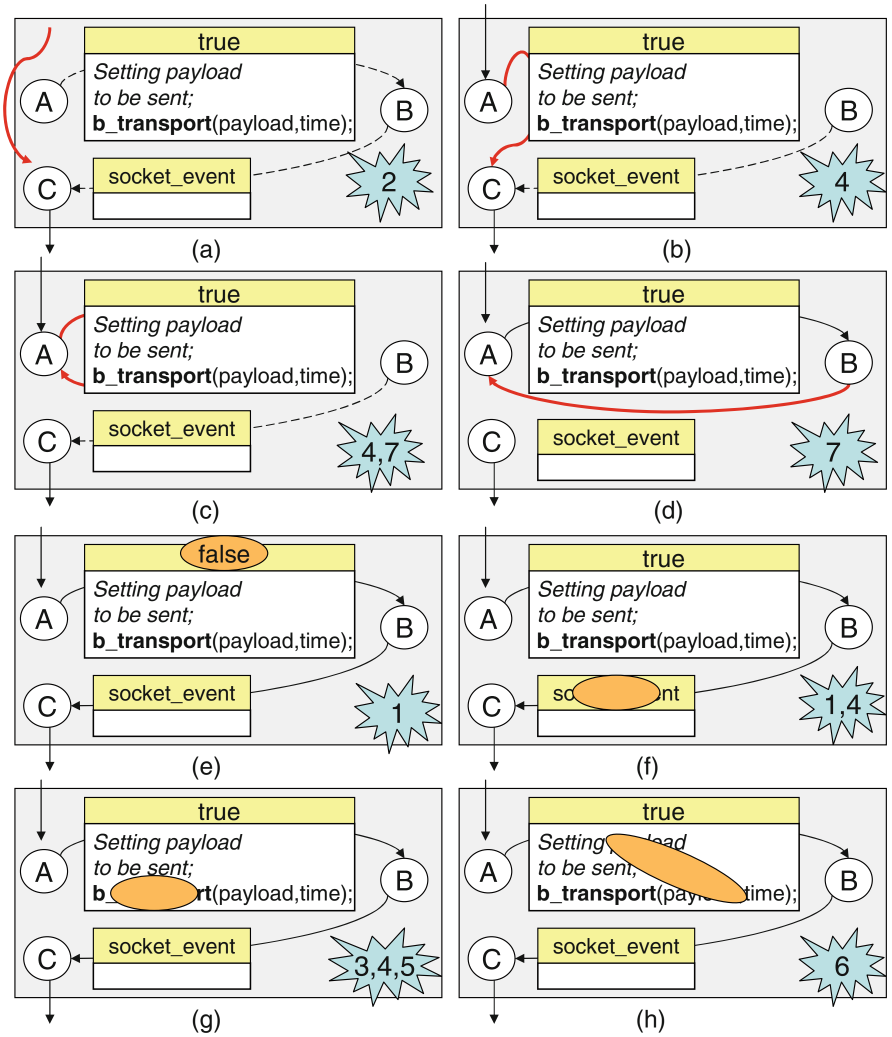


Fig. 22.12 Mutations on EFSM representing the TLM-2.0 primitive b_transport()

Mutations on update functions. Changing the operations performed in the update functions allows us to model design errors #3, #4, #5, and #6. Mutation on operations (shown in case (g)) corresponds to a misapplication of the communication primitives, like, for example, calling a transaction for writing instead of a transaction for reading (design error #3), a `b_transport()` instead of an `nb_transport()` (design error #4), setting the time parameter instead of not setting it (design error #5). On the other hand, mutations on data included in the payload packets (shown in cases (h)) model design errors corresponding to an erroneous handling of the payload packet (design error #6).

22.6.2 Property Qualification

In the last years, research topics investigated how to assess the quality and the comprehensiveness of a set of properties to increase the efficiency and effectiveness of assertion-based verification. Three approaches have emerged:

1. Detection of properties that *pass vacuously*. Properties are vacuously satisfied if they hold in a model and can be strengthened without causing them to fail. Such properties may not cover any behavior of the golden model thus they can lead to a false sense of safety. The vacuous satisfaction points out problems either in property or in environment definition or in the model implementation.
2. Analysis of the *completeness* of a set of properties. A set of properties could be incomplete since some requirements could be only partially formalized into properties. As a consequence behaviors uncovered by properties can exist, so implementation could be wrong even if it satisfies all the defined properties.
3. Identification of *over-specification*. The set of properties could be over-specified if it contains properties that can be derived as logical consequence of other properties. For example, it is possible to define a property whose coverage is a subset of the coverage of another defined property. Thus, all behaviors modeled by the first property are also modeled by the latter. The presence of such over-specification yields the verification time to be longer than it is required to be.

Current approaches to *vacuity analysis*, rely on the pioneering work of Beer et al. [10], where a formula φ is said to pass vacuously in a model M if it passes in M, and there is a sub-formula ψ of φ that can be changed arbitrarily without affecting the passing of φ. All of them, generally, exploit formal methods to search for an *interesting witness*, proving that a formula does not pass vacuously [7, 10, 11, 69, 70]. In this context, an interesting witness is a path showing one instance of the truth of the formula φ, on which every important sub-formula affects the truth of φ. Such approaches are, generally, as complex as model checking, and they require to define and model check further properties obtained from the original ones by substituting their sub-formulas in some way, thus sensibly increasing the verification time.

The analysis of the *completeness* of a set of properties addresses the question of whether enough properties have been defined. This is generally evaluated by computing *property coverage*, whose intent is to measure the percentage of DUV behaviors captured by properties. Current approaches for property coverage can be divided into two categories: mutant based [27–29, 60, 65, 71, 73, 103] and implementation based [66, 82, 102]. The first references rely on a form of mutation coverage that requires perturbing the design implementation before evaluating the property coverage. In particular, [71] gives a good theoretic background with respect to mutation of both specification and design. The latter ones estimate the property coverage by analyzing the original implementation without the need to insert perturbations. The main problem of these approaches is due to the adoption of symbolic algorithms that suffer from the state explosion problem.

Finally, regarding the problem of over-specification, the analysis can be performed by exploiting theorem proving. However, its complexity is exponential in the worst case, and it is not completely automatized, since human interaction is very often required to guide the proof. Fully automatic techniques for dealing with *over-specification removal* have not been investigated in literature, while the problem has been only partially addressed in [30]. The authors underline that given a set of properties, there can be more than one over-specified formula, and they can be mutually dependent; thus, they cannot be removed together. The authors show that finding the minimal set of properties that does not contain over-specifications is a computationally hard problem.

In general, by observing the state of the art in the literature, it appears that most of the existing strategies for property qualification rely on formal methods, which require a huge amount of spatial (memory) and temporal (time) resources, and they generally solve the qualification problem only for specific subsets of temporal logics. Given the previous drawbacks, the next section is devoted to summarizing an alternative qualification approach [39], which exploits mutation analysis and simulation to evaluate the quality of a set of formal properties with respect to vacuity [42], completeness [48], and over-specification [21].

22.6.2.1 Mutant-Based Property Qualification

As shown on top of Fig. 22.13, the basic ingredients for the mutant-based property qualification methodology are the model of the DUV, a corresponding set of formal properties that hold on the model and, if necessary, the description of the environment where the DUV is embedded in. The approach is independent from the abstraction level of the DUV and the logics used for the definition of the properties. Properties are then converted into checkers, i.e., monitors connected to the DUV that allow checking the satisfiability of the corresponding properties by simulation. Checkers can be easily generated by adopting automatic tools, like, for example, IBM FoCs [2] or radCHECK [38].

According to the central part of Fig. 22.13, mutants are injected into either the DUV or the checkers to generate their corresponding faulty versions. Faulty checker implementations are generated for addressing vacuity analysis, while faulty DUV implementations are necessary for measuring property coverage and detecting cases of over-specification.

Entering in the details, the **vacuity analysis** for a property φ is carried on in relation to the effect of mutants that affect the sub-formulas of φ. In particular, the methodology works as follows:

1. Given a set of properties that are satisfied by the DUV, a set of *interesting mutants* is injected in the corresponding checkers. Intuitively, an interesting mutant perturbs the checker's behavior similar to what happens when a sub-formula ψ is substituted by **true** or **false** in φ according to the vacuity analysis approach proposed in [11]. Thus, for each minimal sub-formula (i.e., each atomic proposition) ψ of φ, a mutant is injected in the corresponding checker, such that the signal storing the value of ψ is stuck at true or stuck at false, respectively,

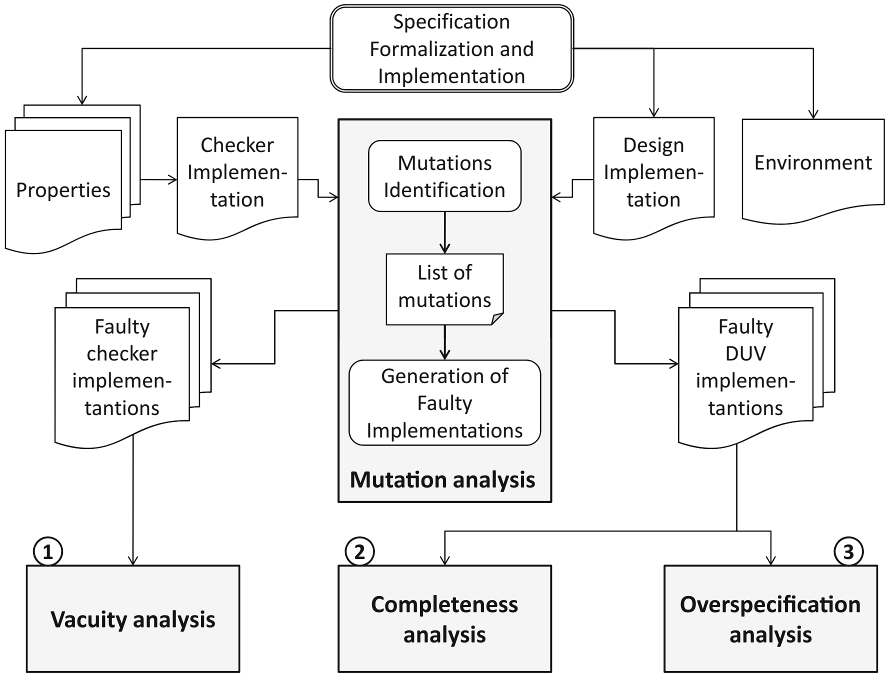

Specification Formalization and Implementation
Properties
Checker Implementation
Mutations Identification
List of mutations
Generation of Faulty Implementations
Mutation analysis
Design Implementation
Environment
Faulty checker implementantions
Faulty DUV implementantions
1
Vacuity analysis
2
Completeness analysis
3
Overspecification analysis

Fig. 22.13 Overview of the property qualification methodology

when ψ has a negative or a positive polarity. (Intuitively, in a logic with polarity, a formula has positive polarity if it is preceded by an even number of not operators; otherwise, it has negative polarity).

2. The faulty checkers are connected to the DUV and the related environment. Then, testbenches are used to simulate the DUV. The vacuity analysis relies on the observation of the simulation result. A checker failure due to the effect of an interesting mutant f corresponds to proving that the sub-formula ψ perturbed by f affects the truth value of φ. Consequently, the sequence of values generated by the testbench that causes the checker failure is an interesting witness proving that φ is not vacuous with respect to its sub-formula ψ. On the contrary, a mutant that does not cause checker failures (i.e., an undetected mutant) must be analyzed to determine if either the property is vacuous with respect to the corresponding sub-formula or the vacuity alert is due to the inefficiency of the testbenches that cannot detect the mutant during the simulation. The latter happens when there exists a test sequence (i.e., an interesting witness) for detecting the mutant, but the testbenches are not able to generate it. In case of an undetected mutant, the verification engineer can manually investigate the cause of undetectability to discriminate between a vacuous property and a low-quality testbench. However, it is also possible to make the disambiguation in an automatic way, by means of a formal approach. In fact, a new property φ' can be generated from φ, where

the sub-formula ψ inside φ is substituted with either *true* or *false*, depending on the polarity of ψ, to reproduce the same effect caused by injecting the mutant f on the checker of φ. Then, the satisfiability of φ' is verified on the DUV by using a model checker. If the model checker returns a successful answer, it implies that φ is vacuous, because ψ does not affect φ; otherwise it means that φ is not vacuous while the testbench is ineffective. In this second case, the counterexample generated by the model checker represents the input sequence that must be added to the testbench to prove the non-vacuity of φ by simulation.

3. Finally, the analysis of interesting mutants that actually correspond to vacuous passes allows the verification engineer to determine the exact cause of the vacuity, which can be either an error in the property, a too strict environment for the DUV, or an error in the model of the DUV itself that does not implement correctly the intended specification.

At the end of the vacuity analysis, it is possible to measure the **degree of completeness** of the remaining properties on the basis of their property coverage. According to the theoretical basis described in [48], property coverage is computed by measuring the capability of a set of properties to detect mutants that perturb the DUV. A low property coverage is then a symptom of a low degree of completeness. In particular, if a mutant that perturbs the functionality of the DUV does not affect at least one property in the considered set, it means that this set of properties is unable to distinguish between the faulty and the fault-free implementations of the DUV, and thus it is incomplete. In this case, a new property covering the fault should be added. On the contrary, the fact that at least one property fails in the presence of each mutant affecting the outputs of the DUV implementation represents a positive feedback about the quality of the property set. Nevertheless, the quality of the mutant model is the key aspect of the overall methodology. A low-quality set of mutants negatively impacts the overall methodology, such that achieving 100% property coverage provides a false sense of security when mutant injection is inadequate.

Independently from the adopted mutant model, the computation of the property coverage consists of two phases:

1. *Generation of faulty DUV implementations.* Perturbations of the design implementation are generated by automatically injecting mutants inside the DUV model. The obtained mutant list must include only *detectable mutants*, which are mutants that, for at least one input sequence, cause at least one output of the faulty implementation to differ from the corresponding output of the fault-free implementation. Only detectable mutants are considered to achieve an accurate estimation of the golden model completeness because undetectable mutants cannot cause failures on the properties since they do not perturb the outputs of the DUV. The set of detectable mutants can be identified by simulating the DUV with either manual testbenches or by using an automatic test pattern generator.
2. *Property coverage analysis.* The presence of a detectable mutant implies that the behavior of the faulty implementation differs from the behavior of the fault-free

implementation. Thus, while the defined properties are satisfied by the fault-free implementation, at least one of them should be falsified if checked on a faulty implementation. The property coverage is then measured by the following formula:

$$PC = \frac{\text{number of mutants detected by the properties}}{\text{number of detectable mutants}}$$

The computation of PC can be done by using a formal approach, i.e., by model checking the properties in the presence of each mutant, or by means of a dynamic strategy, i.e., by simulating the faulty DUV connected to the checkers corresponding to the properties under analysis. The formal approach is generally unmanageable for a large number of mutants, but the higher scalability provided by the dynamic simulation is paid in terms of exhaustiveness. In fact, as for the case of vacuity analysis, an undetected mutant during simulation can be due to the ineffectiveness of the testbenches rather than an incompleteness of the properties. Thus, formal analysis is restricted to the few mutants that remain undetected after simulation.

3. In case of a low property coverage, the verification engineer is guided in the definition of new properties by analyzing the area of the DUV including mutants that do not affect any property till the desired degree of completeness is achieved.

When the achieved degree of completeness is satisfactory (this measure depends on the design team standards), a last process, still based on the property coverage, can be applied for capturing the case of **over-specification**, i.e., the presence of properties that can be removed from the final set because they are covering the same behaviors covered by other properties included in the same set [21].

22.7 Automatic Stimuli Generation

The main purpose of dynamic verification is increasing the confidence of designers in the ES behavior by creating stimuli and evaluating them in terms of adequacy criteria, e.g., coverage metrics. In this context, an effective stimuli generation is at the basis of a valuable functional qualification. Actual value inputs may be either automatically generated or developed by engineers as stimuli suites. Stimuli generation techniques fall into three main categories: *concrete execution*, *symbolic execution*, and *concolic execution*. Concrete execution is based on random, probabilistic, or genetic techniques [79]. It is not an exhaustive approach, but it allows to reach deep states of the system space by executing a large number of long paths. Symbolic execution [68] represents an alternative approach for overcoming concrete execution limitations, where an executable specification is simulated using symbolic variables, and a decision procedure is used to obtain concrete values for inputs. Such approaches suffer from solver limitations in handling either the complexity of formulas or data structures or the surrounding execution environment. Such limitations have been recently addressed by proposing *concolic execution* [77, 93] that mingles concrete and symbolic executions and, whenever necessary, simplifies symbolic constraints with corresponding concrete values. However, module execution is still represented as a symbolic execution tree, growing exponentially in the number of the maintained paths, states, and conditions, thus incurring in state space explosion.

Several tools on the market adopt these approaches and provide the user with automatic stimuli generation addressing coverage metrics. DART [54] combines random stimuli generation with symbolic reasoning to keep track of constraints for executed control flow paths. CUTE [93] is a variation of the DART approach addressing complex data structures and pointer arithmetic. EXE [23] and KLEE [22] are frameworks for symbolic execution, where the second symbolically executes LLVM [72] bytecode. PEX [99] is an automated structural testing generation tool for .NET code developed at Microsoft Research. [77] describes a hybrid concolic stimuli generation approach for C programs, interleaving random stimuli generation with bounded exhaustive symbolic exploration to achieve better coverage. However, it cannot selectively and concolically execute paths in a neighborhood of the corner cases.

22.7.1 EFSM-Based Stimuli Generation

This section presents an EFSM-based concolic stimuli generation approach for ES. The approach is based on an EFSM model of the ES and leads to traverse a *target transition t* (i.e., not-yet-traversed transition) by integrating concrete execution and a symbolic technique that ensures exhaustiveness along specific paths leading to the target transition t.

Algorithm 1 is a high-level description of the proposed concolic approach. It takes as inputs the EFSM model and two timeout thresholds: (i) overall timeout, i.e., the maximum execution time of the algorithm (*MaxTime*), and (ii) inactivity timeout, i.e., the maximum execution time the long-range concrete technique can spend without improving transition coverage (*InaTime*).

Algorithm 1 The EFSM-based concolic algorithm for stimuli generation

```
1  procedure EFSMStimuliGen(EFSM, MaxTime, InaTime)
   input: embedded-application model EFSM, overall timeout MaxTime, inactivity timeout
          InaTime
   output: set of stimuli Stimuli
2  Stimuli ← ∅; RInf ← ∅;
3  DInf ← DependencyAnalysis(EFSM);
4  while elapsed time < MaxTime do
5      while inactivity timeout InaTime not expired do
6          (stimulus, reach) ← LongRangeSearch(EFSM, RInf);
7          Stimuli ← Stimuli ∪ {stimulus}; RInf ← RInf ∪ {reach};
8      (stimulus, reach) ← GuidedWideWidthSearch(EFSM, RInf, DInf);
9      Stimuli ← Stimuli ∪ {stimulus}; RInf ← RInf ∪ {reach};
10 return Stimuli
```

RInf keeps track of the EFSM configurations used for (re-)storing system status whenever the algorithm switches between the symbolic and concrete techniques. At the beginning, both *Stimuli* and *RInf* are empty (line 2). The algorithm identifies the dependencies between internal and input variables and EFSM paths. *EFSM transitions* allow to determine dependency information (*DInf*, line 3), used in the following corner-case-oriented symbolic phases, when further dynamic analysis between *EFSM paths* is performed. Such a dependency analysis selectively chooses a path for symbolic execution whenever the concrete technique fails in improving transition coverage of the EFSM. The stimuli generation runs until the specified overall timeout expires (line 4–5). First, the algorithm executes a long-range concrete technique (line 6), then a symbolic wide-width technique, which exploits the Multi-Level Back Jumping (MLBJ) (see Sect. 22.7.1.3) to cover corner cases (line 7). The latter starts when the transition coverage remains steady for the user-specified inactivity timeout (line 5). The algorithm reverts back to the long-range search as soon as the wide-width search traverses a target transition. The output is the generated stimuli set (line 9). The adopted long-range search (line 6) exploits constraint-based heuristics [40] that focus on the traversal of just one transition at a time. Such approaches scale well with design size and, significantly, improve the bare pure random approach. The following sections deepen the single steps of the algorithm.

22.7.1.1 Dependency Analysis

Without a proper dependency analysis, the stimuli generation engine wastes considerable effort in the exploration of uninteresting parts of the design. Thus, the proposed approach focuses on dependencies of enabling functions, i.e., control part, on internal variables. As a further motivating example, consider the EFSM in Fig. 22.2. Let t_8 be the target transition. We want to compare paths $\pi_1 = t_2 :: t_6$ and $\pi_2 = t_3$, where $t :: t'$ denotes the concatenation of transitions t and t'. The enabling function of t_8 involves the variables "ova" and "pva." Both are defined along π_1 by means of primary inputs. Along π_2 only "pva" is defined by means of primary inputs. Thus, to traverse t_8, MLBJ will select π_1 instead of π_2, since t_8 enabling function is more likely to be satisfied by the symbolic execution of π_1 rather than of π_2. We will consider again this example at the end of the section.

We approximate dependencies as *weights*. Indirect dependencies, as the dependency of d_2 on i_1 in the sequence of assignments "$d_1 := i_1 + i_2;\ d_2 := d_1 + i_3$," are approximated as *flows* of weights between assignments. Given a target transition $\bar{t}$, we map each path ending in $\bar{t}$ to a nonnegative weight representing data control dependencies. Intuitively, the higher the weight, the greater is the dependency of $\bar{t}$'s enabling function, i.e., $e_{\bar{t}}$, on inputs read along such a path, and the higher is the likelihood that its symbolic execution leads to the satisfaction of $e_{\bar{t}}$. An initial weight is assigned to $e_{\bar{t}}$, then we let it "percolate" backward along paths. Each transition lets a fraction of the received weight percolate to the preceding nodes and retains the remaining fraction. The weight associated with a path π is defined as *the sum of weights retained by each transition of* π. The ratio of the weight retained by a transition is defined by its update function.

22.7.1.2 Snapshots of the Concrete Execution

The ability of saving the EFSM configurations allows the system to be restored during the switches between concrete and symbolic phases. This avoids the time consuming re-execution of stimuli. Algorithm 1 keeps trace of the reachability information, i.e., *RInf*, and maintains a cache of snapshots of the concrete execution. Each time a stimulus is added to the set of stimuli, the resulting configuration is stored in memory and explicitly linked to the reached state. The wide-width technique searches feasible paths that both start from an intermediate state of the execution and lead to the target transition. Moreover, during the MLBJ, for a given configuration and target transition, many paths are checked for feasibility, as described in Sect. 22.7.1.3. Thus, caching avoids the cost of recomputing configurations for each checked path. Both the time and memory requirements of each snapshot are proportional to the size of D (see definition in Sect. 22.4.1.1).

22.7.1.3 Multilevel Back Jumping

When the long-range concrete technique reaches the inactivity time-out threshold, the concolic algorithm switches to the weight-oriented symbolic approach; see line 7 in Algorithm 1. Typically, some hard-to-traverse transitions, whose enabling functions involve internal variables, prevent the concrete technique goes further in the exploration. In this case, the MLBJ technique is able to selectively address

Algorithm 2 The core of the MLBJ technique

```
1  **procedure** MLBJ($\bar{t}$, timeout)
   **input**: target transition $\bar{t} \in T$, timeout
   **output**: stimuli for $\bar{t}$, in case empty
2  Let $w^0$ be the initial weight tuple such that
3  $\forall d_i \in D \,.\, w_i^0 = \begin{cases} 1 & \text{if } d_i \text{ occurs in EnablingFunction}(\bar{t}), \\ 0 & \text{otherwise.} \end{cases}$
4  $p \leftarrow \{(\bar{t}, w^0, 0, 0)\}$;
5  **while** *elapsed time* < timeout **do**
6      $(t :: \pi, w, r', r) \leftarrow$ remove_top($p$)
7      // $w = P^*(t :: \pi, w^0)$, $r' = R^*(t :: \pi, w^0)$,
8      // $r = R^*(\pi, w^0)$
9      **if** $t :: \pi$ *is satisfiable* **then**
10         **if** $r < r'$ **then**
11             **foreach** *configuration* $\langle src(t), k \rangle$ **do**
12                 **if** $k \wedge t :: \pi$ *is satisfiable* **then**
13                     **return** stimuli for $\bar{t}$;
14         **foreach** $\{t' \in T \mid dst(t') = src(t)\}$ **do**
15             $w' = P^*(t' :: t :: \pi, w^0)$;
16             $r'' = R^*(t' :: t :: \pi, w^0)$;
17             push$\Big(p, \big(t' :: t :: \pi, w', r'', r'\big)\Big)$;
```

paths, with high dependency on inputs, i.e., high retained weight, for symbolically executing them. Such paths are leading from an intermediate state of the execution to the target transition; thus the approach is exhaustive in a neighborhood of the corner case.

Algorithm 2 presents a description of the core of the MLBJ procedure. A transition $\bar{t}$ is selected in the set of target transition, and then a progressively increasing neighborhood of $\bar{t}$ is searched for paths π leading to $\bar{t}$ and having maximal retained weight, i.e., $R^{*}(\pi, w^{0})$. If the approach fails, another target transition is selected in the set, and the procedure is repeated.

Describing in an elaborate way, a visit is started from $\bar{t}$ that proceeds backward in the EFSM graph. The visit uses a priority queue p, whose elements are paths that end in $\bar{t}$. In particular, each path of p is accompanied by its weight tuple, i.e., $w = P^{*}(\pi, w^{0})$ and retained weights, i.e., $r = R^{*}(\pi, w^{0})$ and $r' = R^{*}(t :: \pi, w^{0})$. At the beginning, the queue p contains only $\bar{t}$ and the associated initial weight w^{0} (lines 2–3); no weight is initially retained (line 4). At each iteration, a path $t :: \pi$ with maximal retained weight is removed from p (line 6). The decision procedure is used to check if the path $t :: \pi$ can be proved unsatisfiable in advance (line 9), e.g., it contains clause conflicts. In this case $t :: \pi$ is discarded so the sub-tree preceding $t{::}\pi$ will not be explored. Otherwise, if the transition t has yielded a positive retained weight (line 10), and then for each configuration associated with the source state of t, the decision procedure checks the existence of a sequence of stimuli that leads to the traversal of $t :: \pi$ and thus of $\bar{t}$ (lines 12–13). In particular, the path constraint is obtained by the identified EFSM path, i.e., $t :: \pi$, and the concrete values of the internal variables, i.e., k, (line 12). In case a valid sequence of stimuli has not been identified, for each transition t' that precedes t, the path $t' :: t :: \pi$ is added to the priority queue p (lines 14–17).

22.8 Conclusion

This chapter focused on the realization of an effective semiformal ABV environment in a Model-Driven Design Framework. First it provided a general introduction to Model-Driven Design and Assertion-Based Verification concepts and related formalisms and then a more detailed view on the main challenges concerning their combined use. Assertion-based verification can hypothetically provide an exhaustive answer to the problem of design correctness, but from the practical point of view, this is possible only if (1) the design under verification is stimulated with testbenches that generate the set of all possible input stimuli and (2) a complete set of formal properties is defined that totally captures the designer's intents. Therefore the chapter addressed assertion definition and automatic generation of checkers and stimuli.

The key ingredient for an effective design and verification framework based on MDD and ABV, is represented by the possibility of defining a model of the desired system and then automatically deriving the corresponding simulatable description to be used for virtual prototyping. This aspect was addressed in the chapter by using automata-based formalisms, together with bottom-up and top-down flows, whose combined adoption allows the generation of a homogeneous simulatable description of the overall system.

Finally, the problem of property qualification was addressed by discussing about property vacuity, completeness and over-specification.

References

1. 3S Software (2012) CoDeSys. http://www.3s-software.com
2. Abarbanel Y, Beer I, Gluhovsky L, Keidar S, Wolfsthal Y (2000) FoCs: automatic generation of simulation checkers from formal specifications. In: Proceedings of international conference on computer aided verification (CAV), pp 538–542
3. Aerospace Valley (2012) TOPCASED project. http://www.topcased.org
4. Alagar V, Periyasamy K (2011) Extended finite state machine. In: Specification of software systems, texts in computer science. Springer, London, pp 105–128. DOI 10.1007/978-0-85729-277-3_7
5. Alexander RT, Bieman JM, Ghosh S, Bixia J (2002) Mutation of Java objects. In: Proceedings of IEEE ISSRE, pp 341–351
6. Alur R, Dill DL (1994) A theory of timed automata. Theoret Comput Sci 126(2):183–235
7. Armoni R, Fix L, Flaisher A, Grumberg O, Piterman N, Tiemeyer A, Vardi M (2003) Enhanced vacuity detection in linear temporal logic (CAV). In: International conference on computer aided verification, vol 2725. Springer, Berlin/Heidelberg, pp 368–380
8. Atego (2012) ARTiSAN. http://www.atego.com/products/artisan-studio
9. Batth SS, Vieira ER, Cavalli A, Umit Uyar M (2007) Specification of timed EFSM fault models in SDL. In: Proceedings of FORTE, pp 50–65
10. Beer I, Ben-David S, Eisner U, Rodeh Y (1997) Efficient detection of vacuity in ACTL formulas. In: International conference on computer aided verification (CAV), vol 1254, pp 279–290
11. Beer I, Ben-David S, Eisner C, Rodeh Y (2001) Efficient detection of vacuity in temporal model checking. Form Methods Syst Des 18(2):141–163
12. Belli F, Budnik CJ, Wong WE (2006) Basic operations for generating behavioral mutants. In: Proceedings of IEEE ISSRE, pp 10–18
13. Black P, Okun V, Yesha Y (2000) Mutation operators for specifications. In: Proceedings of IEEE ASE, pp 81–88
14. Bombieri N, Fummi F, Guarnieri V, Pravadelli G (2014) Testbench qualification of systemc TLM protocols through mutation analysis. IEEE Trans Comput 63(5):1248–1261
15. Bombieri N, Fummi F, Pravadelli G, Hampton M, Letombe F (2009) Functional qualification of TLM verification. In: Design, automation test in Europe conference exhibition, DATE'09, pp 190–195. DOI 10.1109/DATE.2009.5090656
16. Borrione D, Liu M, Morin-Allory K, Ostier P, Fesquet L (2005) On-line assertion-based verification with proven correct monitors. In: Proceedings of international conference on information and communications technology (ICICT), pp 125–143
17. Boulé M, Zilic Z (2008) Automata-based assertion-checker synthesis of PSL properties. ACM Trans Des Autom Electron Syst 13:1–21. http://doi.acm.org/10.1145/1297666.1297670
18. Boutekkouk F, Benmohammed M, Bilavarn S, Auguin M et al (2009) UML 2.0 profiles for embedded systems and systems on a chip (SoCs). J Object Technol 8(1):135–157. DOI 10.5381/jot.2009.8.1.a1
19. Bradbury JS, Cordy JR, Dingel J (2006) ExMan: a generic and customizable framework for experimental mutation analysis. In: Proceedings of IEEE ISSRE, pp 4–9
20. Bradbury JS, Cordy JR, Dingel J (2006) Mutation operators for concurrent Java (J2SE 5.0). In: Proceedings of IEEE ISSRE, pp 11–11

21. Brait S, Fummi F, Pravadelli G (2005) On the use of a high-level fault model to analyze logical consequence of properties. In: Proceedings of ACM/IEEE international conference on formal methods and models for co-design, MEMOCODE, pp 221–230
22. Cadar C, Dunbar D, Engler D (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of USENIX symposium on operating systems design and implementation (OSDI)
23. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) EXE: a system for automatically generating inputs of death using symbolic execution. In: ACM conference on computer and communications security, pp 322–335
24. Cadence (2012) Assertion-based verification. http://www.cadence.com/products/fv/pages/abv_flow.aspx
25. Cheng KT, Jou JY (1990) A single-state-transition fault model for sequential machines. In: IEEE ICCAD'90, pp 226–229
26. Cheung P, Forin A (2007) A C-language binding for PSL. In: Proceedings of international conference on embedded software and systems (ICESS). Springer, pp 584–591
27. Chockler H, Kupferman O, Kurshan R, Vardi M (2001) A practical approach to coverage in model checking. In: Proceedings computer aided and verification, pp 66–78
28. Chockler H, Kupferman O, Vardi M (2006) Coverage metrics for formal verification. Int J Softw Tools Technol Transfer (STTT) 8:373–386
29. Chockler H, Kupferman O, Vardi M (2006) Coverage metrics for temporal logic model checking. Formal Methods Syst Des 28:189–212
30. Chockler H, Strichman O (2007) Easier and more informative vacuity checks. In: Proceedings ACM/IEEE international conference on formal methods and models for codesign, pp 189–198
31. Choi BJ, DeMillo RA, Krauser EW, Martin RJ, Mathur AP, Pan AJOH, Spafford EH (1989) The Mothra tool set (software testing). In: Proceedings of IEEE HICSS, vol 2, pp 275–284
32. Chow T (1978) Testing software design modeled by finite state machines. IEEE Trans Softw Eng 4(3):178–187
33. Dahan A, Geist D, Gluhovsky L, Pidan D, Shapir G, Wolfsthal Y, Benalycherif L, Kamidem R, Lahbib Y (2005) Combining system level modeling with assertion-based verification. In: Proceedings of international symposium on quality of electronic design (ISQED), pp 310–315
34. Das S, Mohanty R, Dasgupta P, Chakrabarti P (2006) Synthesis of system verilog assertions. In: Proceedings of design, automation & test in Europe conference & exhibition (DATE), vol 2, pp 1–6
35. De Simone R, André C (2006) Towards a "synchronous reactive" UML profile? Int J Softw Tools Technol Transfer 8(2):146–155
36. Delamaro ME, Maldonado JC (1996) Proteum – a tool for the assessment of test adequacy for C programs. In: PCS'96, pp 79–95
37. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. IEEE Comput 11(4):34–41
38. Di Guglielmo G, Di Guglielmo L, Foltinek A, Fujita M, Fummi F, Marconcini C, Pravadelli G (2013) On the integration of model-driven design and dynamic assertion-based verification for embedded software. J Syst Softw 86(8):2013–2033. DOI 10.1016/j.jss.2012.08.061
39. Di Guglielmo L, Fummi F, Pravadelli G (2009) The role of mutation analysis for property qualification. In: IEEE/ACM international conference on formal methods and models for co-design, MEMOCODE, pp 28–35
40. Di Guglielmo G, Fummi F, Pravadelli G, Soffia S, Roveri M (2010) Semi-formal functional verification by EFSM traversing via NuSMV. In: Proceedings of IEEE international high level design validation and test workshop (HLDVT), pp 58–65
41. Di Guglielmo L, Fummi F, Orlandi N, Pravadelli G (2010) DDPSL: an easy way of defining properties. In: 2010 IEEE international conference on computer design (ICCD), pp 468–473
42. Di Guglielmo L, Fummi F, Pravadelli G (2010) Vacuity analysis for property qualification by mutation of checkers. In: Design, automation test in Europe conference exhibition (DATE), pp 478–483

43. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2012) A formal support for homogeneous simulation of heterogeneous embedded systems. In: IEEE international symposium on industrial embedded systems (SIES), pp 211–219
44. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2013) UNIVERCM: the universal versatile computational model for heterogeneous system integration. IEEE Trans Comput 62(2):225–241
45. Ebeid E, Fummi F, Quaglia D (2015) HDL code generation from UML/MARTE sequence diagrams for verification and synthesis. Des Autom Embed Syst 19(3):277–299. DOI 10.1007/s10617-014-9158-1
46. Ebeid E, Fummi F, Quaglia D (2015) Model-driven design of network aspects of distributed embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 34(4):603–614
47. Ebert C, Jones C (2009) Embedded software: facts, figures, and future. Computer 42(4): 42–52
48. Fedeli A, Fummi F, Pravadelli G (2007) Properties incompleteness evaluation by functional verification. IEEE Trans Comput 56(4):528–544
49. Ferrari A, Gaviani G, Gentile G, Stara G, Romagnoli G, Thomsen T (2004) From conception to implementation: a model based design approach. In: Proceedings of IFAC symposium on advances in automotive control
50. Ferro L, Pierre L (2010) ISIS: runtime verification of TLM platforms. Adv Des Methods Model Lang Embed Syst SoCs 63:213–226
51. Foster H, Krolnik A, Lacey D (2004) Assertion-based design. Springer, New York
52. Foster H, Larsen K, Turpin M (2006) Introducing the new accellera open verification library standard. In: Proceedings of design and verification conference (DVCON)
53. Gentleware (2012) Poseidon for UML embedded edition. http://www.gentleware.com/uml-software-embedded-edition.html
54. Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In: Proceedings of ACM SIGPLAN conference on programming language, design, and implementation (PLDI), pp 213–223
55. Graaf B, Lormans M, Toetenel H (2003) Embedded software engineering: the state of the practice. IEEE Softw 20(6):61–69
56. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2011) UNIVERCM: The UNIversal VERsatile computational model for heterogeneous embedded system design. In: Proceedings of IEEE HLDVT, pp 33–40
57. HAL – Inria (2012) Gaspard2 UML profile documentation. http://hal.inria.fr/inria-00171137/en
58. Henzinger T (1996) The theory of hybrid automata. In: Logic in computer science (LICS). IEEE Computer Society, New Brunswick, pp 278–292
59. Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. In: Proceedings of IEEE international conference on dependable systems and networks (DSN), pp 24–33
60. Hoskote Y, Kam T, Ho P, Zhao X (1999) Coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE design automation conference, pp 300–305
61. Hyunsook D, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32(9): 733–752
62. IAR Systems (2012) IAR visualSTATE. http://www.iar.com/Products/IAR-visualSTATE/
63. IBM (2012) Rational Rhapsody. http://www.ibm.com/software/awdtools/rhapsody
64. IEEE Computer Society (2010) IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2010)
65. Jayakumar N, Purandare M, Somenzi F (2003) Dos and don’ts of CTL state coverage estimation. In: Proceedings of design automation conference (DAC)
66. Katz S, Grumberg O (1999) Have I written enough properties? – a method of comparison between specification and implementation. In: Proceedings ACM advanced research working conference on correct hardware design and verification methods. Springer, pp 280–297

67. Kim M, Kim Y, Kim H (2011) A comparative study of software model checkers as unit testing tools: an industrial case study. IEEE Trans Softw Eng 37(2):146–160
68. King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
69. Kupferman O, Vardi MY (1999) Vacuity detection in temporal model checking. In: Conference on correct hardware design and verification methods, pp 82–96
70. Kupferman O, Vardi M (2003) Vacuity detection in temporal model checking. Int J Softw Tools Technol Transfer 4(2):224–233
71. Kupferman O, Li W, Seshia S (2008) A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: Proceedings IEEE international conference on formal methods in computer-aided design
72. Lattner C, Adve V (2005) The LLVM compiler framework and infrastructure tutorial. In: Proceedings of international workshop on languages and compilers for high performance computing (LCPC). Springer, pp 15–16
73. Lee T, Hsiung P (2004) Mutation coverage estimation for model checking. In: Proceedings international symposium on automated technology for verification and analysis, pp 354–368
74. Lettnin D, Nalla P, Ruf J, Kropf T, Rosenstiel W, Kirsten T, Schonknecht V, Reitemeyer S (2008) Verification of temporal properties in automotive embedded software. In: Proceedings of design, automation & test in Europe conference & exhibition (DATE). ACM, pp 164–169
75. Lyu MR, Zubin H, Sze SKS, Xia C (2003) An empirical study on testing and fault tolerance for software reliability engineering. In: Proceedings of IEEE ISSRE, pp 119–130
76. Ma YS, Offutt J, Kwon YR (2005) Mujava: an automated class mutation system. Softw Test Verif Reliab 15(2):97–133
77. Majumdar R, Sen K (2007) Hybrid concolic testing. In: Proceedings of IEEE international conference on software engineering (ICSE), pp 416–426
78. Mathur AP (1991) Performance, effectiveness, and reliability issues in software testing. In: COMPSAC'91, pp 604–605
79. McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
80. Mentor Graphics (2012) Assertion-based verification . http://www.mentor.com/products/fv/methodologies/abv
81. Mischkalla F, He D, Mueller W (2010) A UML profile for SysML-based comodeling for embedded systems simulation and synthesis. In: Proceedings of workshop on model based engineering for embedded system design (MBED)
82. Mishra P, Dutt N (2002) Automatic functional test program generation for pipelined processors using model checking. In: Proceedings IEEE high-level design validation and test, pp 99–103
83. Object Management Group, Inc. (2012) MARTE resource page. http://www.omgmarte.org/
84. Object Management Group, Inc. (2012) OMG specifications. http://www.omg.org
85. Object Management Group, Inc. (2012) UML resource page. http://www.uml.org
86. Offutt AJ, Untch RH (2001) Mutation 2000: uniting the orthogonal. In: Wong WE (ed) Mutation testing for the new century. Kluwer Academic Publishers, Boston, pp 34–44
87. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: ICSE'93, pp 100–107
88. Olsson T, Runeson P (2001) System level mutation analysis applied to a state-based language. In: Proceedings of IEEE ECBS, pp 222–228
89. Pinto Ferraz Fabbri SC, Delamaro ME, Maldonado JC, Masiero PC (1994) Mutation analysis testing for finite state machines. In: IEEE ISSRE'94, pp 220–229
90. Riccobene E, Scandurra P, Bocchio S, Rosti A, Lavazza L, Mantellini L (2009) SystemC/C-based model-driven design for embedded systems. ACM Trans Embed Comput Syst 8(4):1–37
91. Seger C (2006) Integrating design and verification – from simple idea to practical system. In: Proceedings of ACM/IEEE MEMOCODE, pp 161–162
92. Selic B (2003) The pragmatics of model-driven development. IEEE Softw 20(5):19–25

93. Sen K, Agha G (2006) CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proceedings of international conference on computer aided verification (CAV). Springer, Berlin/New York, pp 419–423
94. Society IC (2009) IEEE standard for system verilog-unified hardware design, specification, and verification language (IEEE Std 1800-2009)
95. Sparx Systems (2012) Enterprise architet. http://www.sparxsystems.com.au
96. STM Products (2012) radCHECK. http://www.verificationsuite.com
97. SysML Partners (2012) SysML resource page. http://www.sysml.org
98. The MathWorks, Inc. (2012) Simulink. http://www.mathworks.com/products/simulink/
99. Tillmann N, De Halleux J (2008) Pex: white box test generation for. NET. In: Proceedings of ACM international conference on tests and proofs (TAP), pp 134–153
100. Winterholer M (2006) Transaction-based hardware software co-verification. In: Proceedings of forum on specification & design languages (FDL)
101. Xie F, Liu H (2007) Unified property specification for hardware/software co-verification. In: Proceedings of international computer software and applications conference (COMSAC), pp 483–490
102. Xu X, Kimura S, Horikawa K, Tsuchiya T (2005) Transition traversal coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE international conference on formal methods and models for co-design, pp 259–260
103. Xu X, Kimura S, Horikawa K, Tsuchiya T (2006) Transition-based coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE Asia and South Pacific conference on design automation, pp 1–6