MASCOT: A Specification and Cosimulation Method Integrating Data and Control Flow

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

Abstract

We integrate data and control flow at the system specification level, using the two specialized and well established languages Matlab and SDL. For this we provide a modeling technique, which integrates the timing concepts and allows synchronization of vector-based computation with event based state transition. The technique is supported by a library of wrappers and communication functions, which has been implemented to make cosimulation easy to use and almost transparent to the user. A methodology formulates the rules to use the modeling technique, to partition the system, and to select communication modes. A complex industrial example illustrates the modeling technique and the methodology, and shows the efficiency of the Matlab-SDL cosimulation.

1. Introduction

The desire to specify and simulate heterogeneous embedded real time systems at a high abstraction level in a multilanguage environment is recognized in many application areas such as multimedia [4], automotive [3], avionics [5], mechatronics [2], and telecommunication [6]. In all these application domains, we observe both a need for implementation neutrality in the specification model and the necessity to model different parts in conceptually different ways. Implementation neutrality is required to allow for thorough design space exploration [11]. Different parts of a heterogeneous system, such as control flow, data flow, analog and mixed signal components, must be modeled in different ways due to different problem characteristics but also because of traditionally separate design flows, languages and competence of engineers.

For digital embedded real-time systems with complex control and high data throughput we address the problem of co-specification and co-simulation of dataflow and control dominated parts with the MASCOT (Matlab And SDL Codesign Techniques) methodology. It describes how to model and simulate a heterogeneous system specification based on an abstract communication mechanism [7], which, to a large extent, can be implicitly derived from the system specification. For the two parts of control and dataflow, we adopt different languages, SDL ([9],[10]) and Matlab. Both are specialized in their respective domains, well established in their user community and they enjoy sophisticated tool support. They allow the specification at the system level without bias towards implementation. Our guiding principle has been simplicity for the user. Consequently, MASCOT can be applied without restricting the involved languages, without new formalisms, and with minimal design overhead. It leads to a specification that:

- Is complete and unambiguous, rendering a unique interpretation, which governs a correct implementation.
- Can be simulated, providing specification validation that minimizes the number of changes made during implementation and encourages specification maintenance.
- Clearly defines the interface between design domains, eliminating “twilight zones” in the specification.

MASCOT consists of two parts: (1) a modeling and co-simulation technique, which integrates the different timing concepts and allows the synchronization of vector-based computation in Matlab with event based state transition in SDL; (2) a methodology, which provides rules for partitioning a system into SDL and Matlab parts and for selecting communication modes between the two. The communication mode defines synchronization policy, timing resolution, and simulation granularity, and greatly influences simulation speed and accuracy.

Thus, MASCOT provides a technique to glue specifications together, whereas design is still carried out in the appropriate domain for different parts. Design space exploration across the domain boundary must be done at the specification level, when the SDL-Matlab model is developed. The design and implementation is done separately for the two parts with the established methodologies and tools. The paper is organized as follows, the next section is dedicated to related work, section 3 introduces an example that will be used throughout the rest of the paper. Section 4 describes the conceptual modeling technique, section 5 the proposed methodology in a step-by-step fashion, and in section 6 an industrial case study illustrates the techniques presented.

2. Related work

The CoWare approach [6] offers both homogeneous and heterogeneous system specification and allows for

system simulation at different abstraction levels. For heterogeneous specifications the languages C, DFL, VHDL and Verilog are supported to day. All of them are implementation rather than application oriented languages and therefore represent a bias towards a specific implementation technology. In contrast, a specification based on application-oriented languages such as Matlab or SDL allows better trade-off analysis of application relevant parameters and a better design space exploration during the design process. However, synthesis is harder with these languages because more design decisions must be made.

Ptolemy [8] is a general framework that connects a large number of design domains and allows for heterogeneous cosimulation of discrete event and dataflow models. The current implementation of the communication mechanism between discrete event and dataflow domains is based on a technique that transmits samples individually, which is slow.

Kenter et al. [4] propose a design methodology that is based on the Felix VCC environment, which allows for functional simulation. It targets IP block composition, architecture mapping, and implementation. The YAPI communication is implemented as a handshake protocol in ECL, which is an expensive (in terms of simulation speed) communication mechanism.

The MUSIC methodology [2] hosts a set of tools and languages and provides heterogeneous system specification and cosimulation. The heterogeneous specification is assembled using SOLAR [5], and cosimulation is performed concurrently on simulators that can be distributed throughout a network. However, there is no built-in synchronization between the simulators. Also, unless a time coordinator is introduced, there is no way to synchronize stimuli to two different simulators, which restricts the specification of a simulation scenario.

The SONORA framework [1] introduces a model-based Codesign methodology. For specification in the discrete domain, DEVS-Java and possibly StateCharts is used. Components in the continuous time domain are modeled with Difference Equation System Specification (DESS). A behavioral simulation technique is proposed, but to the best of our knowledge, to date the SONORA framework has not yet been fully implemented and no results have been reported. Currently, no link exists between the discrete event and continuous domains, and no cosimulation mechanism has yet been developed.

Other heterogeneous modeling techniques have been proposed, but none addresses the problem in the same way as MASCOT. Some propose new formalisms (e.g. [1],[4]), which we consider a disadvantage given the huge investment in existing languages and tools. Others use implementation-biased languages (e.g. CoWare), which do not support well the system level trade-off analysis. None of the published approaches addresses the synchronization and communication in a mixed event and vector-based computation environment. In general frameworks, such as CoWare and Ptolemy, our technique could also be implemented. However, it has not yet been done and it is not clear, if the result would not become complicated and compare unfavorably to the simplicity and convenience of MASCOT for experienced SDL and Matlab users.

3. Example

Throughout this paper, we will refer to a single example to illustrate the proposed methodology; it is intentionally small and somewhat artificial. The example does not demonstrate the advantages of using SDL and Matlab and is not primarily intended to motivate the methodology but rather to explain it. A motivating example will be thoroughly explained in section 6.

The example is a simple audio equalizer, and will be referred to as the Equalizer. The Equalizer has five input signals and one output signal, summarized in Table 1.

Table 1. Equalizer interface

Name	Dir	Description
AudioIn	In	Digital audio input
BassUp	In	Increase bass button
BassDn	In	Decrease bass button
TrebleUp	In	Increase treble button
TrebleDn	In	Decrease treble button
AudioOut	Out	Digital audio output

The idea is that a digital audio input stream (AudioIn) is fed to the Equalizer from an audio source, e.g. a CD player. Four buttons are connected to the Equalizer (BassUp, BassDn, TrebleUp, TrebleDn), controlling the bass and treble level of the digital audio output stream (AudioOut). The Equalizer reacts to the button presses, not taking into account how long the button is pressed but only how many times it is pressed, and adjusts the audio accordingly. To make the Equalizer a little bit more interesting it detects when too much bass is used, and automatically prevents the user from increasing the bass while it cuts down the bass level.

This short description will be used as a very simple requirement specification for the Equalizer.

4. Modeling

Building a model of a system is essential to system design; the modeling process allows the designer to assemble components and to reason about properties and behavior of the system. Modeling in MASCOT is based on two languages, Matlab and SDL, and the model is built as a hierarchical specification. The top-level specification is written in SDL, which provides a graphical description of the system components and their interconnection. The key component in MASCOT modeling is the process. We assume that each process is specified in either Matlab or SDL. Matlab processes model dataflow behavior and SDL processes model control flow behavior. Matlab processes can communicate with other Matlab processes through

dataflow signals and with SDL processes using control signals. SDL processes can only communicate using control signals. Choosing between Matlab and SDL processes and between dataflow signals and control signals is part of the MASCOT methodology, which is described in the next section. The MASCOT model can be simulated. The SDL simulator drives the simulation and calls the Matlab engine to process dataflow signals. The execution of Matlab processes is based on a cosimulation technique described in [7]. This technique is founded on the assumption that dataflow signals can be executed piece-wise and distinguishes between two types of synchronization; head synchronization and tail synchronization. Piece-wise execution assumes that dataflow streams be split into frames, every frame corresponding to a pre-determined amount of time. Each frame is processed in a single step which is performed either when the frame starts, head synchronized, or when the frame ends, tail synchronized. The selection of synchronization policy of a Matlab process affects its ability to exchange control signals. Choosing head synchronization requires input control signals to be exchanged at the frame start, while output control signals can be exchanged during the frame duration. This implies an approximation of the time of occurrence on input control signals. Conversely, choosing tail synchronization implies an approximation of the time of occurrence on output control signals. The synchronization policy also affects how frames can be split or concatenated between Matlab processes. Output frames from a head synchronized process can be split into smaller frames without a delay penalty, but cannot be concatenated. Conversely, output frames from a tail synchronized process can be concatenated without a delay penalty, but cannot be split. More complex rules apply when head and tail synchronization is mixed, but that is beyond the scope of this text, and the reader is referred to [7].

5. Methodology

The goal of the MASCOT methodology is to write a comprehensive specification that connects the heterogeneous components of the system with each other using abstract communication structures. The specification should be intuitive to understand, easy to change, and straightforward to simulate. The specification should describe the functional behavior of the system and the specification environment should encourage the system designers to “play around” with the specification prior to implementation. Finally, the specification should govern correct implementation of the system. The methodology consists of three phases:

1. Functional Decomposition
2. Domain selection
3. Domain Interface design

Functional decomposition is a natural phase in each system design and is often carried out in a top-down fashion. The Domain Selection phase considers what language should be used to describe the subsystems. Finally, the Domain Interface Design phase integrates the SDL and Matlab simulators in three simple steps.

5.1. Functional decomposition

The first phase decomposes the system into a number of functional blocks. The decomposition can be the result from an earlier analysis activity, such as Object-Oriented Analysis. It is performed in a hierarchical manner using the block structure available in SDL, which allows for subsystem simulation at each hierarchical level. In large projects where a number of different design teams are involved, it is advantageous to let the blocks also reflect the task assignment between groups. The interconnection between blocks and hierarchies should reflect how the subsystems are connected, and what signals they will exchange. Note, that signals in a MASCOT specification can represent both discrete events (control signals) and data streams (dataflow signals).

The block interconnection interfaces will probably change as the specification evolves through design iteration loops and gradual refinement. This evolution is a natural and important process, and by employing a system-level perspective, we will prevent that interfaces are changed in one subsystem without the interconnected subsystems being notified. This is the key to specification maintenance, and will govern a correct implementation and ensure that the specification agrees with the final implementation.

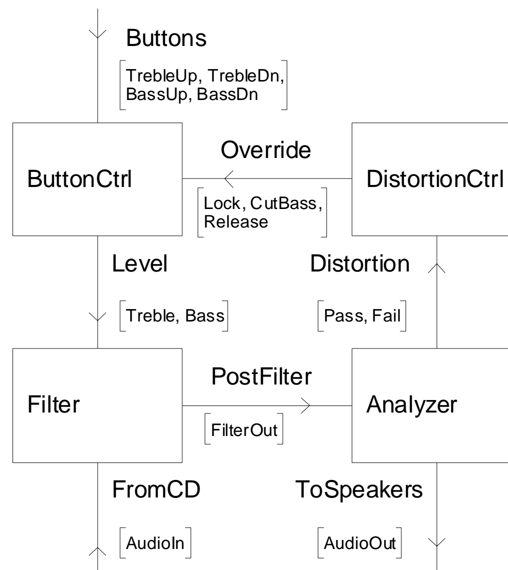

Figure 1. System Equalizer

In the Equalizer example we choose to divide the system into four blocks, shown in Figure 1. The ButtonCtrl block handles the button presses and the override signals from the DistortionCtrl block. The ButtonCtrl block also keeps track of the bass and treble levels, and signals the

Filter block each time a bass or treble level changes. The Filter block contains the audio filter, it transforms the audio input to audio output, taking into account the current bass and treble levels. The Analyzer block analyzes the audio stream after the filter, it issues pass or fail signals to the DistortionCtrl block, and outputs the audio stream. The DistortionCtrl block receives the pass and fail signals from the Analyzer, and determines if the user should be prevented from increasing the bass level, and if the bass level should be cut down.

When the block hierarchy is established, processes are identified. A process is a subsystem that operates concurrently and independently with all other processes in the system. In the Equalizer, to keep things simple, we choose to identify a single process in each block.

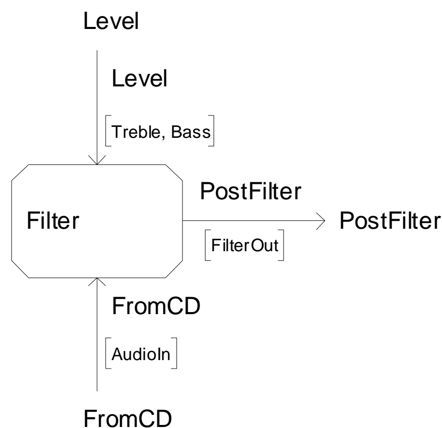

Figure 2. Filter process

The Filter block process is shown in Figure 2; the other blocks are defined in a similar manner.

This far, nothing has been decided on how the different processes will behave, this decision is left to the individual design teams. The behavior is encapsulated in the processes, which can be specified in SDL or Matlab, as we will see in the next section.

5.2. Domain selection

The domain selection decides whether the behavior is going to be described in SDL or Matlab. To this end we inspect the signals in the system and decide whether they are dataflow or control signals. A dataflow signal is a continuous signal sampled at regular intervals. A control signal is a non-periodic signal that occurs at certain time instances. All processes that are connected to one or more dataflow signal have to be specified in the Dataflow domain using Matlab, and the other processes can be specified in the Control domain using SDL.

In the Equalizer example, we decide that AudioIn, FilterOut, and AudioOut are dataflow signals. All other signals are control signals. Hence, the Filter and Analyzer processes will be specified in the Dataflow domain, and ButtonCtrl and DistortionCtrl will be specified in the Control domain.

In SDL, signals are sent on signal paths, which is either channels or signal routes. In the Equalizer we will make sure that dataflow and control signals are sent over separate signal paths. Thus, the dataflow signals are equipped with individual paths: AudioIn, FilterOut, and AudioOut are conveyed over the channels FromCD, PostFilter, and ToSpeakers respectively.

The domain is selected for processes individually or at the block level for all processes in a block.

This far, all external signals that interact with the system are connected to the environment (the frame) of the SDL system diagram. However, dataflow sources and sinks require interaction with the Matlab engine and must be declared explicitly as functional components of the system.

In the Equalizer example, the AudioIn and AudioOut signals are dataflow signals that interact with the environment. These signals require that the environment must be replaced with an explicit dataflow source and sink.

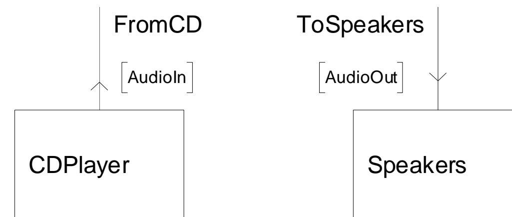

Figure 3. Equalizer source and sink

Figure 3 depicts the CDPlayer source and Speakers sink that are added to the Equalizer specification.

5.3. Domain interface design

The last step in the MASCOT methodology is to select the communication mechanism between the Dataflow and Control domains.

Communication between Control processes, such as ButtonCtrl and DistortionCtrl, only involves control signals and is carried out the normal way in SDL. Inter-domain communication between Control and Dataflow processes such as ButtonCtrl and Filter, and communication between two Dataflow processes such as Filter and Analyzer require a domain interface. The latter requires a domain interface since the simulation is driven by dataflow signals in the Control domain.

Designing a domain interface is simple and involves three steps:

1. Set dataflow processing properties. Determine sampling frequency and frame size for each dataflow signal and process and select synchronization policy for each dataflow process.
2. Set inter-domain communication properties. Decide whether inter-domain signals are notification signals or message signals and assign default values to them.

3. Construct the dataflow process wrapper. Select SDL wrapper templates and write a Matlab or Simulink execution script.

When these steps have been accomplished, a simulation can be performed, using the analysis tools in Matlab and SDL to validate the behavior.

Dataflow processing. For each dataflow signal, and for each dataflow process, the sampling frequency and frame size has to be determined. These parameters can be changed at any time, and constitute a powerful tool to tune the simulation. By increasing the frame size, the number of calls to the Matlab engine will decrease and the simulation will run faster, however, the granularity of the simulation will be coarser. The frequency and frame size together determine the duration of one frame. For signals, the duration corresponds to the amount of time that one dataflow signal represents, for processes, the duration corresponds to the time between process execution calls.

For each dataflow process, the synchronization has to be determined; a dataflow process is either head- or tail-synchronized. The process synchronization policy determines whether the process execution call will be carried out at the start, or at the end of a frame, see [7]. A number of thumb-rules are supplied to guide the designer in the choice of frame size and synchronization policy:

- Dataflow source processes should be head synchronized, using large frames, since a head synchronized frame can be split into smaller frames and/or transferred to tail synchronization without penalty. Furthermore, since sources do not have any inputs, they will not suffer from input signal approximation.
- When a dataflow process needs to accept input control signals without approximation, tail synchronization must be employed. A tail synchronized process produces output control signals with approximation, and since frame splitting after tail synchronization cannot be performed without penalty, the frame size should be small.
- Dataflow sink processes should be tail synchronized, using large frames, since tail synchronization allows frame concatenation without penalty. Furthermore, since sinks do not have any outputs, they do not suffer from output signal approximation.

Following these rules, we decide sampling frequencies, frame sizes and synchronization policies for the signals and processes in the Equalizer example. We assume that the dataflow design team has decided to use a CD source sampled at 44100Hz, and that they want to try out a 4096 sample wide FFT for the Analyzer. The sampling frequency will be equal throughout the whole system; so all dataflow signals and processes are assigned a 44100Hz sampling frequency. Starting from the dataflow source and moving to the sink, we now inspect each process and signal to assign a frame size and synchronization policy. The CDPlayer process is the source and is assigned head synchronization policy, the frame size is set to the largest possible frame size, which is the size of the whole stream used in simulation. The AudioIn signal is also assigned a frame size that equals the size of the whole stream. For the Filter process, we decide that in order to let changes in bass and treble levels affect the filter immediately, we do not want input control signal approximation; hence, we must switch to tail synchronization. Peeking forward in the data stream path we find that the Analyzer will operate on 4096 sample wide input. To match the FFT, we choose frame size 4096. Choosing larger frames would not allow the Analyzer to operate on 4096 sample frames without penalty, and choosing smaller frames would slow down simulation. For the FilterOut signal and Analyzer process, the 4096 sample frame size is thus preserved and tail synchronization is selected. Since the Analyzer process has control signal outputs, we have to accept that they will be approximated. However, in this particular example, it does not matter, since the Analyzer only produces one Pass or Fail signal for each frame that is processed. Finally, we arrive at the AudioOut signal and Speakers process, and guided by the rules, we again select the size of the whole stream as frame size, and tail synchronization as synchronization policy.

Inter-domain communication. The control signals that connect to dataflow processes are either notification signals or message signals. Notification signals do not carry a value and are used to notify a process that an event has occurred. In the Equalizer example, the Pass and Fail signals are examples of notification signals. Message signals carry a value and are used to indicate a change of some variable or parameter. In the Equalizer example, the Bass and Treble signals are examples of message signals. All control signals that are connected to dataflow processes must fall into one of the two categories. There is a third type of inter-domain communication, punch, which is a compound of the notification and message signals. Punch signals are used to extract data from a stream, and consist of a signal pair – one input notification signal and one output message signal. Punch signals will not be covered in detail here, for synchronization policy selection it is sufficient to regard them as output signals. See [7] for details.

Notification signals and message signals are interpreted in different ways when they are converted between the Control and Dataflow domains. In the Dataflow domain, control vectors are used to represent control signals. Control vectors have a sampling frequency and frame size that is inherited from the dataflow process that the control signal connects to. Control vectors also have a default value that must be chosen by the designer. Notification vectors represent notification signals and message vectors represent message signals. Conversion between vectors and signals is carried out on the border between the domains. Since the vectors have sampling frequency, the time that each sample occurs is implicitly defined. Thus, a control signal, which consists of a time-value pair, can be mapped to a specific sample in a vector and vice versa. A

notification vector contains the default value (typically NaN) in all samples that do not map to a control signal, and some other value (typically 1 or 0) in those that do. A message vector contains the default value in all samples up to the first control signal, thereafter it contains the value of the first signal up to the second signal and so on, in effect 'latching' the value into the vector. The representation of notification signals and message signals has to be known to the execution script designer when the dataflow process wrapper is designed, which is described in the next section.

Dataflow process wrapper. A process in the SDL specification that is specified in Matlab contains a wrapper that interfaces between the SDL simulator and the Matlab engine. The MASCOT environment provides a number of wrapper templates designed in SDL that are used to build an SDL wrapper for Matlab processes. The templates, summarized in Table 2, are pieces of SDL code that are assembled in the wrapper process.

Table 2. SDL Wrapper Templates

Template	Description
InitProcess	Initialize the dataflow process
InitInStream	Initialize an input stream port
InitInControl	Initialize an input control port
InitOutStream	Initialize an output stream port
InitOutControl	Initialize an output control port
InStream	Read an input stream
InMsgSignal	Read input message signal
InNotSignal	Read input notification signal
Execute	Execute the dataflow function
OutStream	Write an output stream
OutMsgSignal	Write output message signal
OutNotSignal	Write output notification signal
Punch	Punch a stream

The minimal set of templates needed in a wrapper consists of InitProcess and Execute, which initializes and executes the process. The wrapper templates use C library functions that come with MASCOT. Currently, the SDL wrapper templates are selected manually, but in the near future this will be fully automated.

When the appropriate templates have been selected, the only remaining task is to specify a Matlab execution script, which is called by the Execute wrapper template. This script has to be explicitly defined by the designer. Its purpose is to connect input and output variables through Matlab function calls, and to keep track of the internal state of the process. Not all dataflow processes have an internal state, but those that do need special attention. In the Equalizer example, the Filter process must be able to retain its internal state between function calls. Since the Matlab function filter, called from the execution script, takes a state vector as input argument and returns a state vector in its output, handling the state is just a matter of storing the state in a variable between function calls.

5.4. Simulation

When the domain interface has been designed, the simulator can be built. It will start a new Matlab engine and all Dataflow processing will be performed in the same Matlab workspace. During simulation, the designer has full access to all tools in both SDL and Matlab, and can perform advanced analysis of the system behavior.

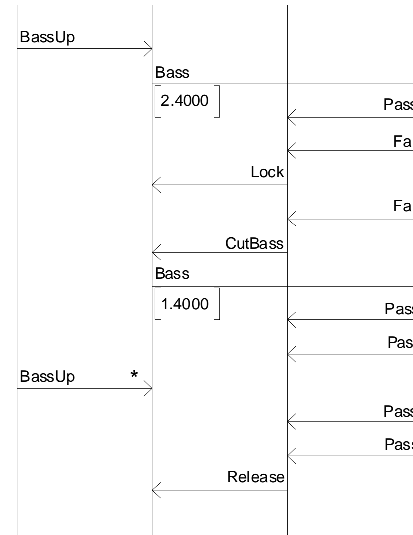

Figure 4. MSC Excerpt from simulation

Figure 4 shows an excerpt from a simulation of the Analyzer traced as an MSC (Message Sequence Chart) diagram. We clearly see how a high bass level causes the Analyzer to issue fail signals, which cut the bass value and locks the BassUp button.

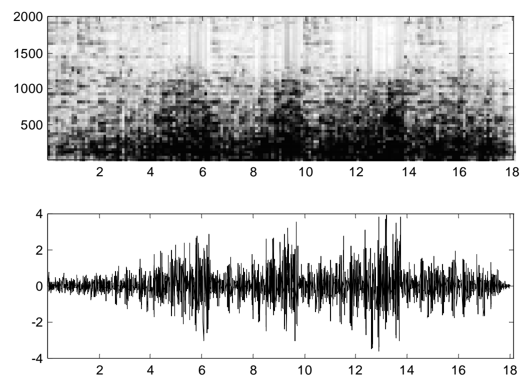

Figure 5. Simulation output

Figure 5 shows 18 seconds of the AudioOut stream visualized as a spectrum response in the 0-2000Hz range (upper diagram) and amplitude (lower diagram) using Matlab. The bass was cut three times during simulation.

6. Case study

To illustrate and test the MASCOT concept, a larger case study has successfully been carried out at CelsiusTech Electronics.

The system selected for the case study is EMSIM, an emitter simulator, which is used to simulate radar pulses that an aircraft is exposed to when flying in a radar environment. EMSIM will be implemented partly as software running on a PC and partly as an accelerator card connected to a VME bus in the PC. At startup, EMSIM reads a scenario configuration from a file and populates the environment with radar emitters. During run-time, EMSIM will either read pre-recorded flight data from a file, or receive flight data from a joystick.

The core of the EMSIM is the emitter simulation subsystem, hereafter only referred to as the Emitter, which contains the pulse generation and amplitude attenuation calculation.

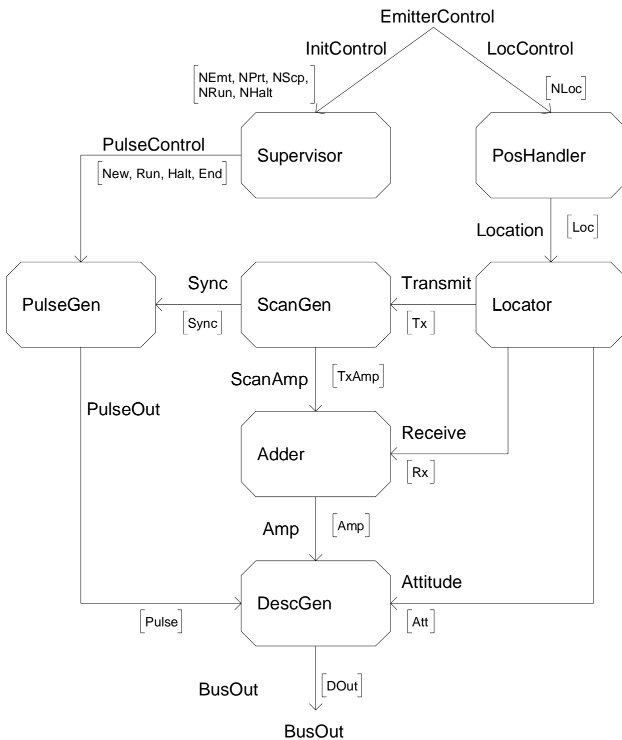

Figure 6. EMSIM emitter simulation subsystem

The Emitter, decomposed in SDL using the MASCOT methodology, is shown in Figure 6. Not shown here is the scenario reader, containing a file parser and flight data source process, and the pulse descriptor receiver. Control and Dataflow signals enter the Emitter from the EmitterControl channel, which is split into two signals routes, one for the Control signals, entering the Supervisor, and one for the Dataflow signal, entering the PosHandler. Thus, the EmitterControl channel is an example of a signal path containing both control signals and dataflow signals.

The PosHandler process receives the flight path data as a stream of position tuples sampled at 15Hz. A position tuple consists of six elements, the *x*, *y*, and *z* coordinates, and the *roll*, *pitch*, and *yaw* angles of the aircraft. PosHandler splits the data stream into frames, and passes the frames to the Locator process. The Locator performs three calculations based on the aircraft position and the position of the Emitter itself. First, the relative position of the aircraft with respect to the Emitter is calculated as elevation and azimuth angles, which are transmitted to the ScanGen process. Second, the position of the Emitter with respect to the aircraft is calculated as azimuth and elevation angles and transmitted to the DescGen process. Finally, the Locator calculates the distance between the aircraft and the Emitter, and based on the distance and relative position of the Emitter, the receiver attenuation with respect to the receiving antennas located on the aircraft is calculated and transmitted to the Adder process. The ScanGen process generates a transmitter attenuation profile, which reflects the scan pattern of the Emitter. The transmitter attenuation profile is a data stream with a sampling frequency of 1200Hz, which means that the position retrieved from the Locator is upsampled 40 times. The ScanGen process also generates a synchronization signal each time a scan pattern is completed. The synchronization signal is transmitted as a notification signal to the PulseGen process. The Adder process adds the transmitter attenuation profile from the ScanGen process with the receiver attenuation from the Locator. Since the data stream from ScanGen has a sampling frequency of 1200Hz, the Adder must upsample the receiver attenuation 40 times. The Supervisor process handles the control signals during initialization; it sets the Emitter pulse pattern, scan pattern, and position. The PulseGen process implements the pulse pattern table and emits pulses according the pulse pattern. A synchronization signal from the ScanGen process will cause PulseGen to jump to a synchronization pulse table entry. The DescGen process receives the pulses from the PulseGen process, and combines the pulse with attenuation data from the Adder and attitude data from the Locator, and generates a pulse descriptor, which is transmitted to the BusOut channel.

The PosHandler, Locator, ScanGen, Adder, and DescGen processes are modeled as Matlab processes since they connect to dataflow signals. The Supervisor and PulseGen processes are modeled as SDL processes. The only dataflow process with an internal state is ScanGen, which has to keep track of its current position in the pattern. Since none of the dataflow processes have input control signals (the Pulse and DOut signals connected to DescGen is a punch signal pair and therefore regarded as an output) we are free to use head synchronization throughout the whole dataflow path without signal approximation. Thus, we are also free to select whatever frame sizes we want, and a first choice might be a frame size of 15 for 15Hz dataflow components and 1200 for 1200Hz components. Dataflow processing will then be carried out one (simulated) second at a time.

The specification of EMSIM using MASCOT was based on an earlier design where the Dataflow part was specified as software in C. The Control part was partitioned into software and hardware, specified in C and as schematics respectively. Migrating the design to a MASCOT specification had several advantages. First, a large reduction (60%-65%) in code size between C and Matlab could be observed. Second, the specification could be simulated, which was not possible earlier. Third, the new specification was implementation independent.

6.1. Simulation speed

To demonstrate the ability to choose simulation speed, a series of simulations was carried out on the EMSIM specification, using different frame sizes. A simulation scenario with one Emitter was used, and the flight path duration was 400 seconds. The result of varying frame sizes from 1 to 40 of the 15Hz components (and from 80 to 3200 of the 1200Hz components to keep a consistent duration of processes) is shown in Figure 7 and Table 3.

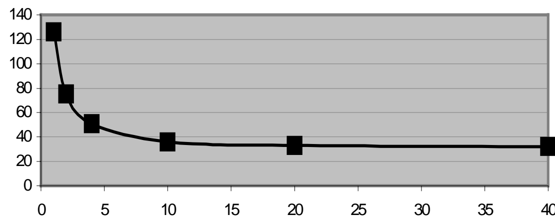

Figure 7. Simulation speed vs. frame size

A dramatic decrease in simulation time can be noted when going from 1 to 4 samples per frame. However, the larger the frame size, the larger the simulation steps, and if only the first second needs to be simulated, it is faster to use frame size 15 than 40, because we cannot interrupt the processing in the middle of a frame.

Table 3. Simulation performance

Frame Size	Simulation Time (s)	Relative Speed
1	126	100%
2	75	60%
4	51	40%
10	36	29%
20	33	26%
40	32	25%

The simulation runs four times faster when frame size is increased from 1 to 40, which gives the designer freedom to trade off simulation speed versus granularity.

7. Conclusion

We conclude that the MASCOT methodology provides the system designer with an environment for heterogeneous modeling of data and control flow dominated systems. The system specification is written in SDL and Matlab, and a minimal amount of work is needed to perform a cosimulation, thanks to the MASCOT wrapper libraries. The cosimulation relies on vector-based execution of dataflow processes, which is fast and offers an opportunity to trade off simulation speed versus granularity. A MASCOT specification is implementation independent, and the methodology presented focuses on the behavioral description of the system.

8. Acknowledgement

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

9. References

[1] S.J. Cunning, T.C. Ewing, J.T. Olson, J.W. Rozenblit, S. Schulz, “Towards an Integrated, Model-Based Codesign Environment”, *Proc. of IEEE Conference and Workshop on Engineering of Computer-Based Systems*, 1999.

[2] P. Coste, F. Hessel, P.L. Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A.A. Jerraya, “Multilanguage Design of Heterogeneous Systems”, *Proc. of the 7th International Workshop on Hardware/Software Codesign CODES’99*, 1999.

[3] P. Le Marrec, C.A. Valderrama, F. Hessel, A.A. Jerraya, M. Attia, O. Cayrol, “Hardware, software and mechanical cosimulation for automotive applications”, *Proc. International Workshop on Rapid System Prototyping*, 1998.

[4] H.J.H.N. Kenter, C. Passerone, W.J.M. Smits, Y. Watanabe, A. Sangiovanni-Vincentelli, “Designing Digital Video Systems: Modeling and Scheduling”, *Proc. of the 7th International Workshop on Hardware/Software Codesign CODES’99*, 1999.

[5] M. Romdhani, P. Chambert, A. Jeffroy, P. de Chazelles, A.A. Jerraya, “Composing ActivityCharts/StateCharts, SDL and SAO Specifications for Codesign in Avionics”, *Proc. of European Design Automation Conference*, 1995.

[6] I. Bolsens, H.J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, D. Verkest, “Hardware/software co-design of digital telecommunication systems”, *Proc. of the IEEE, vol.85, (no.3)*, p.391-418, March 1997.

[7] P. Bjuréus, A. Jantsch, “Heterogeneous System-Level Co-simulation with SDL and Matlab”, *Proc. of the Forum on Design Languages, FDL’99*, 1999.

[8] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, Y. Xiong, “Overview of the Ptolemy Project”, *ERL Technical Report UCB/ERL No. M99/37, University of California, Berkeley, USA*, July 1999.

[9] A. Olsen, O. Færgemand, *Systems Engineering Using SDL-92*, North-Holland 1997.

[10] J. Ellsberger, D. Hogrefe, A. Sarma, *SDL Formal Object-oriented Language for Communicating Systems*, Prentice-Hall 1997.

[11] D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall 1994.