MULTILANGUAGE SPECIFICATION FOR SYSTEM DESIGN AND CODESIGN

A.A. JERRAYA, M. ROMDHANI, PH. LE MARREC, F. HESSEL,
P. COSTE, C. VALDERRAMA, G.F. MARCHIORO, J.M.DAVEAU,
N.-E. ZERGAINOH
TIMA Laboratory
46 avenue Félix Viallet
38000 Grenoble France

1. Introduction

This chapter discusses specification languages and intermediate models used for system-level design. Languages are used during one of the most important steps of system design: the specification of the system to be designed. A plethora of specification languages exists. Each claims superiority but excels only within a restricted application domain. Selecting a language is generally a trade off between several criteria such as the expressive power of the language, the automation capabilities provided by the model underlying the language and the availability of tools and methods supporting the language. Additionally, for some applications, several languages need to be used for the specification of different modules of the same design. Multilanguage solutions are required for the design of heterogeneous systems where different parts belong to different application classes e.g. control/data or continuous/discrete.

All system design tools use languages as input. They generally use an intermediate form to perform refinements and transformation of the initial specification. There are only few computation models. These may be data- or control-oriented. In both cases, these may be synchronous or asynchronous.

The next section details three system-level modeling approaches to introduce homogeneous and heterogeneous modeling for codesign. Each of the modeling strategies implies a different organization of the codesign environment. Section 3 deals with intermediate forms for codesign. Section 4 introduces several languages and outlines a comparative study of these languages. Finally, section 5 deals with multilanguage modeling and cosimulation.

2. System Level Modeling

The system-level specification of a mixed hardware/software application may follow one of two schemes [1]:

1. Homogeneous specification: a single language is used for the specification of the overall system including hardware parts and software parts.
2. Heterogeneous modeling: specific languages are used for hardware parts and software parts, a typical example is the mixed C-VHDL model.

Both modeling strategies imply a different organization of the codesign environment.

2.1 HOMOGENEOUS MODELING

Homogeneous modeling implies the use of a single specification language for the modeling of the overall system. A generic codesign environment based on homogeneous modeling is shown in Figure 1. Codesign starts with a global specification given in a single language. This specification may be independent of the future implementation and the partitioning of the system into hardware and software parts. In this case codesign includes a partitioning step aimed to split this initial model into hardware and software. The outcome is an architecture made of hardware processors and software processors. This is generally called virtual prototype and may be given in a single language or different languages (e.g. C for software and VHDL for hardware).

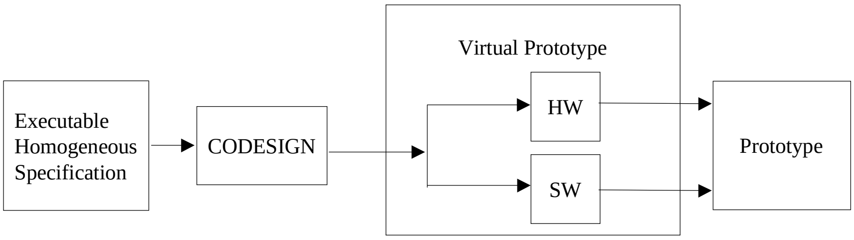


Figure 1 Homogeneous Modeling

The key issue with such codesign environments is the correspondence between the concepts used in the initial specification and the concepts provided by the target model (virtual prototype). For instance the mapping of the system specification language including high-level concepts such as distributed control and abstract communication onto low-level languages such as C and VHDL is a non trivial task [2, 3].

Several codesign environments follow this scheme. In order to reduce the gap between the specification model and the virtual prototype, these tools start with a low-level specification model. Cosyma starts with a C-like model called C ^{x} [4, 5]. VULCAN starts with another C-like language called hardware C. Several codesign tools start with VHDL [6]. Only few tools tried to start from a high-level model. These include Polis [7] that starts with an Esterel model [8, 9], Spec-syn [10, 11] that starts from SpecCharts [12, 13, 14] and [3] that starts from LOTOS [15]. [2,66] details COSMOS, a codesign tool that starts from SDL.

2.2 HETEROGENEOUS MODELING OF HARDWARE/SOFTWARE ARCHITECTURES

Heterogeneous modeling allows the use of specific languages for the hardware and software parts. A generic codesign environment based on a heterogeneous model is given in Figure 2. Codesign starts with a virtual prototype when the hardware/software partitioning is already made. In this case, codesign is a simple mapping of the software parts and the hardware parts on dedicated processors.

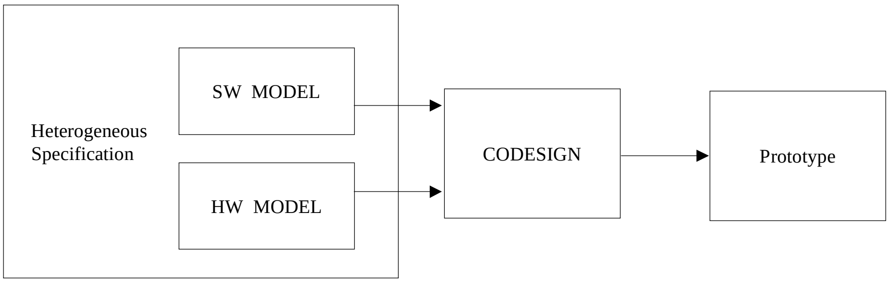


Figure 2 Heterogeneous Modeling

The key issues with such a scheme are validation and interfacing. The use of multilanguage specification requires new validation techniques able to handle a multiparadigm model. Instead of simulation we will need cosimulation and instead of verification we will need coverification. Cosimulation issues will be addressed in section 5. Additionally, multilanguage specification brings the issue of interfacing subsystems which are described in different languages. These interfaces need to be refined when the initial specification is mapped onto a prototype.

Coware[16] and Seamless[17] are typical environments supporting such a codesign scheme. They start from mixed description given in VHDL or VERILOG for hardware and C for software. All of them allow for cosimulation. However, only few of these systems allow for interface synthesis [18]. This kind of codesign model will be detailed in section 5.

2.3 MULTILANGUAGE SPECIFICATION

Most of the existing system specification languages are based on a single paradigm. Each of these languages is more efficient for a given application domain. For instance some of these languages are more adapted to the specification of state-based specification (SDL or Statechart), some others are more suited for data flow and continious computation (LUSTRE, Matlab), while many others are more suitable for algorithmic description (C, C++).

When a large system has to be designed by separate groups, they may have different cultures and expertise with different modeling styles. The specification of such large designs may lead each group to use a different language which is more suitable for the specification of the subsystem they are designing according to its application domain and to their culture.

Figure 3 shows a typical complex system, a mobile telecommunication terminal, e.g. a G.S.M. handset. This system is made of four heterogeneous subsystems that are traditionally designed by separate groups that may be geographically distributed.

a) The protocol and MMI subsystem :
This part is in charge of high-level protocols and data processing and user interface. It is generally designed by a software group using high-level languages such as SDL or C++.

b) The DSP subsystem :
This part is in charge of signal processing and error correction. It is generally designed by "DSP Group using specific tools and methods such as Matlab, Simulink [67], SPW or COSSAP [68].

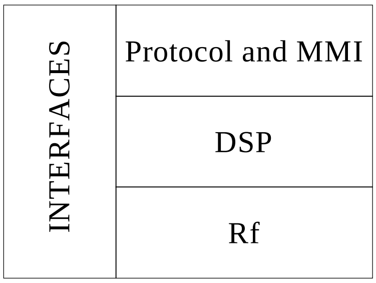


Figure 3 Heterogeneous Architecture of a Mobile Telecom Terminal e.g. G.S.M. handset

c) the DSP subsystem :
This part is in charge of the physical connection. It is generally made by an analog design group using another kind of specific tools and method such as CMS [69].

d) The interface subsystem :
This part is in charge of the communication between the three other parts. It may include complex buses and a sophisticated memory system. It is generally designed by a hardware group using classical EDA tools.

The key issue for the design of such a system is the validation of the overall design and the synthesis of the interfaces between the different subsystems. Of course, most of these subsystems may include both software and hardware.

Figure 4 shows a generic flow for codesign starting from multi-level specification. Each of the subsystems of the initial specification may need to be decomposed into hardware and software parts. Moreover, we may need to compose some of these subsystems in order to perform global hardware/software sub-systems. In other words, partitioning may be local to a given subsystem or global to several subsystems. The codesign process also needs to tackle the refinement of interfaces and communication between subsystems.

As in the case of the heterogeneous modeling for system architecture, the problems of interfacing and multilanguage validation need to be solved. In addition, this model brings another difficult issue: language composition. In fact, in the case where a global partitioning is needed, the different subsystems need to be mapped onto a homogeneous model in order to be decomposed. This operation would need a composition format able to accommodate the concepts used for the specification of the different subsystems and their interconnection.

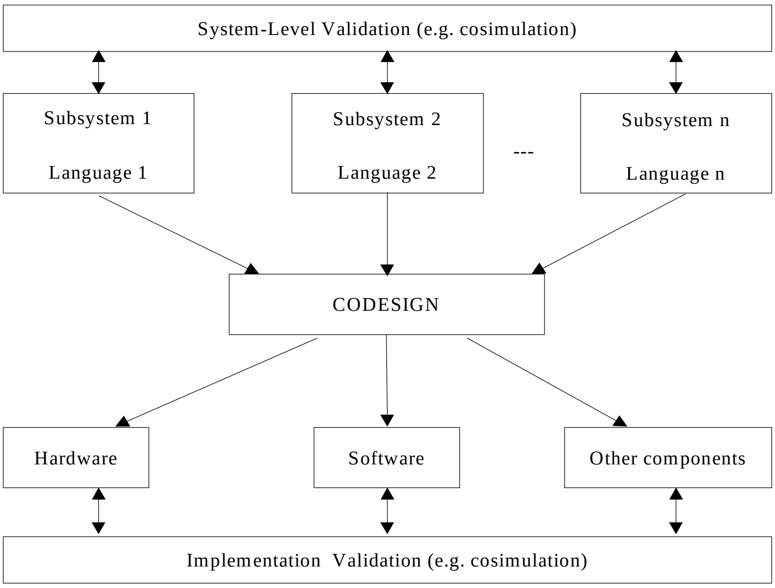


Fig. 4 Multilanguage codesign

In all cases, the key issue with multilanguage codesign is the synthesis of interfaces between subsystems. In fact, the global configuration of the system is a kind of "system-level netlist" that specifies the interconnection between different subsystems. Since different languages may be based on different concepts for data exchange, the interpretation of the link between subsystems is generally a difficult task. These issues will be discussed later in this chapter.

Only few systems in the literature allow such a codesign model. These include RAPID [64] and the work described in [54]. Both systems provide a composition format able to accommodate several specification languages. This codesign model will be detailed in section 5.

3. Design Representation For System Level Synthesis

This section deals with the design models used in codesign. The goal is to focus on the computation models underlying specification languages and architectures. The key message delivered here is that despite the proliferation of specification languages (see next section), there are only few basic concepts and models underlying all these languages.

In fact, most of codesign tools start by translating their input language into an intermediate form that corresponds to a computation model which is easier to transform and to refine. The rest of this section details the intermediate forms and the main underlying concepts and models.

3.1 BASIC CONCEPTS USED IN SYSTEM MODELS

Besides classic programming concepts, system specification is based on four basic concepts. These are concurrency, hierarchy, communication and synchronization. These are detailed in [10].

Concurrency allows for parallel computation. It may be characterized by the granularity of the computation [10] and the expression of operation ordering. The granularity of concurrent computation may be at the bit level (e.g. n-bits adder) or the operation level (e.g. datapath with multiple functional units) or the process level (multiprocesses specification) or at the processor-level (distributed multiprocessor models). The concurrency may be expressed using the execution order of computations or the data flow. In the first case, we have control-oriented models. In this case, a specification gives explicitly the execution order (sequencing) of the element of the specification. CSP like models and concurrent FSMs are typical examples of control-oriented concurrency. In the second case we have data-oriented concurrency. The execution order of operations is fixed by the data dependency. Dataflow graphs and architectures are typical data-oriented models.

Hierarchy is required to master the complexity. Large systems are decomposed into smaller pieces which are easier to handle. There are two kinds of hierarchies. Behavioral hierarchy allows constructs to hide sub-behaviors [11]. Procedure and sub-states are typical forms of behavioral hierarchies. Structural hierarchy allows to decompose a system into a set of interacting subsystems. Each component is defined with well defined boundaries. The interaction between subsystems may be specified at the signal (wire) level or using abstract channels hiding complex protocols.

Communication allows concurrent modules to exchange data and control information. There are two basic models for communication, message passing and shared memory [19]. In the case of message passing, subsystems will execute specific operations to send and to receive data from other components. In the case of shared memory, each subsystem needs to write and to read data from specific memory locations in order to exchange data with other subsystems.

Remote procedure call is a hybrid model allowing to model both message passing and shared memory [20]. In this model communication is performed through primitives that may perform memory read/write or message passing. These primitives are called like procedures and correspond to operations that will be executed by a specific remote communication unit.

Synchronization allows to coordinate the communication or information exchange between concurrent subsystems [20, 21]. There are mainly two synchronization modes; the synchronous mode and the asynchronous mode. When message passing communication is used, synchronization may be achieved using several schemes like message queues (asynchronous mode) or rendez-vous (asynchronous). In the case of shared memory more synchronization schemes are available. These include semaphores, critical regions and monitors. These may be implemented for both synchronous and asynchronous mode.

3.2 COMPUTATION MODELS

The key properties of a specification language, such as its expressiveness, derive from its underlying computation model. In fact, specification languages differ mainly on the manner they provide a view of the basic components, the link between these components and the composition of components. Basic components are described as behavior, these may be control or data oriented. The links fix the inter-module communication and the composition fixes the hierarchy.

Many classifications of specification languages have been proposed in the literature. Most of them concentrated on the specification style. The taxonomy proposed by D. Gajski [10], for example, distinguishes five specification styles: (1) state-oriented, (2) activity-oriented, (3) structure-oriented, (4) data-oriented and (5) heterogeneous. The state-oriented and activity-oriented models respectively provide a description of the behavior through state machines and transformations. The structure-oriented style concentrates on the system structural hierarchy. The data-oriented models provides a system specification based on information modeling. Chapter 2 uses another classification technique and distinguishes nine different computation models.

An objective classification of specification languages is better defined when based on the computation model rather than on the style of writing. In fact, the style of the specification reflects the syntactic aspects and cannot reflect the underlying computation model. The computation model deals with the set of theoretical choices that built the execution model of the language.

The computation model of a given specification can be seen as the combination of two orthogonal concepts: (1) the communication model and (2) the control model. The communication model of a specification language can be fit into either the synchronous or single-thread execution model, or the distributed execution model. The synchronous model is generally a one-thread-based computation, while the distributed model is a multi-thread-based computation with explicit communication between parallel processors. The synchronization arbitrates the exchange of information between processes. The control model can be classified into control-flow or data-flow. This gives the execution order of operations within one process. The synchronous execution model is suitable for mono-processor applications, while the distributed model is adequate for multi-processor applications. In a synchronous model, there is a unique global time reference and the computation is deterministic. Such an execution model is like the one found in DSP applications where data are acquired, processed, then communicated at a fixed rate and in a cyclic manner. The control-oriented model focuses on the control sequences rather than on the computation itself. The data-oriented model focus expresses the behavior as a set of data transformations.

According to this classification we obtain mainly four computation models that may be defined according to concurrency and synchronization. Figure 5 shows these classes and different languages related to these models. Most of these languages will be discussed in the next sections.

Communication Model Concurrency	Single-thread	Distributed
Control-driven	SCCS, StateChart Esterel, SML	CCS, CSP,VHDL OCCAM,SDL
Data-driven	SILAGE LUSTRE SIGNAL	Asynchronous Data flow

Figure 5 Computation models of specification languages

3.3 SYNTHESIS INTERMEDIATE FORMS

Most codesign tools make use of an internal representation for the refinement of the input specification into architectures. The input specification is generally given in a human readable format that may be a textual language (C, VHDL, SDL, JAVA, ...) or a graphical representation (StateCharts, SpecCharts, ...). The architecture is generally made of a set of processors. The composition scheme of these processors depends on the computation model underlying the architecture. The refinement of the input specification into architecture is generally performed into several refinement steps using an internal representation also called intermediate form or internal data structure. The different steps of the refinement process can be explained as a transformation of this internal model.

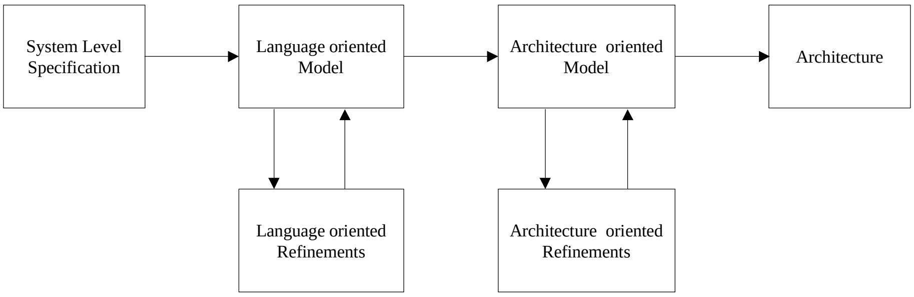

System Level Specification
Language oriented Model
Architecture oriented Model
Architecture
Language oriented Refinements
Architecture oriented Refinements

Figure 6 Language-oriented vs Architecture-oriented intermediate forms

There are mainly two kinds of intermediate forms: the first is language-oriented and the latter is architecture-oriented. Both may be used for system representation and transformations. Language-oriented intermediate forms are generally based on graph representation. They model well the concepts handled in specification languages. Architecture-oriented intermediate forms are generally based on FSM models. These are close to the concepts needed to represent architectures.

Figure 6 shows a generic codesign model that combines both models. In this case, the first codesign step translates the initial system-level specification into a language-oriented intermediate form. A set of refinement steps may be applied to this representation. These may correspond to high level transformations. A specific refinement step is used to map the resulting model into architecture-oriented intermediate form. This will be refined into an architecture.

Although general, this model is difficult to implement in practice. In fact, most existing tools make use of a unique intermediate form. The kind of selected intermediate form will restrict the efficiency of the resulting codesign tool. Language-oriented intermediate forms make easier the handling of high level concepts (e.g. abstract communication, abstract data types) and high level transformations. But, it makes difficult the architecture generation step and the specification of partial architectures and physical constraints. On the other side, an architecture-oriented intermediate form makes difficult the translation of high level concepts. However, it makes easier the specification of architecture related concepts such as specific communication and synchronization. This kind of intermediate form generally produces more efficient design.

3.4 LANGUAGE ORIENTED INTERMEDIATE FORMS

Various representations have appeared in the literature [22, 23, 24, 25], mostly based on flow graphs. The main kinds are Data, Control and Control-Data flow representations as introduced below.

3.4.1. *Data Flow Graph (DFG)*

Data flow graphs are the most popular representation of a program in high level synthesis. Nodes represent the operators of the program, edges represent values. The function of node is to generate a new value on its outputs depending on its inputs.

A data flow graph example is given in Figure 7, representing the computation *e:= (a+c)*(b-d).* This graph is composed of three nodes v_1 representing the operation +, v_2 representing **-** and v_3v representing *. Both data produced by v_1 and v_2 are consumed by v_3 . At the system level a node may hide complex computation or a processor.

In the synchronous data flow model, we assume that an edge may hold at most one value. Then, we assume that all operators consume their inputs before new values are produced on their inputs edges.

In the asynchronous data flow model, we assume that each edge may hold an infinite set of values stored in an input queue. In this model, we assume that inputs, arrivals and computations are performed at different and independent throughputs. This model is powerful for the representation of computation. However it is restricted for the representation of control structures.

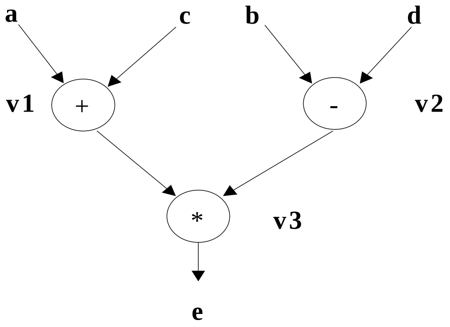


Figure 7 Example of a simple data flow graph

3.4.2. *Control Flow Graph (CFG)*

Control flow graphs are the most suited representation to model control design. These may contain many (possibly nested) loops, global exceptions, synchronization and procedure calls; in other words, features that reflect the inherent properties of controllers. In a CFG, nodes represent operations and edges represent sequencing relations.

While this kind of graph models well control structure including generalized nested loops combined with explicit synchronization statement (wait), control statements (if, case), and exceptions (EXIT), it provides restricted facilities for data flow analysis and transformations.

3.4.3. *Control Data Flow Graph (CDFG)*

This model extends DFG with control nodes (If, case, loops) [26]. This model is very suited for the representation of data flow oriented applications. Several codesign tools use CDFG as intermediate form [4] [Lycos, Vilcar].

3.5 ARCHITECTURE ORIENTED INTERMEDIATE FORMS

This kind of intermediate form is closer to the architecture produced by codesign than to initial input description. The data path and the controller may be represented explicitly when using this model. The controller is generally abstracted as an FSM. The data path is modeled as a set of assignment and expressions including operation on data. The main kind of architecture-oriented forms are FSM with data path model FSMD defined by Gajski [27] and the FSM with coprocessors(FSMC) defined in [28].

3.5.1. *The FSMD representation*

The FSMD was introduced by Gajski [27] as a universal model that represents all hardware design. An FSMD is an FSM extended with operations on data.

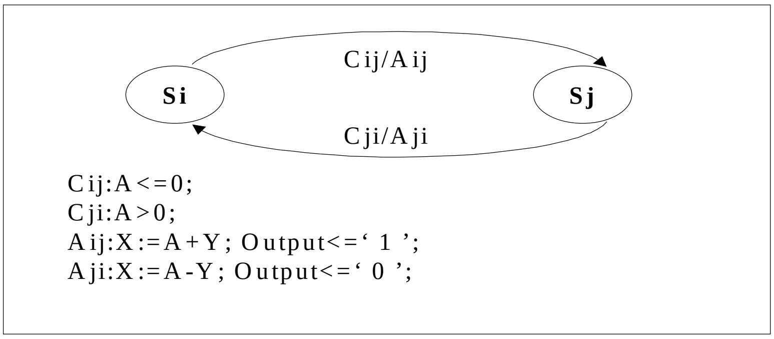
In this model, the classic FSM is extended with variables. An FSMD may have internal variables and a transition may include arithmetic and logic operation on these variables. The FSMD adds a datapath including variables, operators on communication to the classic FSM.

The FSMD computes new values for variables stored in the data path and produces outputs. Figure 8 shows a simplified FSMD with 2 states Si and Sj and two transitions. Each transition is defined with a condition and a set of actions that have to be executed in parallel when the transition is fixed.

3.5.2. *The FSM with coprocessors model (FSMC)*

An FSMC is an FSMD with operations executed on coprocessors. The expressions may include complex operations executed on specific calculation units called coprocessors. An FMSC is defined as an FSMD plus a set of N coprocessors C. Each coprocessor Ci is also defined as an FSMC. FSMC models hierarchical architecture made of a top controller and a set of data paths that may include FSMC components as shown in Figure 9.

Coprocessors may have their local controller, inputs and outputs. They are used by the top controllers to execute specific operations (expressions of the behavioral description). Several codesign tools produce an FSMC based architecture. These include COSYMA [5], VULCAN [29] and Lycos [70].



```
Cij:A<=0;
Cji:A>0;
Aij:X:=A+Y; Output<=‘1’;
Aji:X:=A-Y; Output<=‘0’;
```

Figure 8 FSMD Model

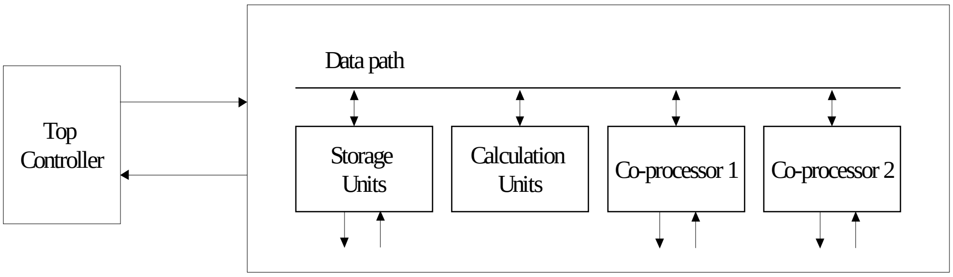


Figure 9 FSMC Architecture Model

In these codesign tools the top controller is made of a software processor and the co-processor are hardware accelerators.

3.6 DISTRIBUTED INTERMEDIATE FORMS

In order to handle concurrent processes several codesign tools make use of intermediate forms that support multithreading. At this level also intermediate forms may be language oriented or architecture oriented. A distributed language oriented intermediate form is generally made of communicating graphs [30]. The most popular format of this type is the task graph [6, 31].

In this model each node represents a simple CFG, DFG or CDFG and the edges represent execution orders that may be control oriented or data oriented. Inter task communication may follow any of the schemes listed above.

Most codesign tools that make use of task graphs, assume that all the tasks are periodic. This assumption induces that the execution time of each task is bounded and allows for all kind of rate analysis needed for task scheduling and performances estimation.

Distributed architecture-oriented intermediate forms are generally made of communicating FSMs. These generally used an extended FSM model in order to allow for data computation within the FSMs. Only few codesign tools support communicating FSMs. SpecSyn [10] and Polis [7] make use of interconnected FSMs. The model used in Polis is based on a specific kind of FSMDs, called Codesign FSMs (CFSM). In this model the communication is made at the signal level and a system is organized a DAG where each node is a CFSM. The COSMOS system is based on a communicating FSMC model called SOLAR[32]. This model allows for a generalized composition model and makes use of RPC [20] for inter-modules communication.

4 System Level Specification Languages

4.1 THE PLETHORA OF SYSTEM SPECIFICATION LANGUAGES

Specification languages were firstly introduced in software engineering in order to support the early steps of the software development [33]. The high cost of the software development and maintenance raised the need of concentrating on the specification and the requirements analysis steps. The software quality and productivity is expected to be improved due to formal verification and gradual refinement of software starting from higher level specification.

For hardware design, the first languages were aimed to the specification of computers. Von Newman used an ad hoc hardware description language for the description of its machine. DDL [34] PMS and ISP are typical examples introduced in the late 60's and the early 70's [10] for hardware specification. Since that time, a plethora of languages is presented in the literature. These are the results of several fields of research. The most productive areas are:

1. VLSI System Design: Research in this area produced what is called hardware description languages (HDLs). ISPS [35], CONLAN [36] and more recently HardwareC [37] , SpecCharts [13, 14], SpecC [12] and VHDL [38] are typical HDLs. These languages try to deal with the specific characteristics of hardware design such as abstraction levels, timing and data flow computation. The community also produced several specialized languages such as SPW [3] and COSSAP [68] for DSP systems modeling and design and several functional models.

2. Protocol specification: several languages have been created for protocol specification. In order to allow for protocol verification, these languages are based on what is called formal description technique (FDT) [39]. SDL [40], LOTOS [15] and ESTELLE [41, 42] are the main languages in this area.

   LOTOS [15] (LOgical Temporal Ordring Specification) is a formal specification language for protocols and distributed systems. It is an OSI standard. The LOTOS specification is composed of two parts:

   - A behavioral part based on the theory of process algebra.
   - A facultative part for data definition based on abstract data types.

   The formal basis of LOTOS is well defined. The specification approach consists in producing a first executable specification, then validate it, and derive an implementation.

   The SDL language [40] was designed in order to specify telecommunication systems. The language is standardized by the ITU (International Telecommunication Union) as Z.100. SDL is particularly suitable for systems where it is possible to represent the behavior by extended finite state machines.

An SDL specification can be seen as a set of abstract machines. The whole specification includes the definition of the global system structure, the dynamic behavior of each machine and the communication in between. SDL offers two forms of representations: a graphical representation named SDL-GR (SDL Graphical Representation) and a textual representation named SDL-PR (SDL Phrase Representation).

ESTELLE [42] is also an OSI standard language for the specification of protocol and their implementation. The specifications are procedural having a Pascal-like construction. ESTELLE is rather a programming language than a specification language. In fact, the ESTELLE specification includes implementation details. These details are generally not necessary during the specification phase.

ESTELLE adopts an asynchronous model for the communication based on message passing. It presents several limitations in the specification of the parallelism between concurrent processes.

3. Reactive system design: reactive systems are realtime applications with fast reaction to the environment. ESTEREL [8], LUSTRE [43] and Signal [44] are typical languages for the specification of reactive systems. Petri Nets [45] may also be included in this area.

   ESTEREL [8] is an imperative and parallel language which has a well defined formal basis and a complete implementation. The basic concept of ESTEREL is the event. An event corresponds to the sending or receiving of signals that convey data. ESTEREL is based on a synchronous model. This synchronism simplifies reasoning about time and ensures determinism.

   LUSTRE [43] particularly suits to the specification of programmable automata. Few real-time aspects have been added to the language in order to manage the temporal constraints between internal events.

   The SIGNAL language [44] differs from LUSTRE in the possibility of using different clocks in the same program. The clocks can be combined through temporal operators, allowing flexible modeling.

   StateCharts [46] is visual formalism for the specification of complex reactive systems created by D. Harel. StateCharts describes the behavior of those systems using a state-based model. StateCharts extend the classical finite state machine model with hierarchy, parallelism and communication. The behavior is described in terms of hierarchical states and transitions in-between. Transitions are triggered by events and conditions. The communication model is based on broadcasting, the execution model is synchronous.

   Petri Nets [45] are tools that enable to represent discrete event systems. They do enable describing the structure of the data used, they describe the control aspects and the behavior of the system. The specification is composed of a set of transitions and places. Transitions correspond to events, places correspond to activities and waiting states.

   Petri Nets enable the specification of the evolution of a system and its behavior. The global state of a system corresponds to a labeling that associates for each place a value. Each event is associated to a transition, firable when the entry places are labeled.

The original Petri Nets [45] were based on a well defined mathematical basis. Thus, it is possible to verify the properties of the system under-specification. However, literature reports on plethora of specialized Petri Nets which are not always formally defined.

4. Programming languages: most (if not all) programming languages have been used for hardware description. These include Fortran, C, Pascal, ADA and more recently C++ and JAVA. Although these languages provide nice facilities for the specification hardware systems, these generally lack feature such as timing or concurrency specifications. Lots of research works have tried to extend the programming languages for hardware modeling with limited results because the extended language is no more a standard. For instance, in order to use C as a specification language, one needs to extend it with constructs for modeling parallel computation, communication, structural hierarchy, interfaces and synchronization. The result of such extensions will produce a new language similar to HardwareC or SpecC (see chapter 10).

5. Parallel programming languages: parallel programs are very close to hard- ware specification because of the concurrency. However, they generally lack timing concepts and provide dynamic aspects which are difficult to implement in hardware. Most of the works in the field of parallel programming are based on CSP [47] and CCS [47]. This produced several languages such as OCCAM and Unity [48] that has been used for system specification.

6. Functional programming and algebraic notation: several attempts were made to use functional programming and algebraic notations for hardware specification. VDM, Z and B are examples of such formats. VDM [49] (Vienna Development Method) is based on the set theory and predicate logic. It has the advantage of being an ISO standard. The weaknesses of VDM are mainly the non support concurrency, its verbosity and the lack of tools. Z [50] is a predicative language similar to VDM. It is based on the set theory. Z enables to divide the specification in little modules, named "schemes". These modules describe at the same time static and dynamic aspects of a system. B [51] is composed of a method and an environment. It was developed by J.R. Abrial who participated to the definition of the Z language. B is completely formalized. Its semantics is complete. Besides, B integrates the two tasks of specification and design.

7. Structural Analysis : In order to master the development of large software applications, several design methodologies were introduced. The structural analysis was initially introduced by [Demarco]. It provides a systematic approach for structuring code and data in the case of designing large software. The key idea behind structural analysis is to decompose large systems into smaller and more manageable pieces. Several improvements of the initial structural analysis were introduced with SART. After the appearance of object oriented programming

several new analysis techniques were reported in literature. These include HOOD and OMT [71]. The latest evolution of these analysis techniques is The Unified Modeling Language (UML) [72]. The goal of UML is to gather several notations within a single language. All these techniques provide powerful tools for structuring large applications, but, they lack full executable models in order to be used by synthesis and verification tools.

8. Continuous languages : These are based on differential equations and are used for very high-level modeling of all kinds of systems. The most popular languages are Matlab [67], Matrixx [73], Mathematica [74] and SABER [75]. Chapter 4 details the characteristics of Matrixx. Most of these languages provide large libraries for modeling systems in different fields. These are very often used for DSP design, mechanical design and hydraulic design. These languages provide a large expression power and make intensive use of floating point computation which make them very difficult to use for synthesis. Additionally the intensive use of specialized libraries make them very flexible and powerful for different application domains but also make them difficult to analyze and to verify.

4.2 COMPARING SPECIFICATION LANGUAGES

There is not a unique universal specification language to support all kinds of applications (controller, heavy computation, DSP, ...). A specification language is generally selected according to the application at hand and to the designer culture. This section provides 3 criteria that may guide the selection of specification languages. These are [52]:

1. Expressive power: this is related to the computation model. The expressive power of a language fixes the difficulty or the ease when describing a given behavior

2. Analytical power: this is related to the analysis, the transformation and the verification of the format. It is mainly related to tool building.

3. Cost of use: this criterion is composed of several debatable aspects such as clarity of the model, related existing tools, standardization efforts, etc.

As explained earlier, the main components of the expressive power of a given language are: concurrency, communication, synchronization, data description and timing models.

The analytical power is strongly related to the formal definition of the language. Some languages have formal semantics. These include Z, D, SCCS and temporal logic. In this case, mathematical reasoning can be applied to transform, analyze or proof properties of system specification. The formal description techniques provide only a formal interpretation : these may be translated using a "well defined" method, to another language that has formal semantics. For instance, the language Chill [53] is used for the interpretation of SDL. Finally, most existing language have a formal syntax which is the weakest kind of formal description. The existence of formal semantics allows an easier analysis of specification. This also make easier the proof of properties such as coherence, consistency, equivalence, liveness and fairness.

The analytical power includes also facilities to build tools around the language. This aspect is more related to the underlying computation model of the language. As stated earlier, graph based models make easier the automation of language-oriented analysis and transformation tools. On the other side, architecture-oriented models (e.g. FSMs) make easier the automation of lower level transformation which are more related to architecture.

The cost of use includes aspects such as standardization, readability and tool support. The readability of a specification plays an important role in the efficiency of its exploitation. A graphical specification may be quoted more readable and reviewable than a textual specification by some designers. However, some of the designers may prefer textual specification. The graphical and textual specifications are complementary.

The availability of tools' support around a given specification language is important in order to take best benefit from the expressiveness of this language. Support tools include editors, simulation tools, proovers, debuggers, prototypers, etc.

The above mentioned criteria show clearly the difficulty of comparing different specification languages. Figure 8 shows an attempt for the classification of some languages. The first four lines of the table show the expressive power of the languages, the fifth line summarizes the analytical power and the three last lines give the usability criteria.

Each column summarizes the characteristic of a specific language:

******* : the language is excellent for the corresponding criteria

****** : the language provides acceptable facilities

***** : the language provides a little help

+ : coming feature

? : non proved star

		HDL	SDL	Statecharts Esterel	SPW COSSAP	Matlab	C, C++	JAVA, UML, ...
Expressive power	Abstract Communication	*	***	/	*	*	/	**
	TIME	***	**	**	*	***	/	*
	Computation Algorithms	***	*++	***	***	***	***	***
	Specific Libraries	HW cores	Protocols+	?	DSP	DSP Math Mechanical +++	***	??
	FSMs, Exceptions Control	**	***	***	/	???	**	**
	Analytical Power (Formal Analysis)	*	***	**	/	/	/	/
	Cost of Use (standard, learning curve, hosts)	***	***	*	*	***	***	???

Figure 8 Summary of some specification languages

VHDL provides an excellent cost of use. However, its expression power is quite low for communication and data structure.

SDL may be a realistic choice for system specification. SDL provides acceptable facilities for most criteria. The weakest point of SDL is timing specification where only the concept of timer is provided. Additionally, SDL restricts the communication to the asynchronous model. The recent version of SDL introduces a more generic communication model based on RPC. The new versions will also include more algorithmic capabilities.

StateCharts and Esterel are two synchronous languages. Esterel provides powerful concepts for expressing time and has a large analytical power. However its communication model (broadcasting) is restricted to the specification of synchronous systems**.** StateCharts provide a low cost of use. However, like Esterel, it has a restricted communication model. The expression power is enhanced by the existence of different kinds of hierarchies (activities, states, modules). Synchronous languages make very difficult the specification of distributed systems. For example, Esterel and StateCharts assume that the transitions of all parallel entities must take the same amount of time. This means that if one module is described at the clock-cycle level, all the other modules need to be described at the save level.

SPW and COSSAP are two DSP oriented environments provided by EDA vendors. The fact that they are based on proprietary language make their cost of use quite high because of lack of standardization and generic tools supporting these languages. These environments also make extensive use of libraries which make formal verification difficult to apply.

Matlab [67] is used by more than 400 000 engineers all over the world and in several application domains. This makes it a de facto standard which also reduces the cost of use. Matlab provides a good expression power for high level algorithm design. It also supports well the concept of time. The main restrictions of Matlab are related to the communication model (simple wire carrying floating values), the expression of state based computation (new versions included statechart like models) and formal semantics. The latter is mainly due to the extensive use of libraries.

C++, C provide a high usability power but they fail to provide general basic concepts for system specification such as concurrency and timing aspects. The use of C and C++ for system specification is very popular. In fact most designers are familiar with C and in many application domains the initial specification is made in C or C++. It is also very easy to build run-time libraries that extend the languages for supporting concurrency and timing. Unfortunately, these kinds of extensions are not standardized and make all the system design tools specific to a given environment.

JAVA and UML are generating lots of hope in the software community. They are also well studied for codesign. UML is just an emerging notation [72]. However Java seems to be adopted by a large section of the research community. If we exclude the dynamic features of JAVA such as garbage collection, we obtain a model with a very high expression power. Although JAVA includes no explicit time concepts , these can be modeled as timers using the RPC concept. The usability power of JAVA is still unknown. The same remark made about extending C applies for extensions to JAVA.

5 Heterogeneous Modeling and Multilanguage Cosimulation

Experiments with system specification languages [54] show that there is not a unique universal specification language to support the whole life cycle (specification, design, implementation) for all kinds of applications. The design of a complex system may require the cooperation of several teams belonging to different cultures and using different languages. New specification and design methods are needed to handle these cases where different languages and methods need to be used within the same design. These are multilanguage specification design and verification methods. The rest of this section provides the main concepts related to the use of multilanguage and two examples of multilanguage codesign approaches. The first starts with a heterogeneous model of the architecture given in C-VHDL. The second makes use of several system-level specification languages and is oriented towards very large system design.

5.1 BASIC CONCEPTS FOR MULTILANGUAGE DESIGN

The design of large systems, like the electronic parts of an airplane or a car, may require the participation of several groups belonging to different companies and using different design methods, languages and tools. The concept of multi-language specification aims at coordinating different modules described in different languages, formalisms, and notations. In fact, the use of more than one language corresponds to an actual need in embedded systems design.

Besides, multi-language specification is driven by the need of modular and evolutive design. This is due to the increasing complexity of designs. Modularity helps in mastering this complexity, promotes for design re-use and more generally encourages concurrent engineering development.

There are two main approaches for multi-language design: the compositional approach and cosimulation based approach.

The compositional approach (cf. Figure 9) aims at integrating the partial specification of sub-systems into a unified representation which is used for the verification and design of the global behavior. This allows to operate full coherence and consistency checking, to identify requirements for traceability links, and to facilitate the integration of new specification languages [33].

Several approaches have been proposed in order to compose partial programming and/or specification languages. Zave and Jackson 's approach [55] is based on the predicate logic semantic domain. Partial specifications are assigned semantics in this domain, and their composition is the conjunction of all partial specification. Wile's approach [56] to composition uses a common syntactic framework defined in terms of grammars and transformations. Garden project [57] provides multi-formalisms specification by means of a common operational semantics. These approaches are globally intended to facilitate the proofs of concurrent systems properties.

Polis, Javatime and SpecC detailed respectively in chapters 2, 10, 11 introduce a compositional based codesign approach. Polis uses an internal model called Codesign FSMs for composition. Both Javatime and SpecC use another specification language (Java and SpecC) for composition.

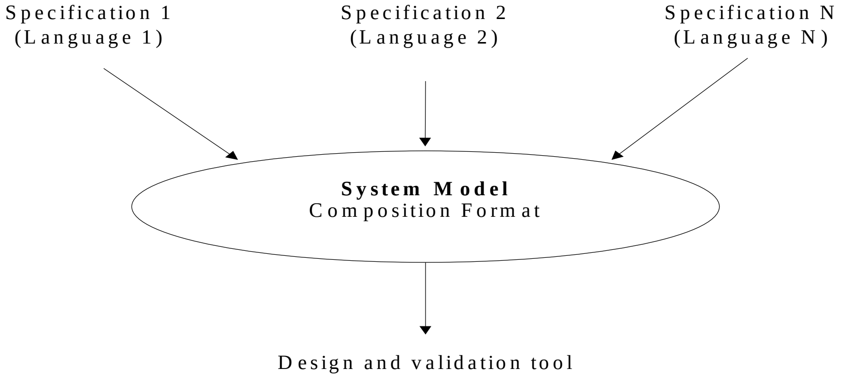


Figure 9 Composition-based multilanguage design

The cosimulation based approach (cf. Figure 10) consists in interconnecting the design environments associated to each of the partial specification. Compared with the deep specification integration accomplished by the compositional approaches, cosimulation is an engineering solution to multilanguage design that performs just a shallow integration of the partial specifications.

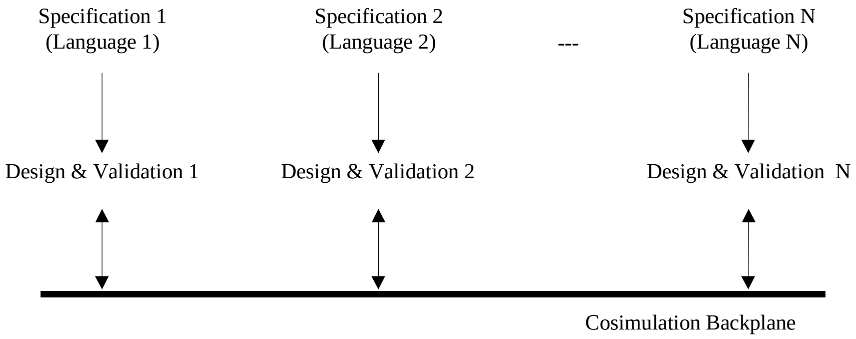


Figure 10 Cosimulation based multi language design

5.2 MULTILANGUAGE COSIMULATION

Cosimulation aims at executing several models given in different languages in a concurrent way. Cosimulation environments may be differentiated by three key factors :

a) The Engine Model

The cosimulation may be based on a single engine or on a multiple engine. The first case corresponds to the composition scheme, all the subsystems are translated into a unique notation that will be executed using a specific simulation engine. The multi engine approach corresponds to the cosimulation based on a multi-language scheme. In this case several simulators may be executed concurrently.

b) The timing model

The cosimulation may be timed or untimed. In the first case the execution time of the operation is not handled. This scheme allows only for functional validation. The time needed for computation is not taken into account during cosimulation. Inter module communication may be done only through hand shaking where the order of event is used to synchronize the data and control signal exchange. In the case of timed simulation, each block may have a local timer and the cosimulation takes into account the execution time of the computation. The time may be defined at different levels of granularity which define different levels of cosimulation, for example in the case of

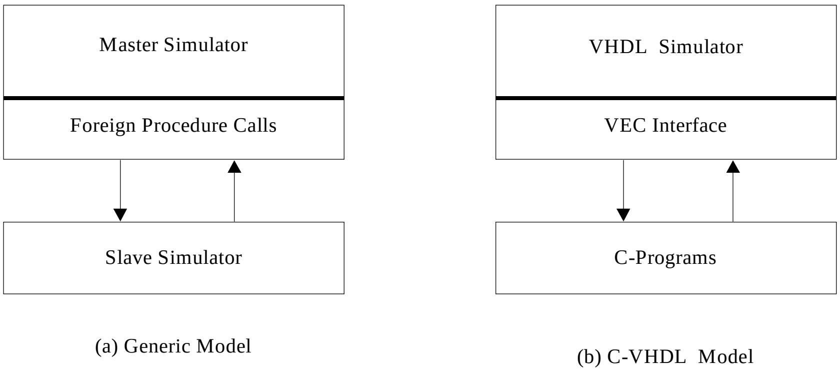
HW/SW systems there are mainly three levels of cosimulation. The instruction level, the RTL level and the physical time level.

c) The synchronization scheme :
In case the cosimulation uses a multi-engine scheme, the key issue for the definition of a cosimulation environment is the communication between the different simulators. There are mainly two synchronization modes: the master slave mode and the distributed mode.

With the master-slave mode, cosimulation involves one master simulator and one or several slave simulators. In this case, the slave simulators are invoked using procedure call-like techniques. Figure 11 shows a typical master-slave cosimulation model. The implementation of such a communication model is generally made possible thanks to:

1. The possibility to call foreign procedures (e.g. C programs) from the master simulator

2. The possibility to encapsulate the slave simulator within a procedure call



(a) Generic Model

(b) C-VHDL Model

Figure 11 Master-slave Cosimulation

Most simulators provide a basic means to invoke C functions during simulation following a master-slave model. In the case of VHDL (fig 1.11b) this access is possible by using the foreign VHDL attribute within an associated VHDL architecture. The foreign attribute allows parts of the code to be written in languages other than VHDL (e.g. C procedures). Although useful, this scheme presents a fundamental constraint. The slave module cannot work in the same time concurrently to the master module. In the case of C-VHDL cosimulation, this also implies that the C module cannot hold an internal state between two calls. Then, the C-program needs to be sliced into a set of C-procedures executed in one shot and activated through a procedure call. In fact, this requires a significant style change in the C flow, specially for control-oriented applications which need multiple interaction points with the rest of the hardware parts [58]. By using this model, the software part for control-oriented applications requires a sophisticated scheme where the procedure saves the exit point of the sliced C-program for future invocations of the procedure. Moreover, the model does not allow true concurrency: when the C procedure is being executed, the simulator is idle.

The distributed cosimulation model overcomes these restrictions. This approach is based on a communication network protocol which is used as a software bus. Figure 12 shows a generic distributed cosimulation model. Each simulator communicates with the cosimulation bus through calls to foreign procedures.

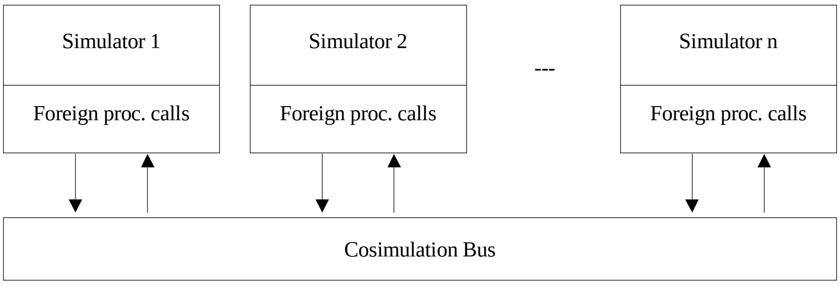


Figure 12 Distributed Cosimulation Model

The cosimulation bus is in charge of transferring data between the different simulators. It acts as a communication server accessed through procedure calls. The implementation of the cosimulation bus may be based on standard system facilities such as UNIX IPC or Sockets. It may also be implemented as an ad hoc simulation backplane [59].

In order to communicate, each simulator needs to call a procedure that will send/receive information from the software bus (e.g. IPC channel). For instance, in the case of C-VHDL cosimulation, this solution allows the designer to keep the C application code in its original form. In addition, the VHDL simulator and the C-program may run concurrently. It is important to note that the cosimulation models are independent of the communication mechanism used during cosimulation. This means that cosimulation can use other communication mechanisms than IPC (for example Berkeley Sockets or any other ad hoc protocol).

5.3 AUTOMATIC GENERATION OF INTERFACES

When dealing with multilanguage design, the most tedious and error-prone procedure is the generation of the link between different environments. In order to overcome this problem automatic interface generation tools are needed. This kind of tools take as input a user defined configuration file, which specifies the desired configuration characteristics (I/O interface and synchronization between debugging tools) and produces a ready-to-use interface [58, 60]. There are two kinds of interfaces needed : cosimulation interfaces and implementation interfaces.

Figure 14 shows a generic interface generation scheme starting from a set of modules and a configuration file, the system produces an executable model. In the case of co-simulation, the communication bus will be a software bus (such as IPC or sockets) and the module interfaces are systems calls used to link the simulators to the cosimulation bus. In the case of interface generation for implementation, the communication bus may correspond to an existing platform and the module interfaces may include drivers for software and specific hardware for other blocks [18].

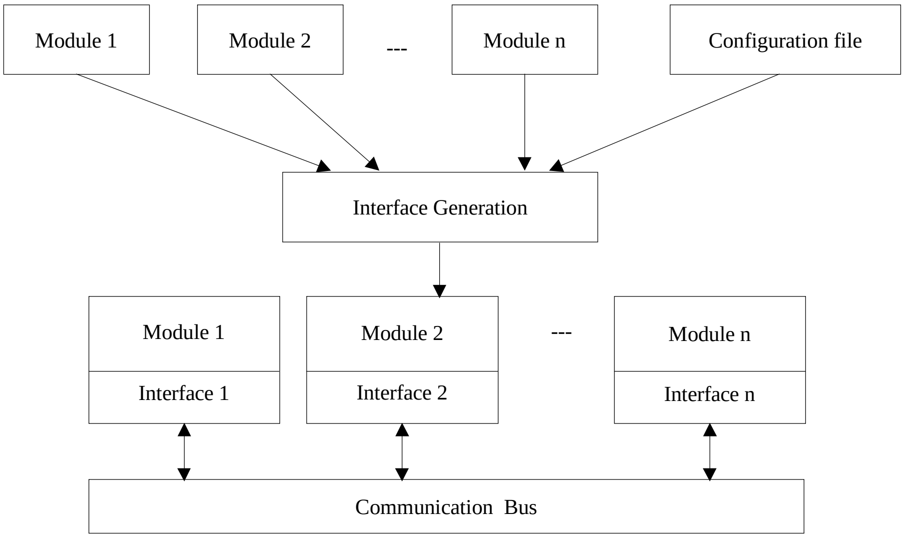


Figure 14 Intermodule interface generation for cosimulation and implementation

The configuration file specifies the inter-module interactions. This may be specified at different levels ranging from the implementation level where communication is performed through wires to the application level where communication is performed through high level primitives independent from the implementation. Figure 15 shows three levels of inter-module communication [61].

The complexity of the interface generation process depends on the level of the inter-module communication.

At the application level, communication is carried out through high level primitives such as send, receive and wait. At this level the communication protocol

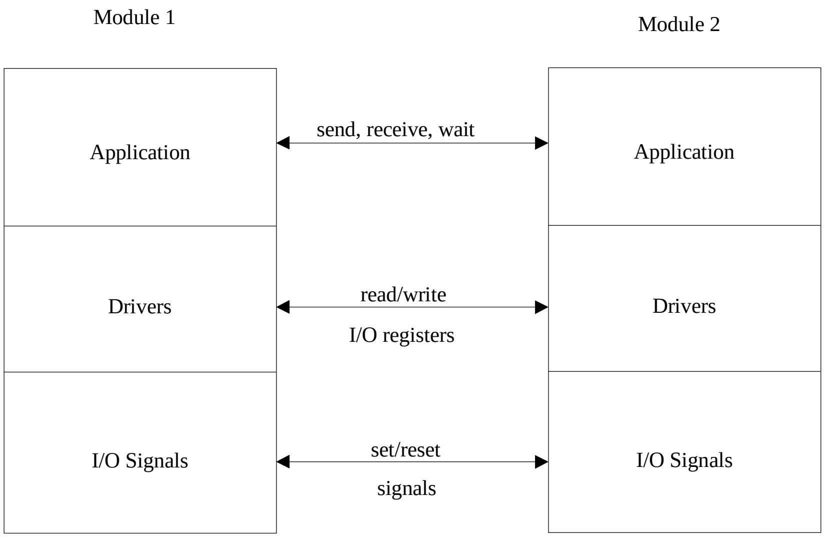


Figure 15 Abstraction Levels of Inter-module Communication

may be hidden by the communication procedures. This allows to specify a module regardless to the communication protocol that will be used later. Automatic generation of interfaces for this level of communication requires an intelligent process able to select a communication protocol. This process is generally called communication synthesis [6, 2].

The next level may be called the driver level. Communication is performed through read/write operation on I/O registers. These operations may hide physical address decoding and interrupt management. Automatic generation of interfaces selects an implementation for the I/O operations. COWARE is a typical multilanguage environment acting at this level [16].

At the lowest level, all the implementation details of the protocol are known and communication is performed using operations on simple wires (e.g. VHDL signals). The interface generation is simpler for this level.

5.4 APPLICATION 1: C-VHDL MODELING AND COSIMULATION

The design of system on chip applications generally combine programmable processors executing software and application specific hardware components. The use of C-VHDL based codesign techniques enable the cospecification, cosimulation and cosynthesis of the whole system. Experiments [63] show that these new codesign approaches are much more flexible and efficient than traditional method where the design of the software and the hardware were completely separated. More details about the design of embedded cores can be found in [76].

This section deals with a specific case of multilanguage approach based on C-VHDL. We assume that hardware/software partitioning is already made. The codesign process starts with a virtual prototype, a heterogeneous architecture composed of a set of distributed modules, represented in VHDL for hardware elements and in C for software elements. The goal is to use the virtual prototype for both cosynthesis (mapping hardware and software modules onto an architectural platform) and cosimulation (that is the joint simulation of hardware and software components) into a unified environment.

Figure 16 shows a typical C-VHDL based codesign method [58, 16]. The design starts with a mixed C-VHDL specification and handles three abstraction levels: the functional-level, the cycle accurate level and the gate-level.

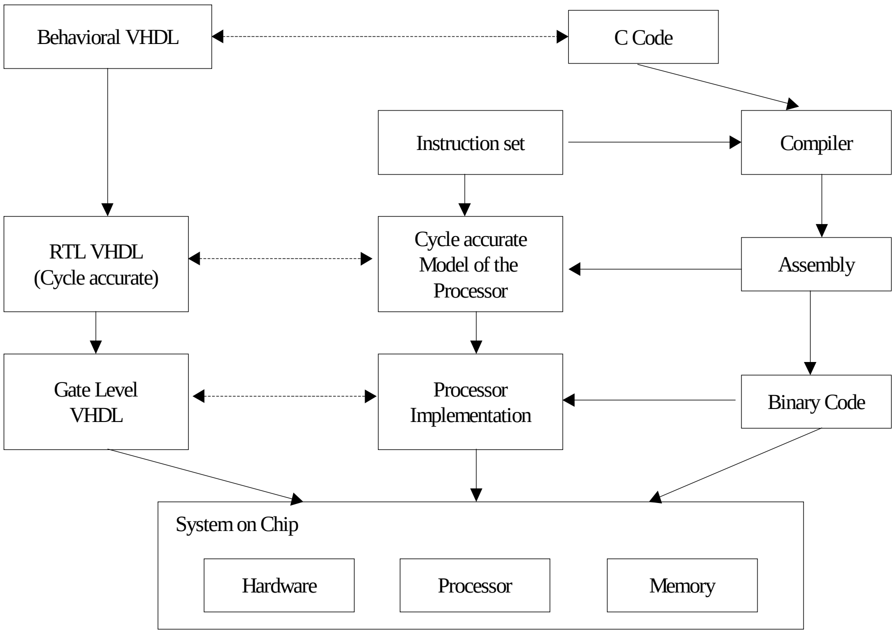


Figure 16 C-VHDL based Codesign for Embedded Processors

At the functional level, the software is described as a C program and the hardware as a behavioral VHDL model. At this level the hardware software communication may be described at the application-level. A C-VHDL cosimulation may be performed for the verification of the initial specification. The C code is executed at the workstation and the VHDL is simulated at the behavioral level. Only functional verifications can be performed at this level, accurate timing verification can be performed with the two next levels.

At the cycle accurate level, the hardware is described at the RT-level and the software at the assembly-level. A cosimulation may be performed for the verification of the behavior of the system at the cycle-level. The C code is executed on a cycle accurate model of the processor. This may be in VHDL or another software model (e.g. a C-program). At this level, the hardware/software communication can be checked at the clock-cycle level.

The cycle accurate model can be obtained automatically or manually starting from the functional level. The RTL VHDL model may be produced using behavioral synthesis. The assembler is produced by simple compilation as explained in [76]; the main difficulty is the interface refinement [16].

The gate-level is close to the implementation. The VHDL model is refined through RTL synthesis to gates. The implementation of the software processor is fixed. The processor may be a hard macro or designed at the gate-level. All communication between hardware and software is detailed at the signal level. Cosimulation may be performed to verify the correct timing of the whole system.

The C-VHDL model described in Figure 16 may handle multiprocessor design thanks to the modularity of the specification and the cosimulation model. [66] gives more details about the specification styles and interface synthesis use in a similar C-VHDL based codesign methodology.

5.5 APPLICATION 2: MULTILANGUAGE DESIGN FOR AUTOMOTIVE

This section introduces a complete design flow used for automotive design [78]. The use of electronics within cars is becoming more and more important. It is expected that the electronic parts will constitute more than 20% of the price of future cars [77]. The joint design control of various mechanical parts with electronic parts and specially micro-controllers is a very important area for automotive design. In traditional design approaches the different parts are designed by separate groups and the integration of the overall system is made at the final stage. This scheme may induce extra delays and costs because of interfacing problems. The necessity for more efficient design approaches allowing for joint design of different parts is evident. Multilanguage design constitutes an important step towards this direction. It gives the designer the ability to validate the whole system's behavior before the implementation of any of its parts.

Multilanguage system design offers many advantages including efficient design flow and shorter time to market. The key idea is to allow for early validation of the overall system through co-simulation. Figure 18 shows the methodology used to produce and to validate the initial specification for an automotive system including hardware, software and mechanical parts. The design starts with an analysis of the system requirements and a high level definition of the various functions of the system. At the top level, partitioning is done manually because we have no formal specification. The mechanical part is modeled in Matlab and the electronic part is modeled in SDL. At this stage, we obtain a multilanguage system level model given in SDL-Matlab.

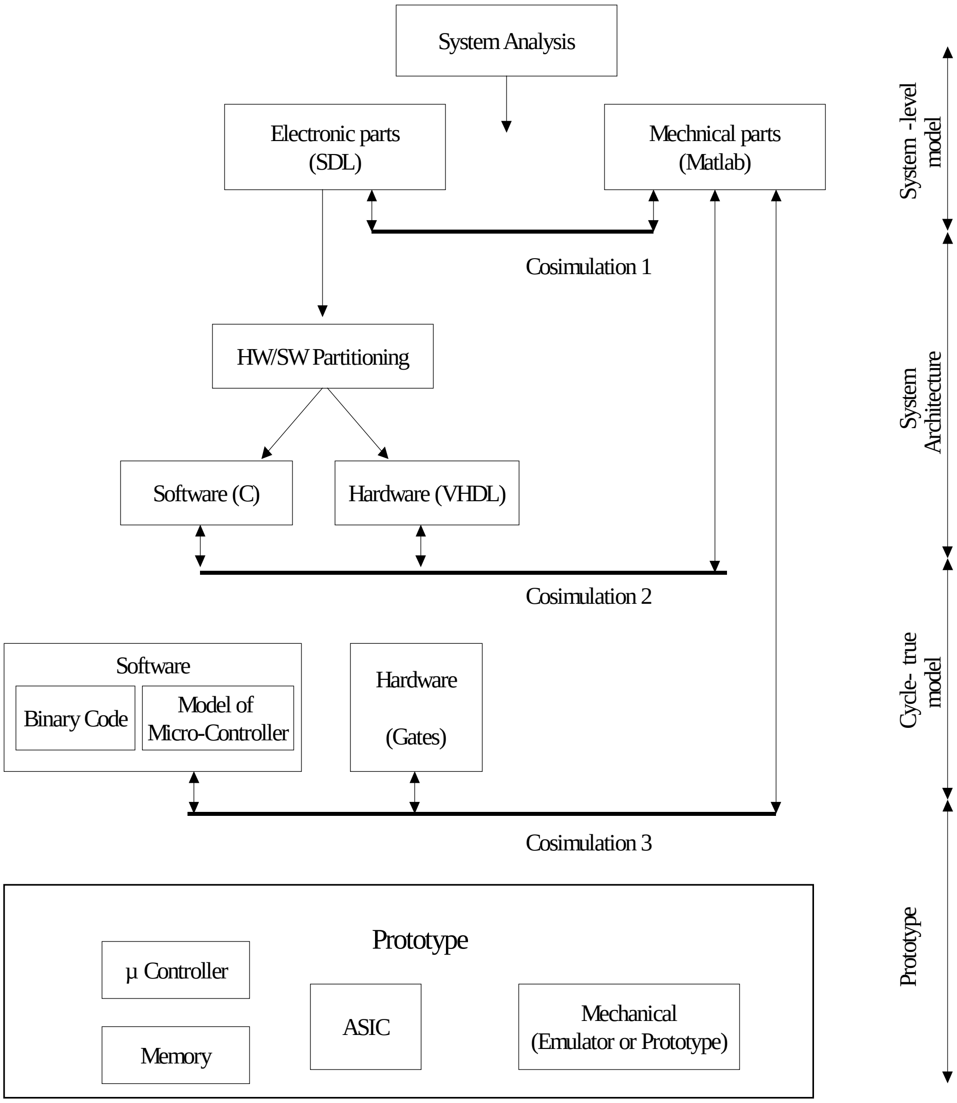


Figure 16 Multilanguage design flow for Automotive

The design flow handles three abstraction levels : the system-level, the system architecture level and the RTL or cycle accurate level.

At the system level, communication is described at the application level. The validation of the overall system may be done using system level co-simulation of SDL-Matlab models. This model may be used as a mock-up of the system in order to fix the final specification. The electronic module needs to be partitioned into hardware and software. This may be performed automatically using COSMOS-like tools [66]. This step produces a mixed hardware/software model of the electronic part.

At this stage, we obtain a system architecture Model. Hardware is modeled as behavioral VHDL, software is modeled as C-programs, Hardware/Software communication is performed through generic wires and the mechanical part remains as a Matlab model.

Cosimulation may be achieved in order to validate the partitioning and the communication protocols. Some timing verification may be achieved at this level.

At the cycle-true level, hardware is refined as an RTL model. The software is executed on a Model of the final processor and the mechanical part may be kept in Matlab. At this level all interfaces are refined to the physical level. HW/SW interfaces should include drivers in the software and some hardware adaptors to link the processor to the application. The model of the processor may be an implementation model (e.g. a gate model or a synthesizable VHDL) or a high-level model (e.g. a C program). At this level, cosimulation may be used to check timing at the clock-cycle level.

The final step in this multi-language design flow is prototyping. At this level, a prototype of the electronic system needs to be built. The mechanical part may be emulated.

Of course, this model may be extended to other languages that may cover other domains. For instance, in automotive hydraulic parts may be simulated using SABER and integrated within the design flow of Figure 16. Another extension would be to combine SDL with a Dataflow-oriented model (e.g. SPW or COSSAP) for the specification of the electronic system. Such an extension may be needed in case the design includes both control and DSP functions.

The multilanguage approach is an already proven method with the emergence codesign approaches based on C-VHDL. It is expected that for the future, multilanguage design will develop and cover multi-system specification languages allowing for the design of large heterogeneous designs. In this case, different languages will be used for the specification of the different subsystems.

ACKNOWLEDGEMENTS

This work was supported by France-Telecom/CNET, ESPRIT-OMI program under project CODAC 24139, SGS-Thomson, Aerospatiale, PSA, ESPRIT program under project COMITY 23015 and MEDEA program under Project SMT AT-403.

6. Conclusion

System-level modeling is an important aspect in the evolution of the emerging system-level design methods and tools. This includes system-level specification languages which target designers and internal format which are used by tools.

Most hardware/software codesign tools make use of an internal representation for the refinement of the input specification and architectures. An intermediate form may be language-oriented or architecture-oriented. The first kind makes easier high-level transformations and the latter makes easier handling sophisticated architectures.

The intermediate form fixes the underlying design model of the codesign process. When the target model is a multi-processor architecture, the intermediate model needs to handle multithread computation.

A plethora of specification languages exists. These come from different research areas including VLSI design, protocols, reactive systems design, programming languages, parallel programming, functional programming and algebraic notations. Each of these languages excels within a restricted application domains. For a given usage, languages may be compared according to their ability to express behaviors (expressive power), their suitability for building tools (analytical power) and their availability.

Experiments with system specification languages have shown that there is not a unique universal specification language to support the whole design process for all kinds of applications. The design of heterogeneous systems may require the combination of several specification languages for the design of different parts of the system. The key issues in this case are multilanguage validation, cosimulation and interfacing (simulator coupling).

References

1. V. Mooney T. Sakamoto and G. De Micheli. Run-time scheduler synthesis for hardware-software systems and application to robot control design. In IEEE, editor, *Proceedings of the CHDL'97*, pages 95--99, March 1997.
2. J.-M. Daveau, G. Fernandes Marchioro, and A.A. Jerraya. Vhdl generation from sdl specification. In Carlos D. Kloos and Eduard Cerny, editors, *Proceedings of CHDL*, pages 182--201. IFIP, Chapman-Hall, April 1997.
3. C. Delgado Kloos and Marin Lopez et all. From lotos to vhdl. *Current Issues in Electronic Modelling*, 3, September 1995.
4. D. HermanN., J. Henkel, and R. Ernst. An approach to the adaptation of estimated cost parameters in the cosyma system. In Proc. *Third Int'l Wshp on Hardware/Software Codesign Codes/CASHE*, pages 100--107. Grenoble, IEEE CS Press, September 1994.
5. J.Henkel and R. Ernst. A path-based technique for estimating hardware runtime in hw/sw- cosynthesis. In *8th Intl. Symposium on System Synthesis (SSS)*, pages 116--121. Cannes, France, September 13-15 1995.
6. W. Wolf. Hardware-software codesign of embedded systems. *Proceedings of IEEE*, 27(1):42--47, January 1994.
7. L. Lavagno, A. Sangiovani-Vincentelli, and H. Hsieh. *Embedded System Codesign: Synthesis and Verification*, pages 213--242. Kluwer Academic Publishers, Boston, MA, 1996.
8. G. Berry and L. Cosserat. *The esterel synchronous programming language and its mathematical semantics. language for synthesis*, Ecole Nationale Superieure de Mines de Paris, 1984.
9. G. Berry. Hardware implementation of pure esterel. In *Proceedings of the ACM Workshop on Formal Methods in VLSI Design*, January 1991.

10. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
11. D. Gajski, F. Vahid, and S. Narayan. System-design methodology: Executable-specification refinement. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC)*, pages 458--463. Paris, France, IEEE CS Press, February 1994.
12. S. Narayan, F. Vahid, and D. Gajski. System specification and synthesis with the speccharts language. In *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 226--269. IEEE CS Press, November 1991.
13. F. Vahid and S. Narayan. Speccharts: A language for system-level synthesis. In *Proceedings of CHDL*, pages 145--154, April 1991.
14. F. Vahid, S. Narayan, and D. Gajski. Speccharts: A vhdl front-end for embedded systems. *IEEE trans. on CAD of Integrated Circuits and Systems*, 14(6):694--706, 1995.
15. ISO, IS 8807. *LOTOS a formal description technique based on the temporal ordering of observational behavior*, February 1989.
16. K.Van Rompaey, D. Verkest, I. Bolsens, and H. De Man. Coware - a design environment for heteregeneous hardware/software systems. In *Proceedings of the European Design Automation Conference*. Geneve, September 1996.
17. R. Klein. Miami: *A hardware software co-simulation environment. In Proceedings of RSP'96*, pages 173--177. IEEE CS Press, 1996.
18. I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Co-design of dsp systems. In *NATO ASI Hardware/Software Co-Design*. Tremezzo, Italy, June 1995.
19. G.R. Andrews. *Concurrent Programming, Principles and Practice*. Redwood City,, California, 1991.
20. G.R. Andrews and F.B. Schneider. Concepts and notation for concurrent programming. *Computing Survey*, 15(1), March 1983.
21. G.V. Bochmann. Specification languages for communication protocols. In *Proceedings of the Conference on Hardware Description Languages CHDL'93*, pages 365--382, 1993.
22. D.C. Ku and G. DeMicheli. Relative scheduling under timing constraints. *IEEE trans. on CAD*, May 1992.
23. R. Camposano and R.M. Tablet. Design representation for the synthesis of behavioural VHDL models. In *Proceedings of CHDL*, May 1989.
24. L. Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.
25. G.G Jong. *Generalized Data Flow Graphs, Theory and Applications*. PhD thesis, Eindhoven University of Technology, 1993.
26. J.S. Lis and D.D. Gajski. Synthesis from VHDL. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378--381, October 1988.
27. D. Gajski, N. Dutt, A. Wu, and Y. Lin. *High-Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.
28. A.A. Jerraya, H. Ding, P. Kission, and M. Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL*. Kluwer Academic Publishers, 1997.

29. R.K. Gupta, C.N. Coelho Jr., and G. DeMicheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 225--230. IEEE CS Press, 1992. References 37
30. A.H. Timmer and J.A.G. Jess. Exact Scheduling Strategies based on Bipartie Graph Matching. In *Proceedings of the European Conference on Design Automation*, Paris, France, Mars 1995.
31. A. Kalavade and E.A. Lee. A hardware/software codesign methodology for dsp applications. *IEEE Design and Test of Computers*, 10(3):16--28, September 1993.
32. A.A. Jerraya and K. O'Brien. Solar: An intermediate format for system-level design and specification. In *IFIP Inter. Workshop on Hardware/software Co-Design*, Grassau, Allemagne, May 1992.
33. Alan Davis. *Software Requirements: Analysis and Specification*. Elsevier Publisher, NY, 1995.
34. J.R. Duley and D.L. Dietmeyer. A digital system design language (ddl). *IEEE trans. on Computers*, C-24(2), 1975.
35. M.R. Barbacci. Instruction set processor specifiocation (isps): The notation and its applications. *IEEE trans. on Computer*, c30(1):24--40, january 1981.
36. P. Piloty al. Conlan report. In Springer Verlag, editor, *Lecture notes in Computer Science 151*, Berlin, 1983.
37. D.C. Ku and G. DeMicheli. Hardware C - a language for hardware design. Technical Report N! CSL-TR-88-362, *Computer Systems Lab*, Stanford University, August 1988.
38. Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual*, Std 1076-1993. IEEE, 1993.
39. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J, 1991.
40. *Computer Networks and ISDN Systems*. CCITT SDL, 1987.
41. S. Budowski and P. Dembinski. *An introduction to estelle: A specification language for distributed systems*. Computer Networks and ISDN Systems, 13(2):2-- 23, 1987.
42. *International Standard, ESTELLE* (Formal description technique based on an extended state transition model), 1987.
43. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow lustre. *IEEE Tranactions on Software Enginering*, 18(9), September 1992.
44. T. Gautier and P. Le Guernic. *Signal, a declarative language for synchronous programming of real-time systems*. Computer Science, Formal Languages and Computer Architectures, 274, 1987.
45. J. L. Peterson. *Petri Net Theory and the modeling of Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
46. D. Harel. Statecharts : *A visual formalism for complex systems.* Science of Computer Programming, 8:231--274, 1987.
47. R. Milner. *Calculi for synchrony and asynchrony*. Theoritical Computer Science, 23:267--310, 1983.

48. E. Barros, W. Rosentiel, and X. Xiong. A method for partitioning unity language in hardware and software. In *Proc. European Design Automation Conference (EuroDAC)*, pages 220--225. IEEE CS Press, September 1994.
49. C.B. Jones. *Systematic software development using vdm*. C.A.R Hoare Series, Prentice Hall International Series in Computer Science, 1990.
50. J.M. Spivey. An introduction to z formal specifications. *Software Engineering Journal*, pages 40--50, January 1989.
51. J. R. Abrial. *The B Book. Assigning Programs to Meaning*. Cambridge University Press, 1997.
52. J. Bruijnin. *Evaluation and integration of specification languages*. Computer Networks and ISDN Systems, 13(2):75--89, 1987.
53. CCITT. *CHILL Language Definition* - Recommendation Z.200. May 1984.
54. M. Romdhani, R.P. Hautbois, A. Jeffroy, P. de Chazelles, and A.A. Jerraya. Evaluation and composition of specification languages, an industrial point of view. In *Proc. IFIP Conf. Hardware Description Languages (CHDL*), pages 519--523, September 1995.
55. P. Zave and M. Jackson. Conjunction as composition. *ACM trans. On Software engineering and methodology*, 8(2):379--411, October 1993.
56. D.S. Wile. *Integrating syntaxes and their associated semantics*. Technical Report RR-92-297, Univ. Southern California, 1992.
57. S.P. Reis. Working in the garden environment form conceptual programming. *IEEE Software*, 4:16--27, November 1987.
58. C.A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, and A.A. Jerraya. A unified model for co-simulation and co-synthesis of mixed hardware/software systems. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC*). IEEE CS Press, March 1995.
59. W.M. Loucks, B.J. Doray, and D.G. Agnew. Experiences in real time hardware-software cosimulation. In *Proc. VHDL Int'l Users Forum (VIUF*), pages 47--57, April 1993.
60. C.A. Valderrama, F. Nacabal, P. Paulin, and A.A. Jerraya. Automatic generation of interfaces for distributed c-vhdl cosimulation of embedded systems: an industrial experience. In *Proccedings of the 7th IEEE workshop on Rapid Systems Prototyping*. IEEE CS Press, June 1996.
61. D.E. Thomas, J.K. Adams, and H. Schmit. Model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, 10(3):6--15, September 1993.
62. D. Filo, D.C. Ku, C.N. Coelho, and G. DeMicheli. Interface optimisation for concurrent systems under timing constraints. *IEEE Trans. on VLSI Systems*, 1:268--281, September 1993.
63. P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing (Special issue on synthesis for real-time DSP*), 1994.
64. N.L. Rethman and P.A. Wilsey. Rapid: A tool for hardware/software tradeoff analysis. In *Proc. IFIP Conf. Hardware Description Languages (CHDL)*. Elsevier Science, April 1993.

65. A.A. Jerraya al., *Languages for System-Level Specification and Design* in Hardware/Software Codesign, Ed. by J. Staunstrup & W. Wolf, Kluwer 1997.
66. C. Valderrama al., COSMOS : *A Transformational Codesign Tool for Multiprocessor Architectures in Hardware/Software Codesign*, Ed. by J. Staustrup & W. Wolf, Kluwer 1997.
67. MATLAB® 5 / SIMULINK® 2 : *Mathworks Inc.* - http://www.mathworks.com
68. Synopsys. 1997 (Jan.). *COSSAP (*Reference Manuals). Synopsys
69. *Advanced Design System.[ADS], HP 1998* http://www.tmo.hp.com/tmo/hpeesof/products/ads/adsoview.html
70. Jan Madsen et al., *Hardware/Software Partitioning using the Lycos System* in Hardware/Software Codesign, Ed. by J. Staunstrup & W. Wolf, Kluwer 1997.
71. P. Mariatos, A. N. Birbas, M. K. Birbas, I. Karathanasis, M. Jadoul, K. Verschaeve, J.L. Roux, D. Sinclair. *Integrated System Design with an Object-Oriented Methodology,* Volume 7. Object-Oriented modelling, J. M. Berge, Oz Levia, J. Rouillard editors, CIEM: Curent Issues in Electronic Modelling, Kluwer Academic Publishers, September 1996.
72. H. Karathanasis, D. Karkas, D. Metafas, DECT Source Algorithmic Components, d0.1s1 *AS PIS ESPRIT project* (EP20287) http://www.ratianal.com/uml/index.shtml
73. *MATRIXx,* Integrated Systems, Inc. 1998. http://www.Isi.com/Products/MATRIXx/.
74. WOLFRAM. 1998. *Mathematica.* http://www.wolfram.com/mathematica/.
75. Saber : *Saber Technology Ltd*. - http://www.saber.bm
76. Clifford Liem. *Retargetable Compilers for Embedded Core Processors : Methods and Experiences in Industrial Applications.* Kluwer Academic Publishers. 1997
77. A. Rault, Y. Bezard, A. Coustre, and T. Halconruy. *Systems integration in the car industry*. PSA, Peugeot-Citroen, 2 route de Gizy, 78140 Velizy Villacoublay, France, 1996.
78. Ph. Le Marrec al. HW/SW and mechanical cosimulation for Automotive Applications. *RSP'98*, Belgium, 1998.