Chapter 9
Model-Driven Methodology for the Development of Multi-level Executable Environments

Fernando Herrera, Pablo Penil, Hector Posadas, and Eugenio Villar

Abstract Electronic system-level (ESL) methodologies have enabled the development of fast executable system performance models by relying on standard languages such as SystemC. Recent system-level dynamic, that is, simulation-based performance estimation techniques have enabled faster assessment of the design alternatives, and thus the design space exploration (DSE) of complex embedded systems. In this context, the development of system environment models able to reflect common and feasible use cases is crucial for achieving efficient and valid solutions at early design stages. However, such environment modelling can be as or more complex and costly than the system model development itself. The adoption of model-driven development (MDD), component-based design (CBD) and abstraction, can improve the productivity of the environment specification as it does for system specification. In this chapter, a multi-level model-driven methodology for the specification of executable environments is presented. The methodology supports the capture of the environment use cases by relying on the UML standard language and on standard profiles, i.e. MARTE and UTP, and uses UML components for a clean separation of system and environment, and of environment actors. Moreover, a SystemC executable counterpart is automatically generated from the UML-based environment model, coupling the documental and performance analysis levels. The approach is able to capture the communication protocol between system and environment, and also the environment functionality, which can embed either an abstract stimuli generation model, or actual functionality of I/O devices. Thus, different abstraction levels are supported in the functional modeling of the environment.

F. Herrera (✉) • P. Penil • H. Posadas • E. Villar
University of Cantabria, ETSIIT, Santander, Spain
e-mail: fherrera@teisa.unican.es; pablop@teisa.unican.es; posadash@teisa.unican.es; villar@teisa.unican.es

J. Haase (ed.), *Models, Methods, and Tools for Complex Chip Design*, Lecture Notes in Electrical Engineering 265, DOI 10.1007/978-3-319-01418-0_9,
© Springer International Publishing Switzerland 2014

9.1 Introduction

Integration capabilities have undergone a continuous growth (from 10^7 to 10^9 transistors) in the last decade, and further integration capabilities are envisaged [4, 14]. These integration capabilities have led to the possibility of producing more complex embedded systems. However, at the same time, this has involved a major challenge to overcome the gap between design productivity and integration capability. One of the main strategies adopted to overcome the design gap is the development of electronic system-level (ESL) design methodologies [18], where the key initial activity is system specification. Model-driven development (MDD) methodologies enable concepts for making specifications simpler and more understandable, which are major requirements for tackling the design challenge [17]. The usage of standard languages such as UML [23] provides understandability and portability of specifications.

After specification, the next task in an ESL design methodology is design space exploration (DSE) [3, 12, 18]. This activity is crucial for an early assessment of the optimal design decision, since about 90 % of the overall costs are determined in the first stages of the design [12].

Due to the high complexity of the systems and to the huge number of design alternatives, new estimation techniques, such as [8] and native simulation [7, 27], have been proposed for the assessment of the performance of each feasible design alternative. These techniques are dynamic, that is, simulation-based and provide simulation speed-ups of two orders of magnitude with regard to instruction set simulators [27]. Dynamic techniques require the definition of a stimuli environment. Performance results greatly depend on this stimulation, and thus on the design decisions resulting from this assessment. This makes dynamic performance estimation techniques suitable for customized and average optimizations of the systems, which is interesting in many application domains, e.g. wide consumer market, where efficient implementations and cost reduction are crucial. However, these techniques require to enable and facilitate the specification of the system environment as a set of use cases which comprise common cases and corner cases, to let the user dimension the system and assign a given quality of service and guarantees on constraint fulfillment when the system works under both “normal” conditions and under expected worst-case conditions.

However, the development of an executable stimuli environment which can be reused and properly linked to the DSE design flow could be easily more costly than the system specification and the extraction of its executable performance model. Because of this, the specification of the stimuli environment should also support design concepts which have been shown useful for system modeling and design. Adopting an MDD approach for the environment model enables abstraction, and other benefits, such as the application of code generation toolsets for the automated extraction of executable counterparts. This way the whole modelling task, and not only the system specification is covered.

In this paper, a methodology for the abstract modeling and automatic generation of an executable counterpart of the multilevel environment of an embedded system is presented. Specifically, in this methodology, the verification environment is modeled by using UML [23], MARTE [21] and the UML Testing Profile (UTP) [22]. After this stimuli environment model has been developed, a code generator enables the automatic production of an executable SystemC code which reflects all the information captured in the UML model of the stimuli environment. This SystemC model can be easily built up with verification functionality.

The methodology enables the modeling of the environment actors and the specific sequences of interface function calls between the system and the environment actors. This enables to exercise the system, in such a way that depending how the environment actors couple system interfaces, the concurrency of the system application can be more or less exploited, which, in general, impacts on the decision of the optimal mapping to the system platform. It also serves to validate the concurrency structure of the system. These modeling aspects, and the generation of the SystemC executable model were introduced in a previous work [11]. This paper presents the overall methodology, which has been enriched to support additional features. Specifically, the methodology supports now a simplified description of the environment, through implicit sequential diagrams. Moreover, a tool-independent link between the environment model and the files containing the functionality of the environment has been enabled. Finally, the methodology provides now means to describe the environment behavior at two abstraction levels, a first one by capturing an abstract, target independent, description of the environment behavior; and a second, more detailed level, where the system code, typically legacy code, or I/O peripheral driver, is considered part of the environment. This is interesting, for instance, when the system component is use as an input for an automatic synthesis process [28].

The structure of the chapter is the following one. In Sect. 9.2, related and previous work will be presented. Section 9.3 presents the methodology for modeling the environment. Section 9.4 introduces the tooling supporting the environment modeling methodology. Section 9.5 explains how the SystemC model is simulated together with the system performance model. Finally, Sect. 9.6 explains how the methodology has been validated. Sections 9.7 and 9.8 ends with the main conclusions and future work, respectively.

9.2 Related and Previous Work

Several UML-based methodologies for the modeling of an embedded system have been proposed. Intuitively, the modeling of the system environment can be tackled by directly applying the system modeling methodology. However, although maintaining some homogeneity in the modeling methodology of both environment and system can be convenient, environment and system modeling have different

constraints and needs. Moreover, certain distinction and asymmetry is required, for instance, to let the implementation framework knows what to synthesize or compile.

At MDD level, this has motivated the development of the UTP standard [22]. Despite the relatively long availability of UTP, only a few approaches have tried to provide support for UTP [15]. In [15] UML and UTP are used for deploying Model-Based Testing in Resource-Constrained-Real-Time Embedded Systems (RTES-RC). This paper aims to close this gap and discusses a concise set of UTP artifacts in the context of model-based testing for RC-RTES. A detailed discussion on the test artifact generation algorithm is presented, demonstrating the applicability of the approach in a real-life RC-RTES example. In [16] the integration of executable uses cases as a supplement to MDD is proposed. It is seen as a model-based approach to requirements engineering. Specifically, a coloured petri-net model is used to express user requirements.

SystemC [13] enables the building of executable, platform agnostic validation environments. SystemC has been widely used for system-level and reusable test bench development, and it has already enabled the development of advanced features for supporting verification and debugging. SystemC has been targeted from several model-based methodologies, focused on the description of a system for the development of executable performance models. Related to verification, in [2] formally sound B models are used to verify model refinement, and translated into SystemC.

The cooperation of fast performance estimation techniques with SystemC has enabled fast simulation of a complex embedded system including SW and custom HW parts. In [19], SW parts are simulated with a virtualization environment called Simics, while SystemC was used for modeling custom HW devices. In [19] the SystemC kernel is made a slave system of the Simics kernel, and an efficient technique for check pointing of the SystemC custom HW was presented. In this approach SystemC is used to model HW devices as an integral part of the system model

The recent merging of the Open SystemC Initiative (OSCI) with Accelera [1] makes targeting SystemC even more interesting, once the proposed modeling environment can benefit from cooperation with other verification approaches, such as the Universal Verification Methodology (UVM) [30].

Although the methodology proposed here does not preclude its extension for supporting a verification methodology, the main motivation was to enable a UML-based, abstract and flexible modeling of the stimuli environment and the automated generation of a standard and executable counterpart. Much of the features of the methodology, presented in [11], were necessary to complete the UML/MARTE COMPLEX modeling and virtual system generation framework [5]. In such a framework, a UML/MARTE-based methodology [9] enables the development of an embedded system model, including the main features in terms of impact on performance. This model can be captured with Papyrus [24], a tool for capturing UML models, which is fully integrated in Eclipse [6]. A related tool, which includes model validators, model-to-text generators [10], and the SCoPE native-based simulation infrastructure [26], enables the automated generation of the

performance model. Moreover, the additional features shown in this chapter, e.g. the integration of target-dependent application code as part of the environment, have been applied to a MDD framework enabling automatic software synthesis for many-core framework.

9.3 Environment Modelling Methodology

The proposed environment modeling methodology enables a UML-based modeling of the environment, which can be smoothly integrated in a component-based methodology [29]. Specifically, it is integrated in methodologies such as COMPLEX [5] and PHARAON [25], where the whole system is enclosed in a UML component, and where different views, in the shape of UML packages are used to capture the system model. The proposed specification of the environment, as shown in the following sections, cleanly separates the system information from the model containing environment actors, their functionality and their interconnection with the system model.

9.3.1 Environment Structure and Connection to the System

The user can develop the model of the environment at the same abstraction level, clearly separating the system from the verification environment. Specifically, the user will enclose all UML modeling elements within a specific view of the model: the verification view. Figure 9.1 shows an example with the hierarchy of UML elements used for the modeling of the environment proposed, and which can be taken as a reference for the following discussion. The verification view is actually a UML package, typed with the «VerificationView» stereotype, which contains the model elements which describe the verification environment facilitating a tool-independent separation of system and verification elements. The verification view declares the whole set of actors which compose the environment as a set of UML

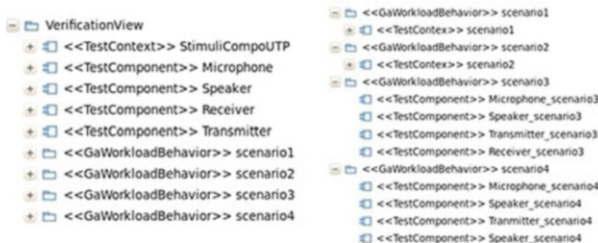

Fig. 9.1 Several scenarios are supported

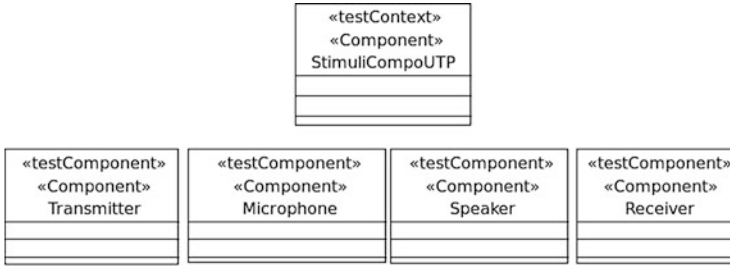

Fig. 9.2 Declaration of environment and test components in the VerificationView

components with the UTP «`TestComponent`» stereotype applied (Fig. 9.2). An additional UML component, with the UTP «`TestContext`» stereotype is used for the declaration of a verification environment (StimuliCompoUTP in Fig. 9.2). The internal structure of this component, depicted in the composite diagram in Fig. 9.3, reflects the interconnection structure of the system and environment component instances. Environment component instances are captured as UML properties typed as «`TestComponent`» components, and the system component instance. The system component is captured as a UML property typed as the UML component reflecting the system, and which exports I/O functional interfaces. Notice that this system component is not in the verification view, but in a view related to the specification of the system description. This means a dependency, so the verification view depends on the system views. In addition, the referred system component must be specified by the UTP «`SUT`» (System Under Test) stereotype. Through this scheme a clear separation is established between the system element and the environment elements.

The composite diagram in Fig. 9.3 also shows the port to port connection. After this interconnection, the environment components that provide the services required by the system are stated. Similarly, services provided by the system can be invoked from the environment modules.

9.3.1.1 Modelling the Behaviour of the Environment: One Scenario

As well as the interconnection between the environment elements and the system, the proposed methodology supports the specification of the behavior of the environment. First, the methodology adds a main concept, the scenario. A scenario models the activity of the different environment components, and their interaction with the system for a given use case. Several scenarios are possible (see Sect. 3.4). It means that, while one scenario can involve activity in all the environment components, each with a specific behavior, a different scenario can model activity only in some environment actors, with a different behavior. Each scenario can be described by:

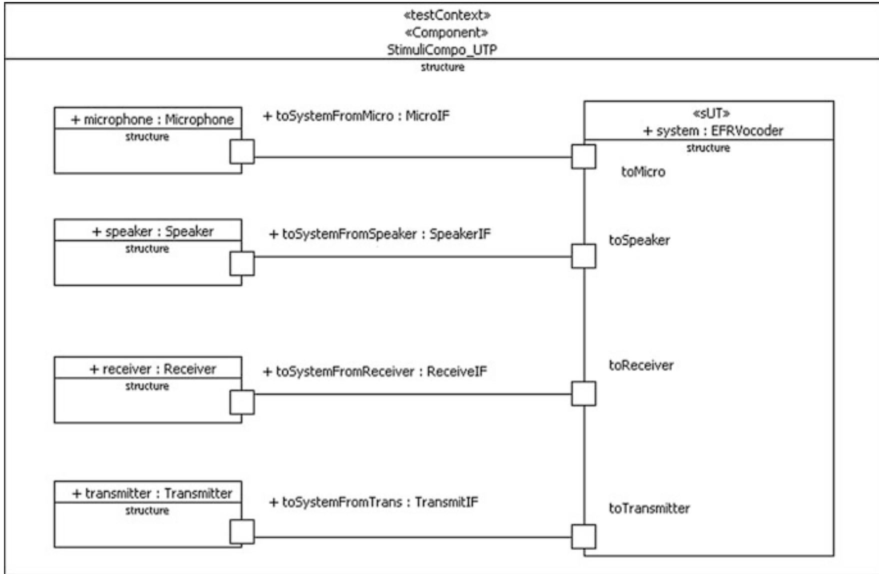

Fig. 9.3 Environment structure for an EFR vocoder

- Interactions, each interaction between an environment component and the system is a totally ordered sequence of service calls, which can be synchronous or asynchronous.
- References to file sets, where the specific functionality is allocated.

A scenario is captured as a UML package stereotyped with the MARTE stereotype «GaWorkloadBehavior». A scenario package has to be a child element of the VerificationView package. The methodology does not enforce the use of both type of functionality. Indeed, the UML/MARTE COMPLEX methodology uses only interaction diagrams, such a later code generation phase produce templates with empty functionality. Therefore, filling the functionality is left to the user, as a manual task. Instead, in the context of the UML/MARTE PHARAON methodology, file references are included, while interaction diagrams are omitted. In practical terms, it involves two different environment modelling styles. However, both of them fit to the more general scheme presented here.

Interaction Modeling

Interaction modeling relies on UML interactions, where UML lifelines can represent either, the system or an environment component. As a prerequisite, the GaWorkloadBehavior package (scenario package) must comprise a UML component with the UTP «TestContext» stereotype. Then, this new

Fig. 9.4 Generalization for referencing environment components in the modeling of each scenario

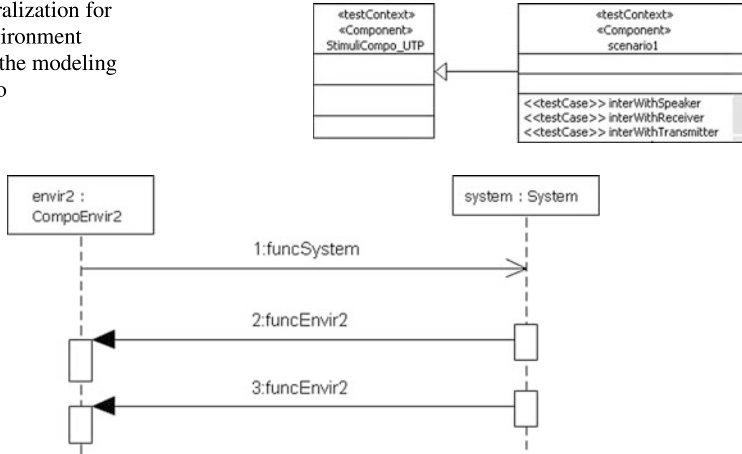

Fig. 9.5 Sequence diagram for a port to port interaction

«`TestContext`» component is generalized by the «`TestContext`» component where the environment structure is specified (Fig. 9.3). This «`TestContext`» components relation is modelled by an UML generalization (Fig. 9.4). This way, the component instances reflecting environment components can be accessed and later on associated to UML lifelines, used for specifying scenario interactions.

A scenario comprises the specification of all the interactions over time between the system and the environment components. They are described by means of one or more UML interactions (as child elements of the «`TestComponent`» component) which also have the UTP «`TestCase`» stereotype applied. A scenario description is complete when all the interactions cover all environment components and their ports. However, this is not a required condition since a scenario can represent a use case which might not require an interaction with all system ports. UML interactions are graphically captured by means of sequence diagrams. Figure 9.5 shows a sequence diagram capturing the interaction between an environment component and the system component. A lifeline references the instance of the system component, while the other lifeline references an instance of one environment component. Making these references is feasible thanks to the specialization shown in Fig. 9.4. As well as the lifelines, the interaction contains the set of UML messages exchanged between the system and the environment component. These messages represent function calls, as services provided either by the system to the environment or viceversa.

The different environment components are communicated with the system by using interfaces and specified by the MARTE stereotype «`ClientServerSpecification`» and they contain the functions used for the component interconnections. The interfaces are included in the model view «`FunctionalView`». Depending on the goals of the designer, these interfaces can represent auxiliary interfaces used for defining functions for validating the

concurrency and behavior of the system in different use cases. After the system validation using the stimuli specifications, the designer can develop the environment interfaces for physical implementation in order to access to the environment actors which represent peripherals. In this case, these interfaces are the implementation mechanisms for accessing these peripherals according to predefined functional and non functional requirements.

The sense of a UML message is captured through its “from” and “to” attributes (in the diagram, the “to” attribute corresponds to the tip of the arrow). The sense states whether the system calls a function provided by the environment (“from=system”) or, on the contrary the environment requires a service provided by the system (“to=system”).

Two different types of UML messages are used: synchronous messages and asynchronous messages. They enable the specification of synchronous and asynchronous services. Synchronous services require the return of the function call, e.g. because the client expects some output data from the service call. However, it might be interesting to specify service calls which just provide input data for triggering the service, and which immediately return and let either the system or the environment component go on executing. A UML synchronous message is represented by a solid arrow head (as in the two messages from the system to the environment component in Fig. 9.5). UML asynchronous messages are represented by open arrow heads (as in the message from the environment component to the system in Fig. 9.5).

The name of the message identifies which function is called. This is required because an interface can comprise several service functions. In the example of Fig. 9.5, the environment component first calls the funcSystem service provided by system component. Next, the system calls the “funcEnvir2” service twice provided by the environment component. These functions must be part of the interface accessed through any of the environment component ports. And, as explained, those interfaces could present more functions, e.g. “funcSystem2” or “funcEnvir1”.

Although the sequence diagram graphically reflects a total order in the exchange of messages, this information is not contained in the UML interaction. Diagrams provide a graphical representation, but not all that graphical information is contained in the UML model. It is required to add this information in a way it can be preserved in the .uml file read, so available to the toolset around the model, e.g. model validation or code generation frameworks. For it, in the proposed methodology, a unique order identifier (“i:”) prefixes the message name, which is part of the UML model. In this way, a total order in the exchange of messages can be specified at local level. Local level means the interaction between the system and a single environment component, which requires two lifelines, as in Fig. 9.5.

The user can specify all the interactions in a compact way. In fact, the messages of a sequence diagram can refer to functionalities of different ports of the environment component, and thus of the system. Moreover, one UML interaction can be used for specifying the interaction of the system with more than one environment component. In Fig. 9.6 a sequential diagram shows the communication between the system and two environment components. In principle, in a diagram like this, the sequence of messages exchanged between the system and one environment

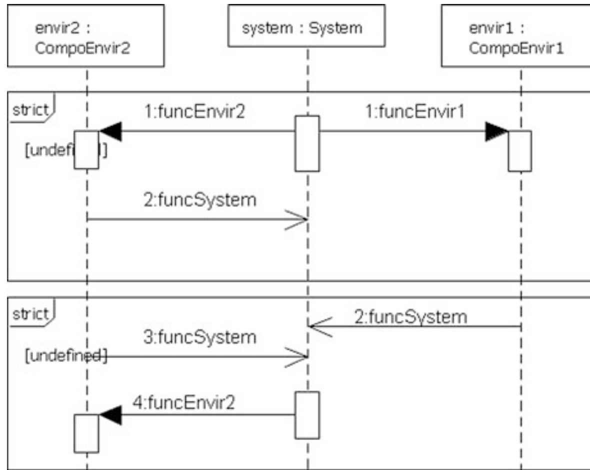

Fig. 9.6 A sequence diagram stating synchronization conditions between the systems and two environment components

component is not related to the sequence of messages exchanged between the system and another environment component. That is, in the example of Fig. 9.6, if the reader forgets by now the strict labeled boxes, there would be in principle no order relationships between the messages exchanged between system/envir2 lifelines, and the messages exchanged between system/envir1 lifelines. That is, the i-th message of system-envir2 communication might happen before, at the same time, or after the j-th message of system-envir1 communication.

However, use cases may actually require the modeling of these types of constraints, because the environment itself can also present dependencies, e.g. among environment components, and thus provoke dependencies between system interfaces which do not have its origin in the system itself. In the proposed methodology, the user can specify order relationships among messages exchanged by different environment components with the system. This is done by using UML CombinedFragments, shown as boxes in Fig. 9.6. Specifically, in the Fig. 9.6 example, a strict combined fragment is used. The strict combined fragment groups the execution of the set of messages it covers, so that all covered messages have to be executed before or after the remaining messages. That is, it defines an atomic region of messages exchange. Taking the previous discussion into account, the use of combined fragments adds a higher ordering level to the specification of the environment, in the sense that all the messages encapsulated in the same combined fragment are associated with a single and higher order implicit ordering index. Moreover, it also adds a global ordering since it covers the interaction of the system with more than one environment component.

For instance, the two combined fragments in Fig. 9.6 state that "1:funcEnvir2", envir2 call to "2:funcSystem", and "1:funcEnvir1" will have an associated higher order k-th index; and will have to be executed before or after "3:funcSystem",

"4:funcEnvir1", and envir 1 call to "2:funcSystem" messages, with a m-th high order index. The total order of each local system-environment component interaction, imposes a total order in the execution of the combined fragments $(m < k)$. In other words, the bottom combined fragment in Fig. 9.6 has to happen after the top combined fragment. Local and global ordering has to be coherent, thus two environment component lifelines cannot impose an order conflict on two combined fragments (e.g., $m < k$, $m > k$). As a result, the methodology enables the specification of a partial order of messages exchange. A partial order of messages is specified where there can be order relationships among the messages sequences exchanged by the system with different components. Specifically, in the Fig. 9.6 diagram, "1:funcEnvir2" and "2:funcSystem" messages, reflecting function calls among envir2 environment component and the system, will take place before the "2:funcsystem" call done by envir1 environment component. Similarly, the diagram in Fig. 9.6 specifies that "3:funcSystem" and "4:funcEnvir1" will take place after "1:funcEnvir1".

The methodology also supports another two combined fragments. The "loop" combined fragment (loop) is used for specifying repetitive subsequences of message exchange. The "parallel" combined fragment (par) is used to model that certain groups of services either provided or required by the same environment component can be executed in parallel.

The features presented up to here enable the specification of a partial order of service calls in the environment. Formally speaking, this is the most abstract way to specify time constraints in the environment model. Furthermore, the proposed methodology enables the association of physical time information with the environment model. Specifically, the initiation of each service call can be placed in a specific physical time stamp. In order to specify it, the MARTE «`TimedProcessing`» stereotype is used. This stereotype is applied to the UML message which reflects the service call placed in physical time. The stereotype provides the attribute "start", which denotes a UML Time Event, which in turn, is placed in physical time through a UML Time Expression.

Implicit Interactions

The methodology admits the use of implicit interactions. It is a practical feature which saves time and complexity in the modeling of common environment models in a specific domain. Specifically, it means that an environment component will have a default interaction scheme associated, if no specific UML interaction has been captured and associated with it. A methodology can define this implicit interaction. For instance, in a domain space oriented methodology, hard real-time analysis methodologies will typically assume a reactive environment, and an active system which does not block because of waiting for environment services, whose response might be non-predictable and/or unbounded. It is typically modeled through an interaction scheme with an infinite loop enclosing an incoming asynchronous UML message. I.e. requested by the system. Therefore, this interaction scheme is a good implicit interaction candidate for space domain oriented applications.

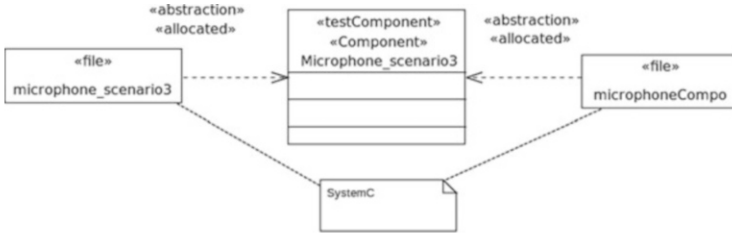

Fig. 9.7 File association with a TestComponent

Association of File Sets

The proposed methodology supports the capture within the environment model of a file set related to an environment component. All the interface functions present in the interactions of the environment component could find its implementation in the file set or not. In the former case, this feature enables a fully automated integration of source code, e.g. previously implemented test benches in SystemC or C/C++, with the rest of the environment model information. The association is also applicable to the case of relying on implicit interactions. In such a case, the functions called can be inferred from the associations or be explicitly captured as environment component operators. The scenario when this happens is explained later on. As mentioned, it might also happen that not all the functions associated with the environment component are found in the associated file artifacts. In such a case, code generators produce the interface function declarations and the implementation templates.

Figure 9.7 shows how file set association is captured in the proposed methodology. Let assume that environment functionality is available in a set of source files. The file set is modeled as a UML artifact typed by the UML standard stereotype «`File`». These `file` artifacts are included in the «`FunctionalView`» package. These `files` represent previously created test-benches and, thus, can be reused in different designs or the user code of the component functionality.

When the environment component functionality is specified by files the corresponding `GaWorkloadBehavior` package should contain additional `TestComponent` components. These `TestComponent` components are generalized from the `TestComponent` defined in `VerificationView` package (Fig. 9.8). These new `TestComponent` have associated the different files where the functionality of the scenario is implemented. This `file-TestComponent` association is modeled by a UML abstraction specified by the MARTE stereotype «`Allocated`» (Fig. 9.7).

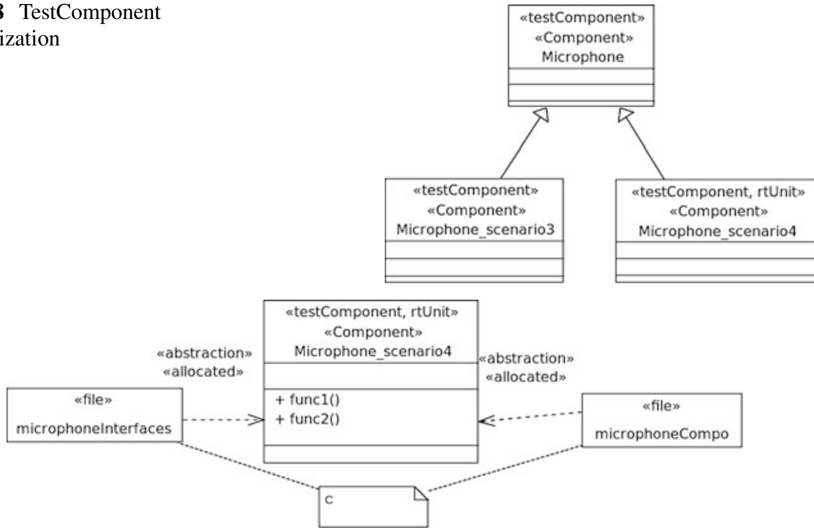

Fig. 9.8 TestComponent generalization

Fig. 9.9 TestComponents as application components

9.3.2 Levels of Abstraction in the Specification of Environment Behaviour

The proposed environment modeling methodology supports de distinction at the modeling level between an abstract (target independent) and a target dependent description of the environment behaviour.

Figure 9.7 exemplifies the modeling of the former case. Then, the model reflects that the source code linked to the environment model reflects a functional model of the environment and which can be therefore used for creating an executable counterpart of the environment model. This is independent from the language. However, a language such as SystemC is typical of this case, since SystemC is a language suitable for implementation agnostic models. Moreover, the proposed methodology supports a system-dependent, and more specifically target dependent, description of the environment component. The case is show in Fig. 9.9. The idea is that methodologies, require the consideration of certain application components as environment components. For instance, in PHARAON methodology, there are application or platform software components which reflect the software layer for accessing I/O peripherals. However, it is not interesting to consider them as part of the system, e.g., for code synthesis effects. However, they can contain C or C++ code reflecting the functionality of the device driver which facilitates and makes realistic the development of the environment, or that can even enable hardware in the loop methodologies (enabling the integration of the peripheral hardware, e.g. a camera, together with its driver, as part of the environment, and the rest of the system under design).

A distinctive aspect of this case is that the source code ("C" code in Fig. 9.9 example) is likely target dependent code. E.g., a high-level device driver code used as part of the test bench will likely make calls to an operative system instance, which in turn is part of the system under design, and thus such operative system instance is captured in the system specification. Therefore, this model needs to break the pure separation of scopes between the environment and the system. As this is not the general rule in a MDD methodology, it is convenient to mark the environment components which such access to the system model internals. It is shown in Fig. 9.9, where the environment component is typed by the «RtUnit» MARTE stereotype, as well as by the UTP «TestComponent» stereotype.

9.3.3 Modeling Several Scenarios

The proposed methodology supports the modeling of several GaWorkloadBehavior packages. This enables a set of different stimuli associated with different use cases to be captured in a single model. Then, a DSE exploration can be done for each use case, and the design can be tailored for a set of use cases. A a wide set of scenarios can be also used for validating a single design for different use cases. In addition, the methodology provides different GaWorkloadBehavior packages where the file association is used. This fact enables the verification of the system with test-bench files (SystemC or C/C++) and so the verification of the peripheral interfaces for the final implementation. In this way, both modeling mechanisms can cohabit in the same VerificationView package which enables the definition of the different design stages in the same model. In order to specify several scenarios, the user only needs to specify a new GaWorkloadBehavior package, in turn containing a TestContext component. This TestContext component again generalizes the TestContext component and owns as many interactions as the user requires for describing the new scenario.

9.4 Toolset

The environment modeling methodology presented is partially supported by a toolset which relies on Eclipse and Papyrus. Specifically, two code generation tools have been separately implemented up to now. The first generator produces a SystemC counterpart from the ULM interactions. The second one, implements the generation of the file structure from the model. These generators have been written in the standard Model-to-Text (M2T or MTL) language [20], to improve its portability across different model-to-text transformation engines.

9.4.1 SystemC Generation

The code generation is in charge of producing all the SystemC code reflecting the structure of components and concurrency present in the UML/UTP/MARTE environment model. It also produces the service calls fulfilling the partial order specified in the environment model by means of sequential diagrams. The generator does not produce functional code, whose insertion is left to the user. However, in order to enable the production of an executable environment model from the first moment after the generation, void functions with debugging printouts are produced. This permits a fast initial check of the SystemC code produced and provides a basis for indicating to the user where to insert functional code.

Code generation is actually done in two phases. First a model-to-text transformer translates the UML environment model into a set of macros. A specific front-end of the SCoPE simulation framework provides the SystemC translation for these macros. The SCoPE framework enables the compilation of a dynamic library (instead of a static executable file) for each scenario. In this way, the approach is modular at executable level, in the sense that each scenario of the SystemC environment has its own .so file, separated from the .so files of the system executable model itself. The generator basically maps all verification views to a single SystemC module (thus there is no UML environment component module mapping). The code generator produces at least one SystemC process containing a sequence of channel accesses for each environment component. This sequence fulfils all the order constraints specified through the sequence diagrams, as explained in Sect. 9.3.

9.4.2 File Structure Generation

The other implemented code generator enables the generation of the file structure. In order to characterize an application component, the files whose functionality is implemented, the interfaces required/provided and the component's functions should be specified. With this information, the code generator creates the application files (`.c/cpp` and `.h`). These files include the declaration of the functions provided by the application component through its interfaces and the other functions specified in the model as internal component functions. The functions of the application environment component can be associated with a specific file in order to specify that a function has to be included in this file. Otherwise, the code generator produces two additional files, apart from the files specified in the model, one which includes the declaration of the functions of the interfaces provided by the component and another file which includes the internal functions of the component. Then, by using a UML comment, the programming language is annotated (Figs. 9.7 and 9.9).

In addition, the second generator enables the generation of the makefiles required for the compilation of the environment components with the rest of the application in order to be executed in the simulation tool. This feature enables the designer to focus on the functionality implementation and not on the infrastructure required for the simulation tool execution.

9.5 SystemC Simulation with the System Performance Model

The system simulation infrastructure used (SCoPE+) works on top of the SystemC kernel. Time advance, buses and peripherals have been developed using the standard SystemC features. As a result, no kernel synchronizations between the system simulation kernel and the environment kernel are required, which contrasts with other approaches such as. However, direct use of SystemC environments is not possible since both parts rely on different models of computation.

On the one side, the system model uses a client-server component-based communication, based on function calls. Each time a client component requires a service, it calls a function that is implemented and provided by another provider component. Under this perspective, the data transferred among components are the input and output arguments of the functions of component interfaces. Each function called by a client component means sending input arguments to server components and, if output arguments are expected, they are sent from the m server component to the client component. On the other side, the SystemC environment relies on interfaces based on transfers which use SystemC channels. These channels receive packaged data and provide different communication semantics: blocking/non blocking, with/without memory, etc.

Interconnection wrappers are used to adapt each channel of the SystemC environment to each function in the system interface. In these wrappers, communication accesses are divided into two steps: a request step, where the input arguments are sent, and a response step, where output arguments return to the calling task. In the meantime, the calling task is blocked, waiting for the response. This approach models the blocking nature of function calls. Additionally, data transfers are packaged by copying the arguments of each function call in a buffer that is sent through the SystemC channel as a single unit.

9.6 Example

The suitability of this methodology has been demonstrated through the development of environment models for an EFR vocoder example. The interrelation between the environment components and the system has been specified in two different ways, by using the UML interactions and by means of the files. The SystemC code was automatically extracted from the environment and simulated with the executable performance model, automatically extracted from the UML/MARTE model of the system, after requiring only the injection of the functional code.

Figure 9.3 shows the structure of a first environment model developed for simulating a full-duplex transmission operation mode. In this use case, the coder and the decoder functionalities of the vocoder are stimulated independently (and thus potentially at the same time) and they have to exhibit concurrent behavior able to attend to coding and decoding services at the same time. This model is composed of

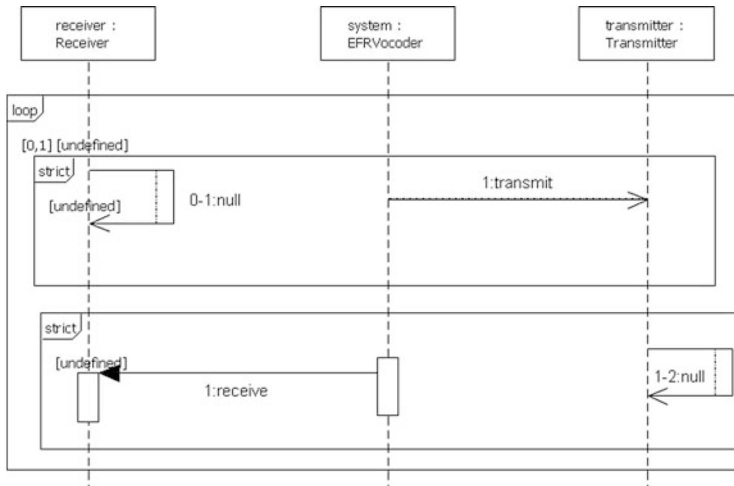

Fig. 9.10 Modeling of order constraint for remote closed loop modeling

four environment components. In this way, independent threads stimulate the coder and decoder within the vocoder, and the collection of coded audio (transmission) also runs independently of the collection of the decoder output (speaker output).

In order to show the usage of more than one scenario, the environment with the structure shown in Fig. 9.3 was extended to support a second scenario, specifying a remote close loop use case. In principle, this could also be done with the structure shown with the addition of some functionality for emulating channel effects. The only additional requirement to cover the specification needs of "scenario2" was the ability to specify that the service for transmitting the i-th coded frame should always be called before the reception of the corresponding i-th received frame, reflecting the same coded frame but corrupted by the channel effect after traversing the remote closed loop. Notice that since different components (and processes) are inferred for transmitter and receiver, and since the receiver can directly read from a file with the coded and corrupted frames, the ordering constraint is required to model the actual causality existing between the i-th coded frame sent and the corresponding received frame after traversing the remote loop.

Figure 9.10 shows the sequence diagram which reflects the interaction of transmitter and receiver with the system. Two strict combined fragments are used to reflect the aforementioned order condition, which will impose an order relationship in the test-bench generation. The combined fragments cover lifelines from both the transmitter and from the receiver environment components. Notice that the null activity of one environment component while the other is attending to a service is explicitly modeled. The sequence diagram in Fig. 9.10 also illustrates the usage of synchronous and asynchronous messages for modeling synchronous and asynchronous service calls. In fact, the EFR Vocoder uses the asynchronous transmit service since the vocoder can go on coding frames after the coded audio frame is delivered to the transmitter. However, the vocoder uses a synchronous call

for the services to be called since the decoder cannot work without having received the output parameter of the received function, that is, the received audio frame.

The scenario "scenario3" (Fig. 9.1) is part of a test-bench collection to be used for checking the vocoder system. In this case, it is not necessary to specify the interactions for functionality description; only the capture of the different files in the model is required. The infrastructure for system simulation (makefiles) is automatically obtained for the SCoPE+ simulation.

Finally, the scenario "scenario4" (Fig. 9.1) specifies the target dependent source code (Fig. 9.9) of the different device drivers for the code synthesis process. The makefiles generated included all the information required for compiling the application in the target board (cross compiler, flags...).

9.7 Conclusions

Support for MDD and related tools in the specification of a stimuli environment is necessary for the development of performance models for complex embedded systems. It enables fast model development and efficient design decisions in the DSE phase. This paper describes a methodology for UML/MARTE/UTP modeling of an environment which supports the specification of the main environment actors and their interconnection with the system; the specification of the interaction of environment components with the system as partially ordered sequences of service calls; and the specification of several scenarios for reflecting different use cases. In addition, the methodology enables the capture of the files which implement the functionality of test-benches for system simulation in different scenarios. This avoids modeling the environment-system interactions in order to take advantage of the previously implemented test-benches. Moreover, the methodology enables the specification of peripheral interfaces to be developed for the final system implementation which, by using the simulation, enables the verification of the interface's functionality required for the final system synthesis implementation.

Tools support the generation of the SystemC code and the makefiles infrastructure for execution in the simulation tool.

9.8 Future Work

Some methodological aspects have still not been implemented in a specific tool. Specifically, there is no support for the generation of the function calls sequence which defines the behavior of an application environment component in a specific scenario. This generator would generate the complete, ordered sequence of function calls established between the system and the environment application component. This sequence of call functions would be included in a file. This file is automatically generated. However, it is possible to define where these function calls should be allocated in a specific file that has previously been captured in the model. In order

to do so, a UML operation should be specified in the corresponding application environment component. Then, this operation is associated with a file artifact. In this way, the sequence of function calls which composes the communication statements are specified in the body of this function allocated in the file artifact. Finally, a complete framework which integrates all the code generators and enables all the environment specification and simulations is still to be implemented.

Additionally, the environment modeling methodology could be extended. A natural extension of this work consists in the addition of verification capabilities, by using further UTP stereotypes for the specification of assertions, supported by an extension of the validation tool.

Acknowledgements This work has been funded by the European FP7-247999 COMPLEX project, FP7-288307 PHARAON project and by the Spanish MCI TEC2011-28666-C04-02 DREAMS project.

References

1. Accellera: http://www.accellera.org/home/ (2013)
2. Cansell, D., Culat, J.F., Méry, D., Proch, C.: Derivation of SystemC code from abstract system models. In: Proceedings of FDL 2004, Lille, Sept 2004
3. Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A.J., Todd, L.: Surviving the SOC Revolution: A Guide to Platform-Based Design. Kluwer, Boston (1999)
4. Chiang, S.Y.: Keynote speech. In: Proceedings of ARM Techcom Conference, Santa Clara, Oct 2011
5. COMPLEX Project: http://complex.offis.de (2013)
6. Eclipse project website: http://www.eclipse.org/ (2012)
7. Gerin, P., Hamayun, M., Petrot, F.: Native MPSoC co-simulation environment for software performance estimation. In: Proceedings of the CODES+ISSS'09, Grenoble, Oct 2009
8. Gligor, M., Fournel, N., Pétrot, F.: Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In: Proceedings of the CODES+ISSS'09, ACM, Grenoble, France (2013)
9. Herrera, F., Peñil, P., Villar, E., Ferrero, F., Valencia, R.: An embedded system modeling methodology for design space exploration. In: Jornadas de Computación Empotrada (JCE), 2012. Alicante, Jornadas Sartenco. Elche, Sept 2012
10. Herrera, F., Posadas, H., Villar, E., Calvo, D.: Enhanced IP-XACT platform descriptions for automatic generation from UML/MARTE of fast performance models for DSE. In: DSD, Izmir, Turkey 2012
11. Herrera, F., Penil, P., Posaads, H., Villar, E.: A model-driven methodology for the development of SystemC executable environments. In: Proceedings of the FDL 12, Viena, Sept 2012
12. Holzer, M.: Design space exploration for the development of embedded systems. Thesis dissertation, Vienna University of Technology, Vienna (Apr 2008)
13. IEEE Std. 1666-2011: IEEE Standard for SystemC® Language Reference Manual. http://standards.ieee.org/getieee/1666/download/1666-2011.pdf (2012)
14. Intel 22 nm Technology: http://www.intel.com/content/www/es/es/silicon-innovations/intel-22nm-technology.html?wapkw=22nm (2013)
15. Iyenghar, P., Pulvermueller, E., Westerkamp, C.: Towards model-based test automation for embedded systems using UML and UTP. In: IEEE 16th Conference on Emerging Technologies And Factory Automation (ETFA), Toulouse, Sept 2011, pp. 1–9

16. Jogesen, J.B.: Executable use cases: a supplement to model-driven development? In: Model-Based Methodologies for Pervasive and Embedded Software, MOMPES, Braga, Portugal 2007
17. Kopetz, H.: The complexity challenge in embedded system design. In: 11th IEEE ISORC, Orlando, May 2008
18. Martin, G., Bailey, B., Piziali, A.: ESL Design and Verification: A Prescription for Electronic System Level Methodology. Systems on Silicon. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007). ISBN: 9780080488837
19. Monton, M., Gladigau, J., Haubelt, C., Teich, J.: Checkpoint and restore for SystemC models. In: Borrione, D. (ed.) Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs. Springer, Dordrecht/New York (2010)
20. OMG: MOF Model to Text Transformation Language (MOFM2T), 1.0. http://www.omg.org/spec/MOFM2T/1.0/ (2008)
21. Object Management Group. UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1. (2011). Available in http://www.omg.org/spec/MARTE/1.1/. Accessed 2013
22. Object Management Group. UML Testing Profile (UTP). Version 1.1. (2012). Available in http://www.omg.org/spec/UTP/1.1/. Accessed 2013
23. OMG Unified Modeling Language: Infrastructure and Superstructure. V2.4.1. www.uml.org (2013)
24. Papyrus: http://www.eclipse.org/modeling/mdt/papyrus/ (2012)
25. PHARAON project web: http://pharaon.di.ens.fr/ (2013)
26. Posadas, H., Real, S., Villar, E.: M3-SCoPE: performance modeling of multi-processor embedded systems for fast design space exploration. In: Silvano, C., Fornaciari, W. Villar, E. (eds.) Multi-objective Design Space Exploration of Multiprocessor SoC Architectures: The MULTICUBE Approach. Springer, New York (2011)
27. Posadas, H., Díaz, A., Villar, E.: Annotation techniques and RTOS modeling for native simulations of heterogeneous embedded systems. In: Tanaka, T. (ed.) Embedded Systems-Theory and Design. Intech, Rijeka (2012)
28. Posadas, H., Penil, P., Nicolás, A., Villar, E.: Automatic synthesis of embedded SW from UML/MARTE models based on memory space definitions. In: Design of Circuits and Integrated Systems (DCIS), Avignon, France 2012
29. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Professional, London (2002)
30. Universal Verification Methodology (UVM) 1.1 Class Reference: http://www.accellera.org/downloads/standards/uvm (2011)