Heterogeneous System-Level Cosimulation with SDL and Matlab

Per Bjuréus*
Axel Jantsch**
**CelsiusTech Electronics, Sweden*
***Royal Institute of Technology, Sweden*

Key words: Cosimulation, SDL, Matlab, System-Level Design, Heterogeneity

Abstract: Many systems consist of a signal processing and a control dominated part. The interaction of the data processing functions and a large variety of system-level control functions are often complex and with far reaching consequences. Thus, an early analysis and assessment of this interaction in a system level model is desirable. We propose a heterogeneous cosimulation environment with Matlab for the signal processing parts and SDL for the control-dominated parts. We describe a communication and synchronisation technique that allows the natural usage of Matlab vectors which often represent data samples over time periods, rather than single events at time instances. This makes the technique both natural to use and efficient in the simulation. We describe two modes of synchronisation, head synchronisation and tail synchronisation, and the conditions under which they can be used together.

1. INTRODUCTION

In the specification and design of dedicated digital signal processing (DSP) systems, the data processing and the system level control parts have traditionally been separated. This was justified because the main problems were in the functional complexity and the tight timing constraints of the data processing, while system level control were comparably simple and were typically added later during the implementation phase. Tools like Matlab, SPW, and COSSAP aid the designer in developing the signal processing algorithms and refining them to bit true models. From there C or VHDL code is generated or written for a custom hardware or a DSP based software solution. At this level, the system control is added to the design.

Today the situation is changing, and requires a system level integration of control and data processing parts due to the following reasons:

1. While specification and implementation of DSP functions are still an important research area, it is a mature field. The integration of these functions in various configurations and with other complex control dominated functions becomes increasingly a major challenge.
2. Many products and product areas develop very fast, and it is difficult to predict the required standards and interfaces for a product when it is in the market one year after the development started. Therefore, products need a high level of flexibility and reconfigurability to adapt to many potential future situations in which the product will be used. The consequence is a complex control accommodating all different standards and variants.
3. Today technology allows us to integrate much more than the essential core functionality in a product. Many functions to enhance the user's convenience, the flexibility of usage, maintainability and online testability, etc. are included. These functions contribute considerably to system level complexity and can constitute up to 90% of the system specification documents.
4. A system level model, which includes both the data processing and the system control, is desirable to assess the interaction of these two parts in an early design phase.

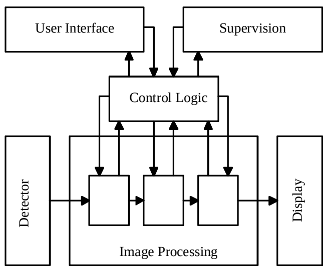

Figure 1. Image Processing System

Consider the image processing system for an infrared camera outlined in *Figure 1*. The system consists of an infrared detector, an image-processing unit, a user interface, a system supervision unit, control logic, and a display. The user interface allows the user to interact with the system and manually adjust the appearance of the image displayed. The supervision unit interfaces to external systems. It monitors detector temperature and external error conditions, controls lenses, and carries out self-tests. The control logic coordinates the system components. It reads and writes parameters to and from the image processing unit and handles user and supervision interaction. The detector scans the field of view, the image-processing unit transforms the image, and the display presents the image to the user. The image is updated several times per second (typically 25-50), which yields a data rate of several million pixels per second. The image processing is carried out in real-time and the throughput latency must be kept within certain limits, which implies strict real-time constraints. Meanwhile, the control logic is non-trivial, and may operate in several different modes, controlled by the user interface, the supervision unit, and feedback from the image processing system. It is crucial that the control logic behaves in a predictable way, and that it does not hang or enter undefined states that would impair the image processing.

When a system this complex is specified it is convenient to use different models of computation for the image processing part and the control logic. The image processing is effectively modelled using a dataflow paradigm whereas a finite state-machine model is better suited for the control logic. Since the control logic and image processing units are intimately connected to each other, the behaviour of the system is the combined behaviour of both units and their mutual interaction. The system modelling is often carried out by different design teams, each team specialised to work with one model of computation, e.g. dataflow or finite state-machines. It is therefore beneficial to provide heterogeneous system cosimulation that allows different design teams to simulate their contribution in an environment that considers the whole system. Simulation at an early design phase has several advantages. Errors are found quickly, and are easier and cheaper to correct. Feasibility and performance of different design solutions are easily explored, which results in a cheaper and more optimal design and shorter time to market.

We propose to use languages and tools, which are well established in their domains, namely Matlab [14] for the data processing parts and SDL [13] for the control dominated part. A major challenge is to integrate the timing and synchronisation concepts in a way that is intuitive to use. This is further complicated by our objective to take advantage of one of the essential ingredients of Matlab models, i.e. the usage of vectors and their transformations. Since vectors often represent data samples of a time period, their synchronisation with specific time instances and events in the SDL part has a subtle effect. We propose two synchronisation techniques, called head synchronisation and tail synchronisation, and show under which conditions they can be used together.

2. RELATED WORK

Current approaches to system modelling can be divided into two groups, homogeneous and heterogeneous models. Homogeneous models are based on a single formalism or language such as VHDL, C++, CSP, SpecChart, etc. These languages are considerably rich and can typically be used far beyond their original scope. VHDL has been proposed as system-specification language [1], sometimes by extending it with advanced features such as communication facilities [15] or object-oriented concepts [4]. Similar attempts have been put forward for popular software languages such as C [3], C++ [2], Java [5][6], or SDL [7][8]. However, such homogeneous solutions come at a price. A language, which is well established in one community, is not always well received in another community. There are both accidental and essential reasons for this. The investment in a given language in terms of tools, competence, and existing designs is often so enormous, that an abrupt switch to another language cannot be justified. Also, the modelling concepts of general-purpose languages such as VHDL and C++ are not always a perfect match to the concepts of a given application problem.

For the two domains of control-dominated systems and signal processing, it is difficult to find a language that naturally accommodates both worlds. For these reasons heterogeneous frameworks have been proposed, those build on existing models and languages and devise techniques to integrate them. A very general and most influential framework is Ptolemy [9]. Ptolemy defines several models of computation such as discrete event or data flow domains. It provides a general mechanism for communication between different domains. A mechanism for communication and synchronisation between data flow and discrete event models has been implemented in Ptolemy, which transforms each single event on the border between the two domains. If we adopt this approach for the integration of SDL and Matlab, we would essentially lose the powerful vector handling in Matlab, which is both a user convenience and key to simulation efficiency. Hence, we chose to develop an alternative technique that avoids this disadvantage. Note, that our technique could be implemented in Ptolemy also, but for our particular purpose of integrating SDL and Matlab models, a more specific and less general solution is easier to realise.

Similarly, in CoWare [10] no mechanism is provided to communicate vectors in a synchronised way between different domains, which are described in C++, VHDL, and DFL (Data flow language). The communication is based on remote procedure calls, which can communicate any type of data, but the time attributes of this data have to be dealt with explicitly in the models by the designer.

VCI [11] is a cosimulation back plane system, which allows running several simulation engines concurrently. Marrec et al. [12] use it to provide an environment for VHDL and Matlab cosimulation in an untimed and a timed mode. The untimed mode does not utilise any timing information and is for functional validation only. The timed mode operates on a cycle-true timing model with concrete architectural components such as microprocessors. Our approach also allows an untimed functional simulation between SDL and Matlab. However, the timing model is more abstract than the timed mode in [12] because it is based on the timing behaviour of the input data, not on implementation components. Thus, it allows including the timing information of the input signals in the functional simulation, and based on this, facilitates the derivation of timing constraints for implementation components.

As a summary we can conclude that our proposal, in contrast to other approaches, addresses the problem of integrating the timing behaviour of input signals into a functional cosimulation of control dominated and data transformation parts modelled in SDL and Matlab, respectively.

3. HETEROGENEOUS SYSTEM MODELING

Our target applications are typically embedded systems with a known interface towards their environment. The system is divided into a set of subsystems to make the system manageable with respect to size and complexity. A subsystem is modelled as a set of processes that operate concurrently and interact with each other. We consider two types of processes, control processes and dataflow processes. A control process interacts with its environment by exchange of control signals. Such a process has a state that changes over time as the process reacts to incoming control signals. A dataflow process is dominated by transformations of streams of data. A dataflow process consumes and produces data streams at a fixed rate. Streams flow from one dataflow process to another. The control processes and dataflow processes are allowed to interact with each other by exchange of control signals.

SDL employs an extended finite state-machine (EFSM) model of computation and is suitable for control systems modelling. The heterogeneous system specification is written using SDL at the top-level and to describe the structural hierarchy. SDL processes communicate with each other asynchronously through infinite FIFOs. The communication may or may not have a delay. In Matlab, data is transformed continuously from input stream to output stream. Streams are modelled as vectors or matrices, and transformations are modelled as functions with input and output parameters. All information exchange between the environment and the function must be passed as parameters.

3.1 Event Model

Processes communicate with each other by passing messages; this applies both to control and dataflow processes. A message may contain data or it may be empty. A message is associated with an event. An event consists of a message, the source and destination address of the message and a time stamp when the event occurs. The events are globally ordered in the system by the time stamp. An event that contains an empty message is referred to as a notification event. A dataflow message is referred to as a frame and the associated event is referred to as a continuous event. A control message is referred to as a signal and the associated event is referred to as a discrete event. Frames are used to represent streams and thus a frame always contains data and has duration. The frame data consists of a number of samples, which are the atomic elements of the stream. A stream is characterised by its sampling frequency, which specifies number of samples per time unit. Parameters can be passed to the dataflow model as control signals from the SDL model, and parameters can be passed back to the SDL model as status signals.

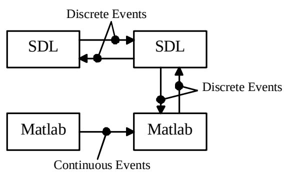

Figure 2. Event Model Outline

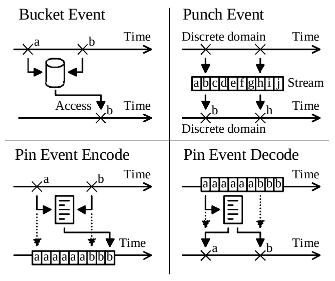

Figure 3. Event Types

Figure 2 shows the relation between process models and event types. The dataflow processes, specified in Matlab, communicate with other dataflow processes using continuous events whereas the control processes, written in SDL communicate both with control processes and dataflow processes using discrete events.

Discrete events are instantaneous and have no duration. The message of a discrete event is a signal that is allowed to carry a value. Discrete events are generated externally by the environment of the system and internally by the processes in the system. In SDL, timers are available that can be set for a timeout that causes a discrete event. Simulation time can be accessed during simulation and is used for synchronisation of the control and dataflow processes.

A stream often has an analogue source that can be measured by an electronic device. Examples of continuous streams are audio, video, microwaves, temperature etc. The system cosimulation targets embedded digital systems, and therefore only sampled streams are dealt with. A sampled stream is continuous since the samples in a stream are associated with a time and duration. At any time instant a unique sample is valid, i.e. the time and duration of the samples in a stream cover all time without overlapping. The continuously sampled stream is associated with a sample frequency, which specifies the duration of the samples in the stream. Different streams may have different sampling frequencies, but the sampling frequency of any one stream is constant.

An efficient way to represent streams is by using vectors whose elements correspond to samples. Using vectors allows large amounts of data to be processed efficiently. An individual sample is very similar to the discrete event presented earlier. However, an essential difference is that the sample has duration whereas the discrete event is instantaneous.

The Matlab models are assumed to operate on frames with fixed duration. A frame is provided to the Matlab model as a vector, where the order of the elements, i.e. the vector index, decides the time during which each sample is valid. A key concept when streams are converted to frames is stream splitting. When a stream is split, it becomes an ordered set of frames, where each frame is associated with a continuous event. Continuous events represent streams in the specification, but outside the Matlab environment they are modelled as SDL signals using discrete events.

When parameters are passed from the control model to the dataflow model and back as control and status signals, there are different ways to treat the signal depending on simulation timing.

Figure 3 depicts three different event types used to synchronise the message passing. The event types are referred to as bucket, pin, and punch events.

Bucket events are events that are collected into a “bucket”. Each signal has its own slot and only the latest signal is stored, without a time stamp. When the bucket is accessed, the signal value is read and associated with an event that occurs at the time of the access. If the bucket is accessed and no event has occurred since the previous access, a default value is read.

Pin events are collected in a list. All events are recorded and marked with a time stamp. When the pin event list is accessed, it is translated between signals and a continuous stream vector with a predefined sampling rate. To encode a vector, signal values are “pinned” to elements in the vector and elements between “pins” are set to the value of the previous signal. To decode a stream vector with a known sampling rate, pin events are extracted and collected in an event list.

Finally, there is the punch event, which is used solely to “punch” out a value from a stream. The punch event immediately triggers a new discrete event containing the value of the stream at that particular time instant.

3.2 Synchronisation

A stream event is treated as a signal in the SDL simulator. The event is used to notify the dataflow models that data is available. The problem is to decide when the stream event should occur. First and maybe most naturally, one may cause the event when the first sample in the frame begins to occur. This will be referred to as head synchronisation, and can be viewed as information about the future. One may also do the opposite, and let the stream event occur when the last sample in the frame ceases to occur (which equals the time that the first element in the next frame begins to occur). This is referred to as tail synchronisation, and can be viewed as information about the past. “Begin to occur” and “cease to occur”, relates to the fact that a sample has duration.

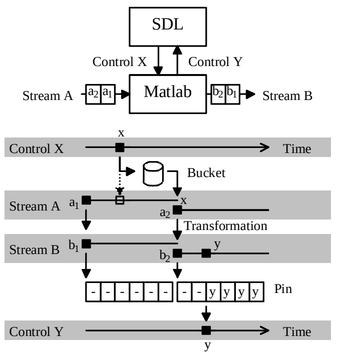

Figure 4. Head Synchronisation

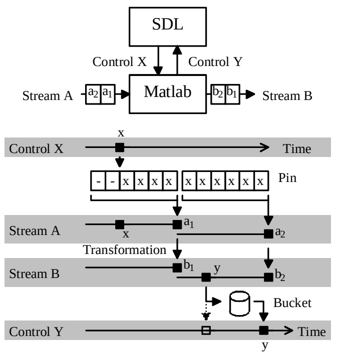

Figure 5. Tail Synchronisation

Figure 4 and *Figure 5* illustrate the head and tail synchronisation concept, which will be described in detail throughout the remainder of this section.

Head Synchronisation: When data is fed to a dataflow model, i.e. a Matlab function, the frame represents future data, and we expect the function to calculate new future data as its output, which propagates in the system. This means that signals that occur during the duration of the frame cannot be taken into account, since they have not yet occurred. This can be resolved by using the bucket event. During the duration of the frame, all input events are collected in a bucket, which is accessed with the next function call. The output from the dataflow model consists of frame and parameter vectors. The parameter vectors are translated to pin event lists, and the events can be transmitted to the rest of the system at the time they occur. *Figure 4* illustrates the head synchronisation technique. Filled squares represent the time of the events and a line connected to a stream event represents the duration of the frame. In the figure time extends from left to right. The first thing that happens is that the Matlab model receives frame a_1, and immediately transforms a_1 into b_1. The next thing that happens is that a control signal x is transmitted from the SDL model to the Matlab model. This signal cannot influence the transformation of a_1 to b_1 although it appears within the duration of a_1. Therefore, it is stored as a bucket event. Next, frame a_2 reaches the Matlab model, and the signal x can be accessed from the bucket and taken into account when transforming a_2 into b_2. The transformation of a_2 into b_2 yields a status signal y, which is returned in a vector that can be translated into a pin event list. The pin event list is used to transmit signal y from the Matlab model to the SDL model at the time it occurred.

Although head synchronisation does not allow input parameters to influence the calculation immediately, the stream samples can be accessed instantaneously with a punch event. The stream that is punched must be the input stream or the output stream of the dataflow model. Since the output stream already has been calculated, the punch event cannot alter the data.

Head synchronisation is most useful for dataflow models that lack input parameters and produces output parameters. A good example of such a process is data stream sources.

Tail Synchronisation: Tail synchronisation passes continuous events when they cease to occur. This means that input parameters can be collected in a pin event list, which can be translated to a vector and passed to the Matlab function. Output parameters cannot be sent when they occur, because when the function is called and the output parameter is calculated, that time has already passed. Thus, the output parameters are collected as bucket events. All output parameters are sent simultaneously after the function call.

Figure 5 shows the tail synchronisation technique. Filled squares still represent events, and lines connected to stream events represents frame duration, note however that the frame duration extends to the left of the event as opposed to head synchronisation. The first thing that happens is that signal x is transmitted from the SDL model to the Matlab model. The signal is collected in a pin event list. Next, frame a_1 reaches the Matlab model, the pin event list is translated to a vector that accompanies a_1 in the transformation, and the transformation of a_1 into b_1 is carried out. Finally, frame a_2 reaches the Matlab model and a_2 is transformed into b_2, which yields a status signal y that is collected in a bucket. The signal appears as a signal transmitted from the Matlab model to the SDL model at the time the transformation is carried out.

Since the input frames are passed to the dataflow model when they have ceased to occur, punch events are not allowed in tail synchronisation.

Tail synchronisation is most useful for dataflow models with input parameters that influence the data stream instantly. It is less useful for models that produce parameters.

Delay Requirements: It is possible to mix head and tail synchronisation in a system model, but certain rules must be obeyed when converting a stream from head to tail synchronisation and vice versa. The duration, or frame size, used by a dataflow process is denoted λt seconds. The computation delay, i.e. latency, of a dataflow process is denoted δt seconds. This means that a process collects data during λt seconds, and the delay from data input to output, or latency, is δt seconds.

Figure 6 shows two communicating dataflow processes where data flows from P1 to P2 via a signal route in SDL. P1 has duration λt_1, and P2 has duration λt_2. The computation delay of P2 is δt_2. Note that although the duration of a process requires that the input and output frame duration is equal, the duration of different processes operating on the same stream may differ. This situation is resolved using a buffer on each input port of dataflow processes, which provide means to concatenate or split incoming frames to match the process duration.

Table 1. Delay Requirements

Relation	P1	P2	Delay Req.
$\lambda t_1 = \lambda t_2$	Head	Head	None
	Head	Tail	None
	Tail	Head	$\delta t_2 \geq \lambda t_1$
	Tail	Tail	None
$\lambda t_1 < \lambda t_2$,	Head	Head	$\delta t_2 \geq (n-1)\ \lambda t_1$
$n\lambda t_1 = \lambda t_2$	Head	Tail	None
	Tail	Head	$\delta t_2 \geq n\lambda t_1$
	Tail	Tail	None
$\lambda t_1 > \lambda t_2$,	Head	Head	None
$\lambda t_1 = n\lambda t_2$	Head	Tail	None
	Tail	Head	$\delta t_2 \geq n\lambda t_2$
	Tail	Tail	$\delta t_2 \geq (n-1)\ \lambda t_2$

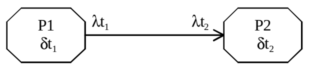

Figure 6. Process Communication

Table 1 shows the computation delay requirements depending on frame size for all combinations of synchronisation between two dataflow processes. If several streams are connected to the same dataflow process, all requirements for all combinations of input and output streams must be met.

4. IMPLEMENTATION

Wrappers are used to wrap up the dataflow components in the SDL specification in order to provide the synchronisation and data exchange during simulation.

Figure 7 depicts the relation between environments and wrappers used in simulation. The SDL model contains dataflow processes, each of which has a customised SDL wrapper. The SDL wrapper has access to a C-wrapper implemented as a set of C functions, which ultimately calls the Matlab engine through a set of C-library functions.

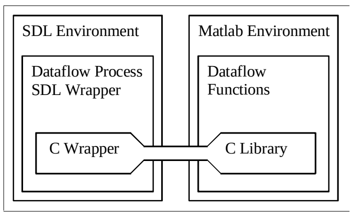

Figure 7. Wrappers and Environment

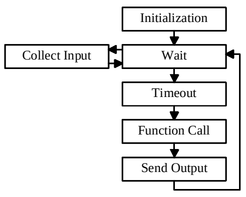

Figure 8. Wrapper Overview

Figure 8 shows a schematic overview of the SDL wrapper, which is implemented as a state-machine that handles input and output of streams and signals. The SDL wrapper declares a number of variables that are used throughout the signal processing. Two timers are declared: one that controls the dataflow function call, and one for output event synchronisation. Two variables are declared for each input stream signal, one holding the duration of the input frame, *t*, and one holding the ratio, *r* between the input stream frame duration and the duration of the process. The ratio is used to verify that all data is available before the function is called.

There is only one active Matlab engine, which executes all the dataflow processes. Vectors and variables, which are used only by Matlab functions, are never passed to SDL processes, which in turn allows for efficient simulation. Thus, typically only identifiers denoting the frames but not the frames themselves are passed between the SDL and Matlab models.

During initialisation, the Matlab engine is started, the duration *D* of the process and the sample rate of every stream, connected to the process, is determined. Those configuration parameters are defined in the Matlab environment. The frame ratio for each stream is derived from the sample rate of the stream and the duration of the model. After initialisation, the process enters the “Wait” state. Finally, the timer has to be set to give a timeout when the process is supposed to call the dataflow function for the first time.

This timeout should occur at time *now* for a process with head synchronisation, and at time *D+now* for a process with tail synchronisation.

In the SDL wrapper there is a branch from all states for every input stream signal. When an input stream event occurs, a stream event counter is increased by the ratio between the input stream frame size and the duration of the process. This implies that a frame larger than the duration of the dataflow model is split into several frames, and frames smaller than the duration of the dataflow model are concatenated into larger ones. After the dataflow function has been called, the output streams are made available for receiving dataflow functions within the Matlab environment and a notification signal is issued for each output signal in the SDL environment.

For all input discrete event signals, there is a branch from all states in the SDL wrapper. When an input signal arrives, the time it arrives is recorded and stored along with the signal value in a signal list. The list structure is part of the C wrapper, which allows adding a signal to the end of the list and removing a signal from the beginning of the list. In addition to these functions, the list can be encoded to or decoded from a Matlab vector according to the pin event-handling scheme.

5. CONCLUSION

Assuming that a heterogeneous modelling style is beneficial for todays and tomorrows heterogeneous systems, we have proposed a cosimulation technique, which accommodates data flow and control intensive parts in a system, based on Matlab and SDL. The technique allows the natural and intuitive usage of Matlab vectors. The challenge of synchronisation between these two worlds has been addressed with two different synchronisation modes, head and tail synchronisation, which can be combined under certain conditions. Our model only considers timing imposed by environment constraints on streams and events, but not delays of implementation components. Thus, it is used for architecture independent modelling and facilitates the derivation of timing constraints for the implementation. An experiment that demonstrates the technique has been performed, the results were reported in [16]. It has been omitted here due to lack of space.

Future work will apply the technique to larger applications. Furthermore, a link to the implementation will be established by deriving timing constraints and estimating performance properties of implementation variants.

Acknowledgements: The Telelogic Tau tool suite was used for SDL entry and simulation, courtesy of Telelogic AB, Sweden.

6. REFERENCES

[1] W. Ecker, "Using VHDL for HW/SW Co-Specification", pp. 500 - 505, European Design Automation Conference, September 1993.

[2] Bill Lin, "A System Design Methodology for Software/Hardware Co-Development of Telecommunication Network Applications", Proceedings of the Design Automation Conference, 1996.

[3] R. Ernst and J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware extraction", Proceedings of the International Workshop on Hardware-Software Co-Design, September 1992.

[4] Peter J. Ashenden, Philip A. Wilsey, and Dale E. Martin, "SUAVE: Extending VHDL to Improve Data Modeling Support", IEEE Design & Test of Computers, pp. 34-44, April-June 1998.

[5] Rachid Helaihel and Kunle Olukotum, "Java as a Specification for Hardware-Software Systems", Proceedings of the International Conference on Computer-Aided Design, 1997.

[6] James Shin Young, Josh MacDonald, Michael Shilamn, Abdallah Tabbara, Paul Hilfinger, and Richard Newton, "Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement", Proceedings of the 35th Design Automation Conference, 1998.

[7] Jean-Marc Daveau, Gilberto Fernandes Marchioro, Carlos Alberto Valderrama, and Ahmed Amine Jerraya, "VHDL generation from SDL specifications", Proceedings of Computer Hardware Description Languages, April 1997.

[8] Bengt Svantesson, Shashi Kumar, Ahmed Hemani, "A Methodology and Algorithms for efficient interprocess communication synthesis from system description in SDL", Proceedings of the IEEE International Conference on VLSI Design, 1998.

[9] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", International Journal of Computer Simulation, 1992.

[10] Ivo Bolsens, Hugo de Man, Bill Lin, Karl van Rompaey, Steven Vercauteren, and Diederik Verkest, "Hardware/Software Codesign of Digital Telecommunication Systems", Proceedings of the IEEE, vol. 85, no. 3, pp. 391 - 418, March 1997.

[11] C. Valderrama, A. Changuel, P. Raghavan, M. Abid, T. Ismail, and A. Jerraya, "A Unified Model for Cosimulation and Cosynthesis of Mixed Hardware/Software Systems", Proceedings of the European Design and Test Conference (ED&TC95), 1995.

[12] P. Le Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, and O. Cayrol, "Hardware, Software and Mechanical Cosimulation for Automotive Applications", Proceedings of the Ninth International Workshop on Rapid System Prototyping, pp. 202 - 206, 1998.

[13] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma, SDL - Formal Object Oriented Language for Communicating Systems, Prentice Hall Europe, 1997.

[14] MATLAB: High-performance Numeric Computation and Visualization Software. User’s Guide, 1992.

[15] Petru Eles, K. Kuchcinski, Zebo Peng, and A. Doboli, "Hardware/software partitioning of VHDL system specifications", European Design Automation Conference (Euro-DAC), 1996.

[16] Per Bjuréus, Axel Jantsch, “Heterogeneous System-Level Cosimulation with SDL and Matlab”, Proceedings of Forum on Design Languages (FDL), 1999