ICS

TECHNICAL REPORT

Co-design of Emulators for Power electric Processes Using SpecC Methodology

Slim Ben Saoud, Daniel D. Gajski

Technical Report ICS-01-46
July 25, 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

Slim Ben Saoud
Fulbright Visitor @ CECS
INSAT-Tunis-TUNISIA
sbensaou@ics.uci.edu
http://www.cecs.uci.edu/~sbensaou

Daniel D. Gajski
CECS
UCI-California-USA
gajski@ics.uci.edu
http://www.cecs.uci.edu/~gajski

Information and Computer Science
University of California, Irvine

Co-design of Emulators for Power electric Processes Using SpecC Methodology

Slim Ben Saoud, Daniel D. Gajski

Technical Report ICS-01-46
July 25, 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

Slim Ben Saoud
Fulbright Visitor @ CECS
INSAT-Tunis-TUNISIA
sbensaou@ics.uci.edu
http://www.cecs.uci.edu/~sbensaou

Daniel D. Gajski
CECS
UCI-California-USA
gajski@ics.uci.edu
http://www.cecs.uci.edu/~gajski

Abstract

Emulation of CMS systems is an interesting approach to complete the validation of new digital control unit and to perform the diagnosis tasks. However to be efficient, the emulator have to run in real time in order to reproduce exactly the physical process functioning.

Today, realization of this emulator is not possible using standard electronic components. Therefore, we oriented our work to the development of new embedded systems specific to these applications of emulation.

This report describes the design of this emulator employing the system-level design methodology developed at CECS-UC Irvine (SpecC methodology). Starting from the abstract executable specification written in SpecC language, different design alternatives concerning the system architecture (components and communications) are explored and the emulator is gradually refined and mapped to a final communication model. This model can then be used with backend tools for implementation and manufacturing. For illustration of this approach, we discuss at the end of this report the case of a DC system emulator and we describe in details the different stages undergone.

Contents

List of Figures

Co-design of Emulators for Power electric Processes Using SpecC Methodology

Slim Ben Saoud, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

ABSTRACT

Emulation of CMS[1] systems is an interesting approach to complete the validation of new digital control unit and to perform the diagnosis tasks. However to be efficient, the emulator have to run in real time in order to reproduce exactly the physical process functioning.
Today, realization of this emulator is not possible using standard electronic components. Therefore, we oriented our work to the development of new embedded systems specific to these applications of emulation.
This report describes the design of this emulator employing the system-level design methodology developed at CECS-UC Irvine (SpecC methodology). Starting from the abstract executable specification written in SpecC language, different design alternatives concerning the system architecture (components and communications) are explored and the emulator is gradually refined and mapped to a final communication model. This model can then be used with backend tools for implementation and manufacturing.
For illustration of this approach, we discuss at the end of this report the case of a DC system emulator and we describe in details the different stages undergone.

1 Introduction

In this project we propose to design a real time emulator for electrical system using SpecC methodology [1,2]. This emulator will be used with the control device either for complete validation of this one at the development and validation stage or for diagnosis at normal functioning stage. In both cases, the emulator should behave like the physical system in real time.

Realization of this emulator is essentially faced to the execution time constraints. Indeed the emulator has to replace high dynamic systems in real time. So we distinguish three approaches in which real time emulator can be implemented: Digital, Analog and Hybrid. These approaches are discussed in previous publications [3,4,5].

The main difficulty in this realization is to satisfy both specifications of real time functioning and of flexibility. Association of these characteristics imposed uses of digital approaches, which guaranty user-friendliness.
On the other side, improvements in VLSI technology have to led the wide spread use of specific processor, which may also be used to realize complete system.

According to these considerations, our work is oriented to the design of embedded systems specific to the emulation application. Therefore, we study the synthesis process of the emulation systems directly from their specification. This approach requires the ability to synthesize specified functions into software or hardware to meet the given constraints. It has the most flexibility since software, architecture and each component are custom made. However, it requires a well-defined methodology with clear steps, easy transformations and efficient tools to help the designer in the synthesis process.

In this project, we apply the SpecC methodology for the design of this real-time emulator.

In this report, we describe this new approach and we present the case of a DC system as an example. We present at the beginning an introduction to the emulation principles. From this description and according to the SpecC methodology, we deduce the specification model of the emulator. Then, we describe the different steps and transformations used to convert this model to a communication model according to the SpecC methodology. This model can be then transformed to an implementation model ready for manufacturing.

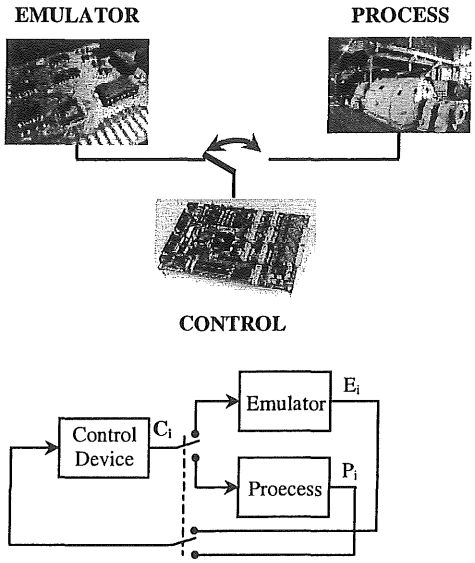
2 Emulator Principles

The objective of the emulation approach in the electric drive applications is to design an electronic system, which can reproduce the physical system functioning in real time and with high precision. This system, called emulator, will be used for both of the new control device validation and of diagnosis:

- Validation of control device: before using the control device on the physical system and in order to avoid any design surprise (usually destructive and expensive), the control device is validated on the emulation step where it is connected to the emulator. This emulator should behave exactly like the physical

[1] Static Converters / electric Motors / Sensors

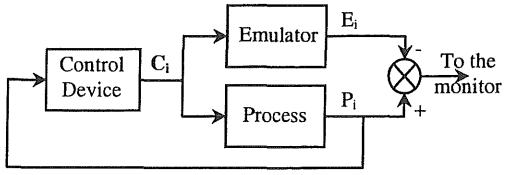
system, in sense to make the control device believe that it is connected to the real CMS[2] system. Thereby it has to generate and receive information similarly to the physical process: it receives control signals and generates information about the system state in forms identical to those obtained by sensors. After this step, the control device is completely validated and can be switched for use with the physical process as shown on figure 1.



C_i Control signals / E_i Emulator outputs / P_i Process outputs

Figure 1: Structure of the emulation application

- Diagnosis: The emulator is used in parallel with the process and must receive the same control signals as this process. Outputs of the emulator will be continuously compared in real time and in order to detect any dysfunction of this process. Stored results of the emulator will be used to analyze problems and to avoid or detect their origins.



C_i Control signals / E_i Emulator outputs / P_i Process outputs

Figure 2: Structure of the diagnosis application

In both cases, the emulator has to reproduce accurately the process functioning. Therefore, it has to compute the System State and to convert the obtained results in forms identical to those obtained by sensors.

[2] Converter/Motor/Sensors

For this and in order to obtain precise results, we should use precise models of different components used on this process and a high performance computing system with the lowest computing step as possible (1µs or less).

On the other hand, this emulator must be flexible, easily configurable by the user according to her/his application, and it must allow the storage of results and if necessary monitoring.

According to these previous considerations, the emulator structure will be composed of three main modules:

- Computing module: it computes, according to the digitized models, the system state variables. This computing is performed in a loop manner with a period *hor* (computing step) equal to 1µs or less for high precision applications. This module receives digital control signals and generates corresponding numeric results representing states of different components on the process (converter, motor, load and sensors).
- Emulation sensors module: it converts numeric results on different other forms identical to those obtained by the used sensors. Different types of conversion will be done like numeric/numeric (solver, ...), numeric/digital (encoder, ...) , numeric/analog (LEM, ...). The execution of each conversion will be done in a periodic manner with a period that depends on the temporal characteristics of the sensor and the captured magnitude. However, these periods constraints are usually less severe than the computing module one.
- Monitoring module: it performs the acquisition of new parameters and the storage of results. It can also perform some monitoring tasks. For most applications, the initialization task is performed only at the beginning of the emulator functioning. However, for high performances applications, it can be executed several times within the emulation running in order to introduce variation of system parameters due, for example to the variation of the environment like the temperature,... On the other hand, the storage operation will be done in a periodic manner with a period, Ts = x * *hor*, where x is an integer configurable by the user.

3 SpecC Methodology [1,2]

With the ever increasing complexity and time-to-market pressures in the design of systems-on-chip (SOCs) or embedded systems in general, both industry and EDA vendors are trying to move the design to higher levels of abstraction, in order to increase productivity. At higher levels, there is no difference between hardware and software. An SOC is the combination of hardware and

software, and at the system-level the disciplines merge. Great productivity gains can be achieved by starting design from an executable system specification instead of an RTL description as the golden reference model, throwing away all system models developed earlier in the process. However, we are still just at the beginning of understanding the design process at the system level. No tools and no well-defined design flows are available from industry or EDA vendors.

Managing the complexity at higher levels of abstraction is not possible without having a very well defined system-level design flow. A well-defined design methodology is the basis for all, synthesis, verification, design automation, and so on. Only then can we find or create a language that actually fits the desired flow, and not vice versa.

SpecC System-level design methodology and SpecC language are the result of decades of research done in the area of SOC design at the Center for Embedded Computer Systems (CECS) at the University of Irvine California (UCI).

SpecC language was developed exactly for the purpose of supporting a system-level design flow, and it therefore satisfies all the requirements of synthesizability, verifiability, and so on. SpecC is a superset of C and adds a minimal, orthogonal set of concepts needed for system design. It is currently in the process of being standardized.

The SpecC methodology is a set of models and transformations on the models (Figure 3). The models written in programming language (SpecC language) are executables descriptions of the same system at different levels of abstraction in the design process. The transformations are a series of well-defined steps through which the initial specification is gradually mapped onto a detailed implementation description ready for manufacturing.

The SpecC design methodology is based on 4 well-defined models, namely a specification model, an architecture model, a communication model, and finally, an implementation model. In the following section, we will give a brief description of each model and of the refinement tasks leading from a functional specification model all the way to a cycle-accurate implementation model in SpecC.

Specification model: The SpecC system-level design methodology starts with the capture of the intended functionality in the form of an executable specification as shown in figure 3. This initial specification model describes the functionality as well as the performance, power, cost and other constraints of the intended design. It does not make any premature allusions to implementation details.
During specification capture the designer may reuse existing code segments, functions or procedures by instantiating them out of an algorithm library.
Specification model is a purely functional model that abstracts the system functionality. It is the starting point of system design process and the input to architecture exploration task.

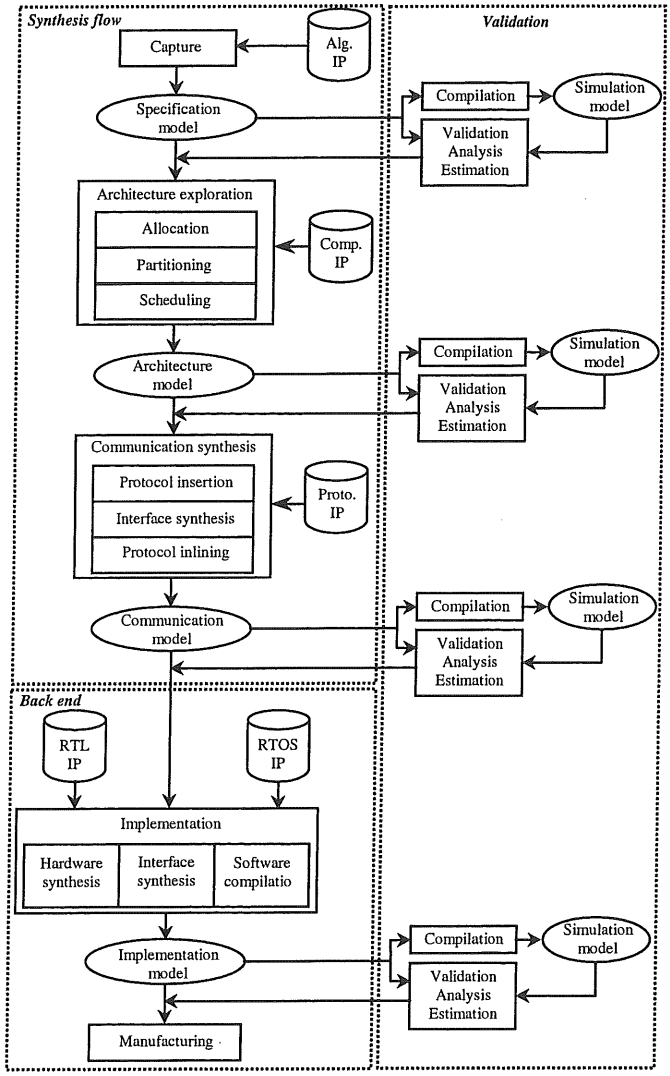


Figure 3: SpecC methodology

Architecture exploration: It refines the specification into an architecture model. It includes the design steps of allocation, partitioning of behaviors, channels, and variables, and scheduling.
Allocation determines the number and types of the system components, such as general-purpose or custom processors, memories, and busses, which will be used to implement the system behavior. Allocation includes the

reuse of intellectual property (IP), when IP components are selected from the component library.
Behavior partitioning distributes the behaviors (or processes) that comprise the system functionality amongst the allocated processing elements. *Variable partitioning* assigns variables to memories, and *channel partitioning* assigns communication channels to busses.
Scheduling determines the order of execution of the behaviors assigned to either the standard or custom processors after partitioning. In other words, scheduling is used for software and hardware components.

Architecture exploration is an iterative process culminating with an architecture model that represents a refinement of the specification model. Estimators evaluate each architecture candidate's satisfaction of the design constraints; until all constraints are satisfied, component and connectivity reallocation is performed and a new architecture with different components, connectivity, partitions, schedules or protocols is generated and evaluated.

Architecture model: It describes the system functionality as well as the overall structure of the final implementation for the design. The communication in the architecture model is through the abstract global channels.

Communication Synthesis: It refines the abstract communication between behaviors in the architecture model into an implementation. The task of communication synthesis includes the insertion of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. In the resulting communication model, communication is described in terms of actual wires and timing relationships are described by bus protocols.

Communication model: It is the final output of the system-level design process which describes the system structure as a set of components connected through the wires of the set of buses.

Backend: The result of the synthesis flow is handed off to the backend tools, as shown in the lower part of figure 3. The software part of the hand-off model consists of C code for compilation and the hardware part consists of behavioral C (VHDL) code for high-level synthesis. The backend tools include compilers and high-level synthesis tool. The compilers are used to compile the software C code for the chosen processor. The high-level synthesis tool synthesizes the functionality assigned to custom hardware and the functionality of transducers which are necessary for connecting different processors, memories, and IPs.
After software compilation and hardware synthesis, the final implementation model is generated.

Implementation model: It represents a clock-cycle accurate description of the whole system. This description, in turn, then serves as the basis for manufacturing of the system.

In each of the tasks the designer can make design decisions manually by using an interactive graphical user interface, for example, while transformations from one model into another can be accomplished automatically by following the refinement rules or model guidelines. After each refinement step in the synthesis flow, a corresponding SpecC model of the system is generated, which means that design decisions made in each design task are reflected in the generated models. Thus, in the validation flow that is orthogonal to the synthesis flow in the SpecC methodology, one can perform simulation, analysis and estimation of the SpecC models generated after each task.
After each design step, the design model is statically analyzed to estimate certain quality metrics such as performance, cost, and power consumption. Analysis and estimation results are reported to the user and back-annotated into the model for simulation and further synthesis.
The design can be statically analyzed or simulated after each step for validation of design correctness in terms of functionality, performance, and other constraints. A simulation model is compiled after each step which can be run on the host computer to validate correctness for simulation.
At any stage of the refinement process, a standard software debugger can be used to locate and fix the errors if verification fails. Such debuggers enable one to set break points anywhere in the source code and to perform detailed state inspection at any time.

4 Specification Model

According to the previous description, the emulator can be described by a main behavior (*Emul*) including different sub-behaviors: one for the computing module, one for sensors and one for monitoring (Figure 4).

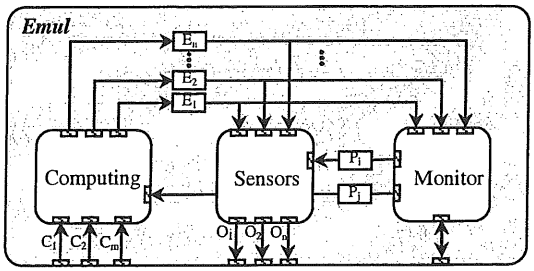


Figure 4: Specification model of the emulator

These behaviors are usually composed of other sub-behavior according to the modular structure of the

physical system. So, we use a sub-behavior for each of the converter, motor/load and sensor component and we add separate sub-behaviors for initialization process and for the monitoring module (including storage). These behaviors can also be decomposed of child-behaviors as we usually do with the motor/load behavior. Indeed, this behavior is usually decomposed on mechanic-child-behavior and electric-child-behavior (Figure 5).

Computing steps used with different modules are not the same since they don't have the same temporal characteristics. So we propose, for more flexibility, to add to each of the obtained child-behaviors a clock-generator behavior that controls its execution. The occurrence of the clock event is defined according to the module temporal characteristics.

However, we usually use the same computing step for the CM system (1μs) and different computing steps for each different type of sensor...

The obtained specification model is shown on figure 5. It was validated for the case of DC and AC systems.

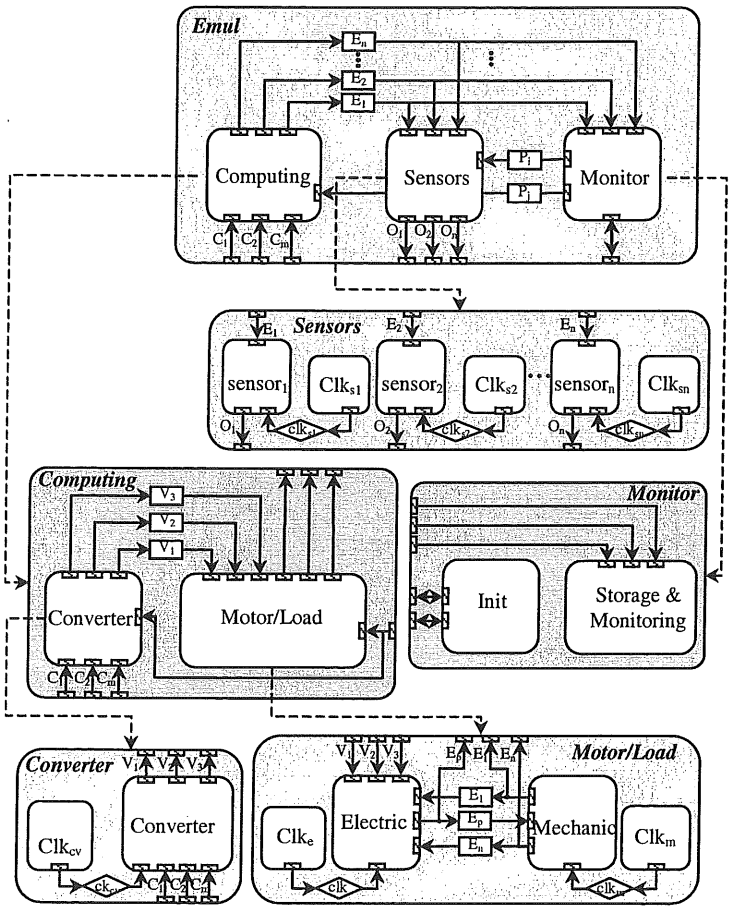


Figure 5: Detailed Specification model of the emulator

Constraints: One of the most important constraints with this application is the computing time of the CM System State. Indeed, this computing must be done, according to the given models, in a periodic manner with a computing step of 1μs or less. While this seems to be realizable using High performance standard processors for the case of simple system using simple models, it is not possible for high performance application that needs sophisticated models and smaller computing step. Temporal problems of this computing, implemented into a processor, are related to the computing load and to the use of interruption for the periodic functioning management.

Other constraints such as the resolution of the used DAC devices and the resolution and precision of the used sensors (C.I.O, ...) will certainly influence the design procedure.

5 Architecture Exploration

At the architecture allocation phase, different possibilities can be used according to the application specifications (flexibility, time, cost, precision,...). However, since we need to store emulator results for monitoring and study, we distinguish two main structures according to the emulator integration on the physical system:

- Monitored emulator: with this structure, a master processor (PC or the micro-controller implemented on the control device) monitors the emulator. This master will initiate the emulator functioning by sending new parameters and will be used in parallel with the emulator to store results and treatment of these results for monitoring. Transmission of results is done in a periodic manner Ts.
  In this case, the emulator will interrupt the processor functioning at each new storage period in order to begin a transmission task. This solution imposed the use of interruption, which can disturb the running task performed by the master (case of the microcontroller).
- Autonomous emulator: the emulator is associated to an external memory on which will be stored parameters and emulation results. The emulator functioning is then independent from other computing systems. At the starting step, it reads new parameters from the external memory block and then, at each new storage period, it will write new results into successive registers of this memory. Another processor used for initialization and monitoring will manage this memory when the emulation is off-line in order to initialize the process parameters and to restore emulation results. This solution is very useful when used with the micro-controller of the control device since it will not disturb the control tasks.

However it introduces some delay in the monitoring operation.

For each of these structures, different allocations can be studied. These allocations represent different possibilities of implementation of the emulation core *EmulCore* (computing and sensors emulation modules) between full-software and full-hardware solutions.
In our project, temporal constraints of our system are very severe, especially in the case of complex models. So, we chose to study the full-hardware solution.
In this case, we usually use one ASIC for computing of CM system states and a custom hardware for each (or some) sensors. This introduces communications between these different hardware blocks and the ASIC component. In order to simplify this presentation, we consider that all modules are included in the same HW component ***EmulCore***.

In the following sections, we present the study of this solution when applied to the two described emulation structures (Monitored and Autonomous). Therefore, we will present for each of these structures the different modifications applied to the specification model in order to obtain the architecture model.

5.1 Monitored Emulator

In this structure, the emulator is supervised by a software application, which initiates the emulator functioning and receives results from it. This solution is useful for monitoring and the software can be implemented on the control device microcontroller. However, in this case, cautions should be taken about the management of communication without disturbing the control tasks (since these transfers will be done using interruption of the microcontroller...).

5.1.1 Allocation and Behaviors Partitioning

The first step in architecture exploration is to allocate a set of processing elements (PEs) and to map the behaviors of specification onto the allocated PEs. In this structure and according to the previous considerations, the emulation system will be composed of two PEs: the hardware component *EmulCore* that performs emulation tasks and the software component used for initialization and results storage.

The obtained refined model after behavior partitioning is represented by figure 6. In this model, two behaviors are added to *EmulCore-PE1* (*init* & *storage* behaviors) in order to synchronize and establish communication with the software component. This communication becomes system-global and it is moved to the top-level connecting the PE behaviors.

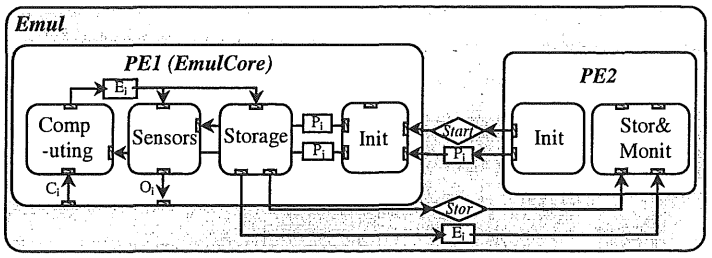


Figure 6: Architecture model after allocation and behavior partitioning (monitored emulator)

5.1.2 Variable Partitioning and Scheduling

The number of exchanged variables between HW and SW components is very limited (usually less than 10), so we chose to use local copies for these variables in each PEs. Therefore in the refined model obtained after variable partitioning, global variables for results and parameters are replaced with their respective abstract channels C_{Pi} (for parameter Pi) and C_{Ei} (for result magnitude Ei). Code is inserted into behaviors to communicate variable values over these channels. Note that data are exchanged in a vector type: one for parameters and one for results.

Scheduling The next step in the architecture exploration process is to schedule behavior executions on the inherently sequential processing elements. Processing elements have a single thread of control only. Therefore, behaviors mapped to the same PE can only execute sequentially and have to be serialized.

In this application, at the starting point, the emulator is waiting for a start order from the master. Then, it receives new parameters of the system sent by the master and begins its functioning tasks. At each x computing steps (storage period), the emulator will send results (im, Wm) to the master. Two functioning mode can be used with the master:

- It can be used only for monitoring of the emulation: initiating and receiving results. In this case after sending parameters, the master will wait for the results. In this solution, implementation of communication will be easy, however the master can't do any thing else at the same time.
  In this case, behaviors are executed in a fixed and predetermined order. The static scheduling approach will be applied easily by converting all concurrent statements into sequential compositions.

- The master is also used for other tasks such as control or treatment of these results for monitoring and diagnosis. Therefore, a dynamic scheduling approach is required.

In this case synchronization for information exchanges will be done using interruption. The emulator will send an interruption signal to the master each storage period in order to begin transfer. This solution is more adequate for embedded systems since it can be integrated easily in the control device, allowing then both application of validation and diagnosis.

In order to preserve the shared semantics of the variables and to keep the local copies inside PEs in synchronization, updated data values are exchanged between the two components at the existing synchronization points. Therefore, these updated data values are communicated over the existing message-passing channels together with synchronization of the behaviors execution among the PEs.

According to these purposes, the intermediate architecture model obtained after variable partitioning and scheduling is represented by figure 7.

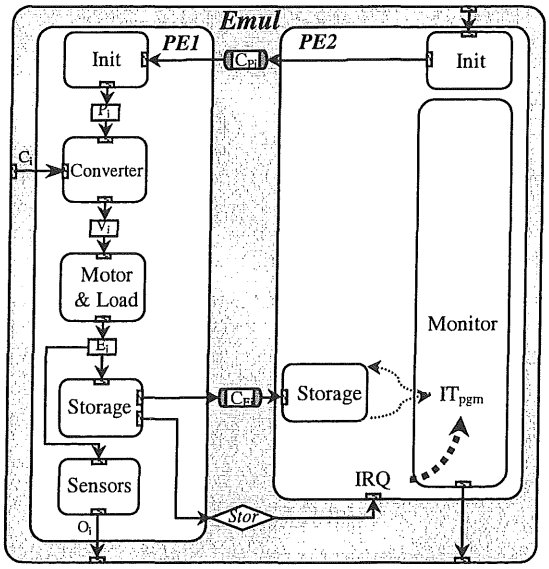


Figure 7: Architecture model after variable partitioning and scheduling (monitored emulator)

5.1.3 Channel Partitioning

The goal of the channel partitioning is to group and encapsulate all the channels existing between the communicating blocks into one bus. The bus is also a type of channels in SpecC and it implies that the future implementation would be wired busses.

In our application, we have only two components communicating with each other. Therefore, only one system bus is allocated connecting PE1 and PE2, and all communication channels are mapped onto that bus.
The obtained refined model is represented by figure 8.

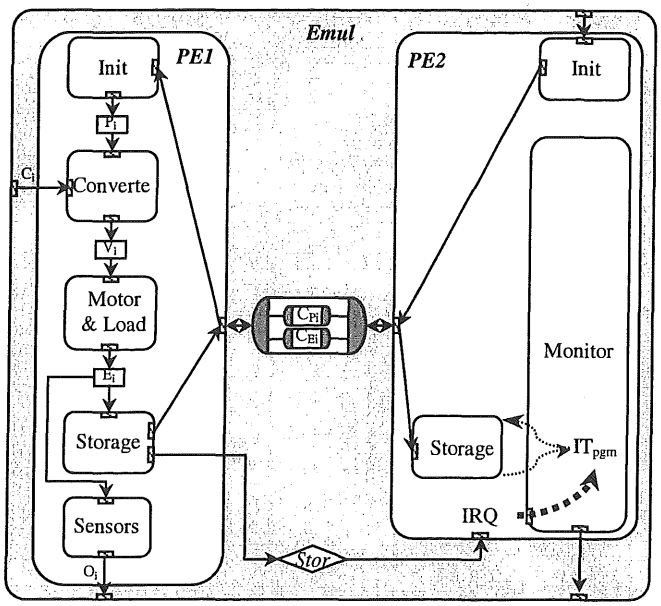


Figure 8: Architecture refined model after channel partitioning (monitored emulator)

5.2 Autonomous Emulator

A memory is added to the emulator system in order to obtain an independent emulation system. In this memory will be stored both the system parameters and the emulation results. At the beginning, the emulator starts by reading of parameters at corresponding addresses on the memory, and then it will store, at each storage period, the obtained results. Both the emulator and the monitor will manage this memory.
This solution can be used either alone or with a software component (as on the control device). The main advantage is the portability and the autonomy of this emulator since it will not interrupt the software component. This last can at each time ask memory for information. The start/stop of the emulator can be done by the SW component.

To simplify the application, we assumed that the monitor manages this memory only when the emulator is stopped. So, there's no management of transfer conflicts to perform. However, synchronization between Hardware and Software components must be done at the beginning and at the end of the emulation procedure.

5.2.1 Allocation and Behaviors Partitioning

As specified before, the architecture target will be composed of three components: the EmulCore hardware

component, the software component (processor) and the memory block. This memory, shared by the two active components, is used for storage of parameters and emulation results.
However, local copies are added, especially, to the emulator component in order to perform its computing as fast as possible without asking the memory at each computing step for data. These local copies correspond to the used parameters and to the results obtained in the last computing step (or in the n last computing steps according to the used method for digitizing).

The obtained refined model after behavior partitioning is represented by figure 9. In this model, two behaviors are added to EmulCore (init & storage behaviors) in order to synchronize and establish communication with the software and memory components.
This communication becomes system-global and it is moved to the top-level connecting the PE behaviors. Synchronization is done at the beginning and the end of the emulation procedure by using two events *Start* and *End*. The processor generates the *Start* event after computing the new system parameters according to the user configuration. It signals to the init behavior in the EmulCore component that parameters are ready. Then, the emulator starts its computing and storage procedures. At the end of its functioning, it sends the *End* event to the software component signaling that emulation is stopped and that data are ready in the memory block.

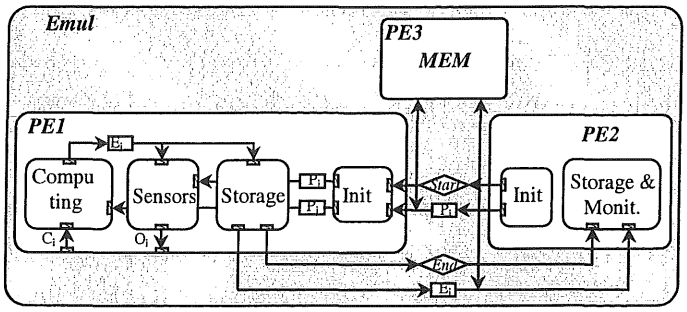


Figure 9: architecture model after allocation and behavior partitioning (autonomous emulator)

5.2.2 Variable Partitioning and Scheduling

In this case and according to the previous specifications, synchronization between HW and SW components are performed by using of two events *Start* and *End*.
The *Start* event can be associated to the parameters variables Pi and encapsulated into a Message-Passing channel that models the abstract communication semantics of blocking, unbuffered message-passing between any two client-behaviors.
On the other hand, the *End* event will be connected to a processor interrupt input. The interruption program is used to set a flag *F_end* when this interruption is activated. When the software program needs the emulation results for monitoring, it tests this flag. If it is set, it performs the data read sequences from memory, otherwise it waits for the interruption activation.

The obtained refined model after variable partitioning and scheduling is shown on figure 10.

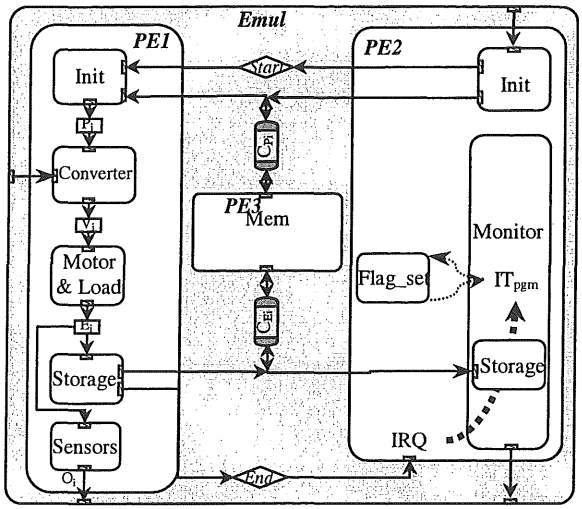


Figure 10: architecture model after variable partitioning and scheduling (autonomous emulator)

Note that, the emulation can generate the digital signal *F_end* instead of the event *End*. And then, no interruption is required for the synchronization.

5.2.3 Channel Partitioning

The obtained architecture target is composed of two active PEs (PE1 and PE2) that share a memory block (PE3). Therefore, only one bus is used into which all the global channels and their implementation are encapsulated (figure 11).

5.3 Remarks

These structures can be decomposed into others components according to the complexity of the studied system. This bursting will usually concern the distribution of sensors on others hardware components as shown on figure 12.

We can also distribute a complex child behavior like the *Motor/Load* behavior on several components. However this will introduce communication protocols and then cautions must be taken in order to obtain the best compromise "performance/cost". These distributions have to be studied in more details with estimation tools according to the defined application.

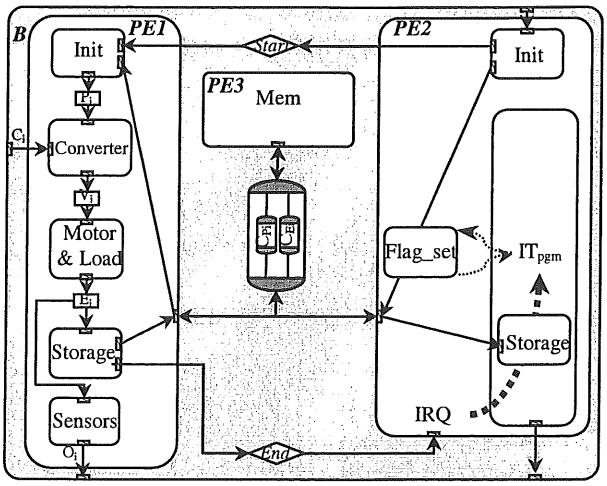


Figure 11: Architecture refined model after channel partitioning (autonomous emulator)

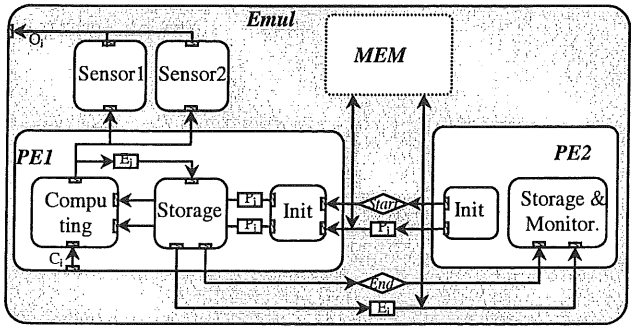


Figure 12: Decomposed architecture of the emulator

6 Communication Synthesis

In the SpecC architecture model, obtained from architecture exploration, the communication between components is still modeled on a high level through abstract channels. The channels were obtained by simple encapsulation of global variables and their corresponding synchronization. Although the channels represent the grouping according to the mapping onto busses, they do not yet contain any information about the actual implementation of the communication primitive's semantics (*send(), receive(),...*). The communication synthesis, therefore, is to gradually refine the channels in the system model down to an actual implementation with data transfers over wires. This comprises the steps of protocol insertion, transducer synthesis and protocol inlining [6].

In our application, the software component is usually chosen from standard processors (IP components). Therefore, PE2 will be replaced at the communication synthesis by this predesigned processor chosen from the component library. This processor has predefined functionality and fixed external interfaces and communication protocols.

In the process of protocol insertion, the SW component will then be replaced with a model of the IP component that includes the IP component behavior and the protocol wrapper (Figure 13). The wrapper provides the abstract canonical interface for communication with the environment.

Note that in case of protocol incompatibilities, transducers components will have to be inserted between the IP component and the bus to translate between the protocols.

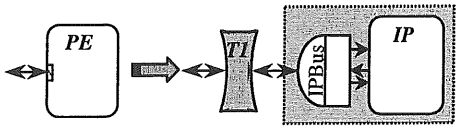


Figure 13: IP component insertion into the architecture model

For the illustration of our application we will use the DSP56600 (Motorola) as the chosen IP component for implementing the software behaviors [7].

6.1 Monitored Emulator

6.1.1 Protocol Insertion

Figure 14 shows the emulator model after insertion of a bus protocol for the system bus, and after the processor behavior has been replaced with a model of the real processor with a wrapper.

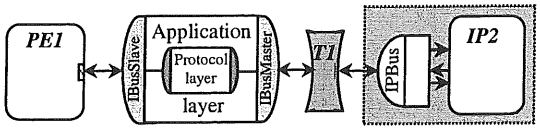


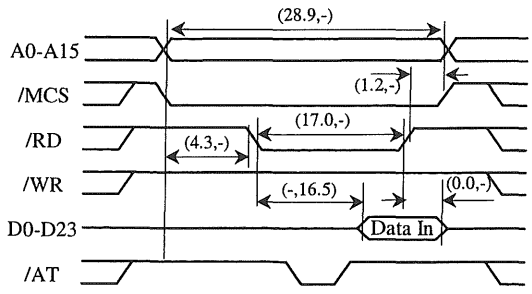
Figure 14: Intermediate communication model after protocol insertion (monitored emulator)

In this case, we propose to use the processor protocol as the system protocol. Then the transducer will be removed during inlining, since processor and bus protocol are identical. However, transducer will be added, if needed, between hardware protocol and the system bus protocol. This transducer will be later synthesized with the hardware component.

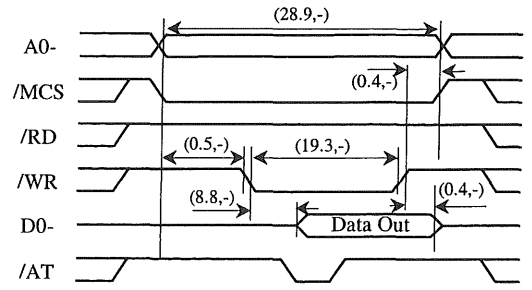
Usually, no protocol is defined for the hardware part, so we need to analyze the HW datapath to generate the I/O protocol and then insert the transducer component.

Note that in order to simplify this illustration, we suppose that the hardware component has the same protocol as the system bus (DSP56600 protocol). Then, the transducer is not required.

The protocol channel in the system bus and the wrapped processor model describe and implement the processor bus protocol according to its timing diagram, shown in figure 15. The protocol layer provides primitives for performing read/write transfers and for raising interrupts over the processor bus.
On top of the protocol layer, the application layer created during protocol insertion implements the semantics of the abstract communication of the bus channel, using the primitives provided by the encapsulated protocol channel.
On the software side, the communication primitives of the application layer are customized I/O routines inside the processor that become part of the generated RTOS. The I/O routines together with customized interrupt handlers perform the necessary handshaking and data conversions to implement the semantics of the communication primitives using the processor's I/O instructions.
On the hardware side, the communication primitives are part of the system bus application layer. They will later be inlined into hardware and realized as additional FSMDs that implement the low-level bus protocol and the high level handshaking and data conversions.



DSP56600 - SRAM Read



DSP56600 - SRAM Write

Figure 15: Protocols of the DSP56600 external bus

In the monitored emulator of the emulator example and after protocol insertion, the DSP is the central component and the master of the system bus. The software on the DSP initiates all data transfers on the system bus between software and hardware components. However, this software is associated to a DSP interrupt, which is triggered by the hardware by using asynchronous events.

The DSP is the communication master, the required blocking, synchronous message passing semantics are realized as follows: the ASIC signals its ready state by raising an interrupt whenever it reaches a communication point (at the beginning of each new Ts period). The software on the DSP performing the task of control or of monitoring will be interrupted in order to execute the transfer program. So, it begins by transferring the data one word at a time by repeatedly executing instructions that initiate read or write cycles on the external bus.

The hardware has a dedicated address that corresponds to a location in the DSP memory space. Once software and hardware at both ends are synchronized, the DSP as the master on the bus initiates and controls data transfers by reading from or writing to the memory location with the address of the custom hardware. The ASIC, on the other hand, detects its own address and answers DSP requests by supplying or storing the requested data from and to their local registers or memories.

On top of the actual low-level transfers, synchronization and handshaking between hardware and software is handled by sending and receiving events on both sides. On the processor side, the operating system sends events to the custom hardware by writing to the external bus address assigned to it. Than, signaling an event is implemented by initiating a data transfer over the processor bus and to the hardware component and therefore both can be combined. The hardware component listening on the bus receives the transfer request event and supplies or stores the data. On the other hand, the Custom hardware PEs sends event to the software by interrupting the processor. Indeed, the hardware signals the availability of new computation results to the processor by raising an interrupt line.
Note that the interrupt assigned to the hardware component should be with the highest priority in order to perform emulation in real time.

6.1.2 Protocol Inlining

Finally, inlining of the wrapper functionality into hardware components is performed (Figure 16): the resulting SFSMDs for interrupt generation, address decoding and bus protocol handling functionality are combined with the SFSMD executing the behavior originally assigned to the custom hardware coprocessor. Both parts will then be synthesized together to generate the final custom hardware. After inlining, the actual ports and connections are exposed and visible, resulting in the final system model as actually seen after implementation.

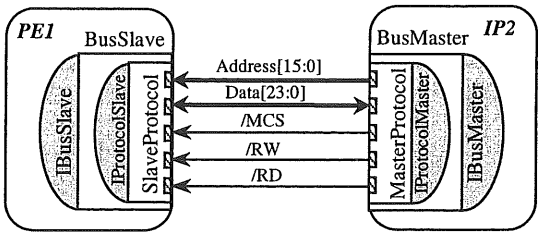


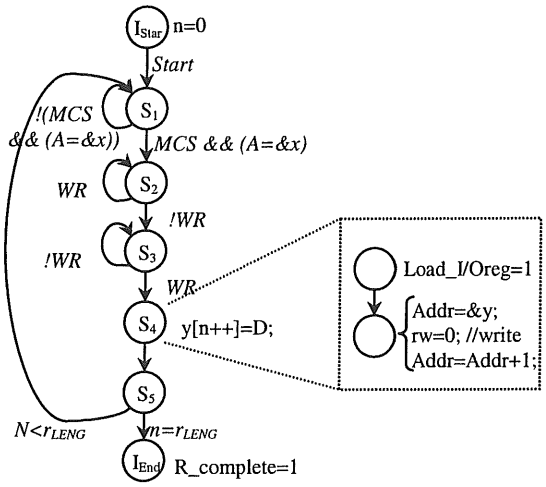
Figure 16: Communication model after protocol inlining (monitored emulator)

In the case of the ASIC emulator system, communication primitives are inlined into the *init* and *storage* behaviors that have been created during partitioning for synchronization and communication between the ASIC and the processor. Both the applications and protocol layers of the communications primitives that had been created during protocol insertion are inlined into the custom hardware behaviors.
The application layer performs synchronization with the DSP and converts the complex data structures into bus transfers. The protocol layer performs the actual bus transfers according to the protocol shown in figure 15.

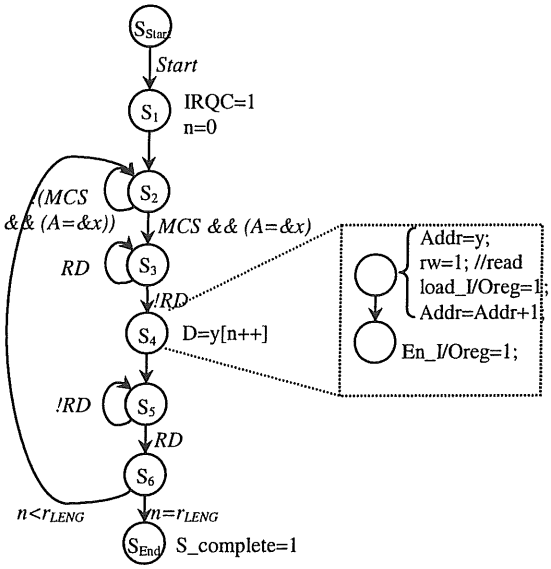
The *Init* SFSMD synchronizes with the software on the DSP, receives the process parameters over the processor bus, and starts the emulator computing. At each new storage period (Ts=Xs cycles), the storage SFSMD synchronizes with the DSP (by generating an interruption) in order to transfer the emulation results back to the processor. For their bus transfers, the SFSMDs (figure 17) implement the bus protocol according to the timing diagram shown in figure 15.

The Init SFSMD waits for a falling edge of the chip select signal MCS and samples the bus address in state S1 until a transfer with the address of the ASIC is recognized. In state S2, the WR control signal is sampled until a falling edge has been detected that signals the beginning of a bus write cycle. In state 3, the Init SFSMD waits for the rising edge of signal WR before latching the data bus contents into I/O register, writing the data from the I/O reg into local memory Mem, and incrementing the address register Addr in superstate S4. Finally, it increments and checks the loop counter in state S5, and branches back to wait for the start of the next word transfer until all data items have been received. Then it will activate the emulator computing.

The storage SFSMD synchronizes with the DSP by raising the processor's interrupt line *IRQC* in its first state *S1*. Then it waits for a falling edge of the chip select signal MCS and samples the bus address in state S2 until a transfer with the address of the ASIC is recognized. In state S3, the RD control signal is sampled until a falling edge has been detected that signals the beginning of a bus read cycle. In superstate S4, the storage SFSMD reads the data word from the custom hardware memory Mem into the I/O reg, increments the local address register Addr, and enables I/O reg on the data bus. It then waits for the rising edge of RD in state S5. Finally, it increments and checks the loop counter in state S6, and branches back to wait for the start of the next word transfer until all data items have been transmitted.



(a) *Init* SFSMD



(b) *Storage* SFSMD

Figure 17: HW communication SFSMDs

In case of this application, we assume that the synthesized hardware for the Init and Storage SFSMDs will be fast enough to handle bursts of successive transfers initiated by the master processor at the maximal processor bus speed (2 processor cycles per bus transfer). Therefore, the delay of the loops in the SFSMDs has to be less than 2 processor cycles. Otherwise, wait states have to be added to the bursts of bus transfers on the processor side, or

more elaborate handshaking scheme (e.g. DMA or interrupt-based acknowledgment of single transfers) would become necessary.

Figure 18 shows the implementation of the interface between hardware and software after final inlining.

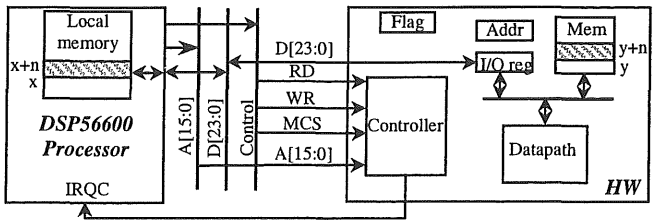


Figure 18: HW/SW interfacing model (monitored emulator)

The sequence for transferring data and control from the DSP to the ASIC is:

1- Under the assumption that the coprocessor is ready whenever the software on the DSP wants to initiate the emulator parameters, the DSP starts writing these parameters onto the DSP bus. The processor successively writes the block of data for the hardware onto the bus one word at a time by initiating a sequence of bus write cycles with the address assigned to the coprocessor.
2- The Init SFSMD of the HW component is triggered by an address match, receives the data word over the bus, latches it in its I/O reg and finally stores the latched data in the local memory of the ASIC as described previously.
3- When the last data has been received, control in the ASIC model is transferred from the Init SFSMD to the SFSMD corresponding to the emulator-computing behavior.
4- The emulator computing SFSMD reads the process parameters from the local memory and starts the emulator computing process.

The transfer of emulator results to the DSP is handled in a similar manner:

1- The emulator computing SFSMD writes the obtained results back into the memory. At the beginning of each Ts period, control is transferred to the Storage SFSMD in the hardware component.
2- The Storage SFSMD interrupts the processor and then waits for the start of the bus data transfer.
3- The interrupt program on the DSP starts the communication software. The processor reads the data from the hardware component I/O reg over the bus one word at a time by initiating a sequence of successive bus read cycles. The data is read one word at a time using the address assigned to the I/O reg in the ASIC.
4- The storage SFSMD in the ASIC decodes the bus address, reads the results from the local memory, and puts the requested data on the bus.
5- After each read, the ASIC loads the I/O reg with the next data word until the complete complex data is sent to the DSP.

Note that for each message, the processor transfers the data items sequentially over the bus one word at a time in a predefined, fixed order. Hence, at each step in the sequence of data transfers it is implicitly defined which word of which data item is currently being transferred. The custom hardware and the DSP keep track of the sequence of data transfers, and according to their internal sides determine how to process the transferred item.
The interrupt priority assigned to the ASIC must be with high priority in order to not interrupt the transfer between the emulator and the DSP. This allows a real time functioning of the emulation system.

6.2 Autonomous Emulator

6.2.1 Protocol Insertion

Figure 19 shows the emulator model after the insertion of the DSP bus protocol as the system bus protocol, and after the processor behavior has been replaced with a model of the real processor with a wrapper. Transducer will be added, if needed, between hardware protocol and the system bus protocol. On the other hand, memory must be able to respond to the read and write requests from DSP and ASIC. This again requires design of a bus interface for memory to respond to bus read/write requests. The behaviors and complexity of such interfaces depends on the time-constrained behavior of these components at their ports. So, we must discuss the timing behavior of these components and then analyze these interfaces [8].

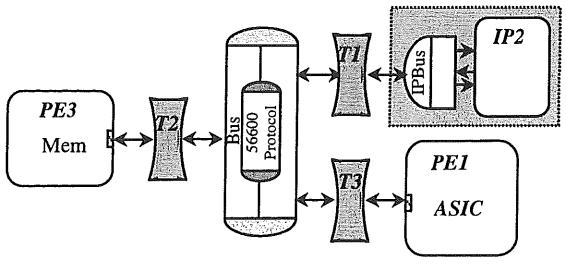
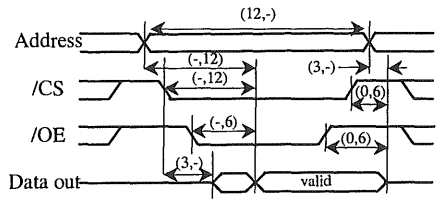


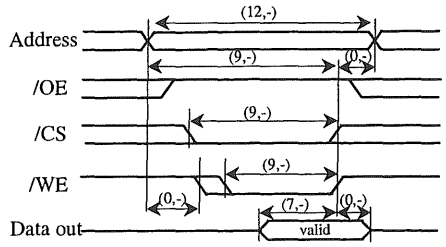
Figure 19: Communication model after protocol insertion (autonomous emulator)

Note that in order to simplify this illustration, we assume that the hardware component has the same protocol as the system bus (DSP56600 protocol).
On the other hand, we experimented with Samsung memory KM68257C [9], which is a CMOS static RAM

and has 8 common input and output lines. The different pins and the memory specification for read and write cycles are shown in figure 20.



KM68257C - Read



KM68257C - Write

Figure 20: Memory bus protocol (KM68257C)

According to these specifications, this memory is fast enough and will be used directly without any additional interface according to the schema of figure 21. Therefore, no interfaces are required with this architecture model.

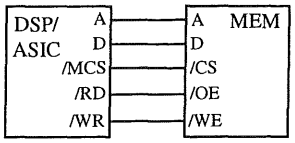


Figure 21: Interfacing with memory block

Parameters and results are stored in the memory at precise corresponding locations with precise addresses that correspond to free locations in the DSP and ASIC memory spaces.

According to this structure architecture model, we distinguish two masters sharing the same bus and the same memory component:

- DSP: it writes at the beginning parameters values to their corresponding locations in the memory block, and reads results tables;
- ASIC: it reads at the beginning parameters from their corresponding locations, and writes results at each execution of the storage behavior (Ts period).

So, cautions must be taken in order to avoid communication conflicts. Two solutions can be considered:

- Without management of the bus: The DSP initiates the emulator functioning after initialization of the memory with new parameters. Indeed, it sends to the emulator the start event to start the emulator functioning. At the reception of this signal, the emulator, begins by reading parameters from the corresponding memory locations and then starts the computing process. According to the previous specifications, the emulator writes results to the memory at each Ts period. These results are written in a table. Along this time, the DSP can perform others tasks; however it is not allowed to do transfer on its external bus while the emulator is running. When the emulation process is finished, the hardware informs the DSP by sending an external event that can be used with interruption that sets a corresponding flag or by setting a signal connected to a processor digital input. So the DSP, when it is ready for results acquisition will test this flag, and then perform the acquisition of the result table for analysis. The two main advantages of this solution are that the required management of the bus is very simple and the monitoring processor performs other tasks while running emulator. However, the DSP has some functioning restrictions (forbidden access to the external bus during emulation) and cannot read results before the end of the emulator computing.

- With management of the bus: The beginning of operation is done like the previous solution: the DSP initialize the memory then initiate the emulation functioning. Then, both the DSP and the ASIC can access to the memory for read/write. Specific primitives in the application layer will manage conflicts and priority is given to the ASIC. The main advantage of this solution is to allow the access of the DSP to the memory to change parameters or read results in order to perform the monitoring operations. However, this solution requires a sophisticated protocol to manage conflicts. This solution can be used for high performance application indeed it can introduce, for example, in real time parameters variations due to the temperature variation and to asynchronous events and it can perform monitoring in real time since it has not to wait the end of the emulation and the DSP can at each time read results from the memory.

As described in previous sections, the application layer, used in the components (ASIC and DSP), wraps around the protocol layer and implements the abstract communication semantics required by the behaviors in the application over the primitives supported by the protocol. In this structure of the emulator system, this layer has to perform tasks like synchronization, arbitration, addressing

of data on the bus, slicing of abstract data types into bus words.

In this project, we implement the case of the first solution (without management of the bus), which represents a simple bus management protocol.
In this case the required synchronous passing semantics are realized as follows: The DSP determines new parameters and it stores them in their locations inside the memory block. Then, it signals by an asynchronous event to the emulator that data are ready and it continue execution of other tasks without using its external bus. The emulator waiting for the start signal from the DSP, receives this event and then execute the init behavior before beginning emulator computing. At each Ts period, the emulator increments the memory address and store new results in a table. At the end of its computing, the ASIC signals by an asynchronous event to the DSP that results are ready in the memory. This event will interrupt the DSP program and set a flag. The DSP, tests this flag when results are required in order to perform new acquisition.

6.2.2 Protocol Inlining

The communication model obtained after inlinig is represented by figure 22. As detailed in the previous section, the applications and protocol layers are inlined into the *init* and *storage* behaviors that have been created during partitioning for synchronization and communication between the ASIC and the processor. The application layer performs synchronization with the DSP and converts the complex data structures into bus transfers. The protocol layer performs the actual bus transfers according to the protocol shown in figure 15.

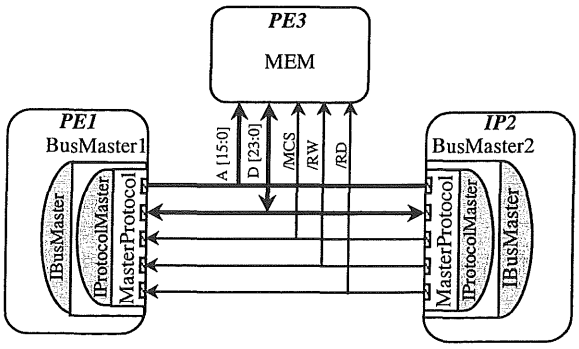
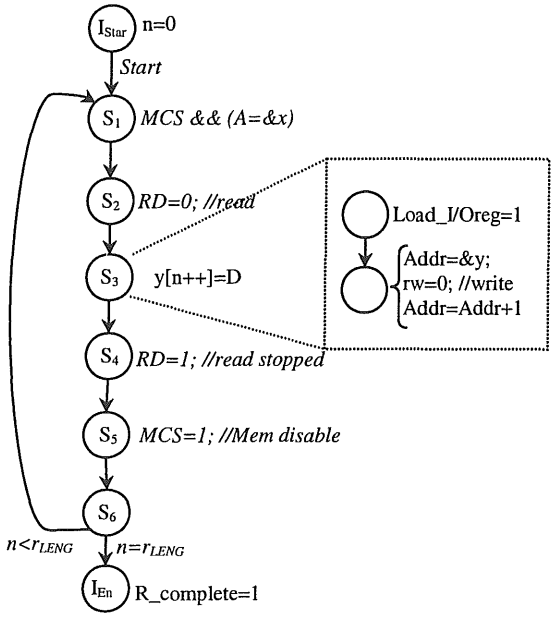
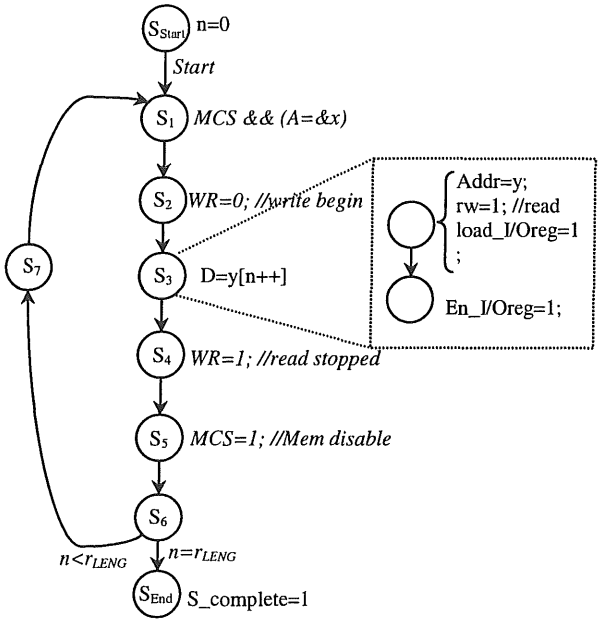


Figure 22: Communication model after protocol inlinig (autonomous emulator)

The *Init* SFSMD synchronizes with the software on the DSP, it receives the start signal from it. Then, it reads the process parameters over the processor bus from the memory, and starts the emulator computing. At each new storage period ($Ts=x_s$ hor), the storage SFSMD writes emulation results into the memory table. When finished, the ASIC send an event to the DSP signaling that data are ready in the memory and that bus is "free". For the bus transfers, the SFSMDs implement the bus protocol according to the timing diagram shown in figure 23.



(a) *Init* SFSMD



(b) *Storage* SFSMD

Figure 23: HW communication SFSMDs (Autonomous emulator)

The *Init* SFSMD waits for a falling edge of a signal *start* generated by the DSP. Then it begins the parameters read from the memory: it begin by enabling the address

register on the address bus and falling the signal MCS for the memory selection (S1). Then, it falls the signal RD in state S2 for signaling the beginning of a bus read cycle. In the superstate S3, it latches the data bus contents into the I/O reg and writes the data from the I/O reg into local memory Mem and increments the address reg Addr. In state S4, it ends the read cycle by raising the RD and the MCS signals (S5). Finally, it increments and checks the loop counter in state S6, and branches back to start the next word read cycle until all data items have been received. Then it will activate the emulator computing. In this transfer, parameters have same address in both the memory block and ASIC local memory.

At each new storage period, the storage SFSMD performs the writing process into the memory of the emulator computing results. It begins by enabling the address reg on the address bus and falling the signal MCS for the memory selection (S1). Then, it falls the signal WR in state S2 for signaling the beginning of a bus write cycle. In the superstate S3, the storage SFSMD reads the data word from the custom hardware memory Mem into the I/O reg, increments the local address register Addr, and enables I/O reg on the data bus. In state S4, it ends the write cycle by raising the WR signal and then it disables the memory selection by raising the MCS signal (S5). In state S6, it increments and checks the loop counter in state S4, and branches to state S7 until all data items have been received. In state S7, it increments the address corresponding to the next register to write in the external memory block. At the end of this transfer sequences, the Storage SFSMD gives control to the emulator computing SFSMD in order to continue the emulation process.

Note that in the SFSMDs description in figure 23 we omitted the temporal synchronization representation in order to simplify the schema. This synchronization is done according to the protocol diagram of figure 15.

Figure 24 shows the implementation of the interconnection between the three used components after final inlining.

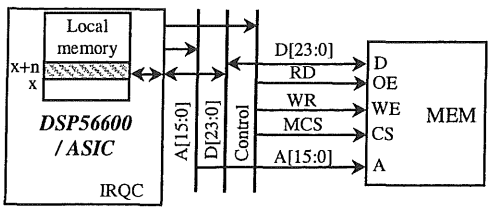


Figure 24: DSP/Memory interfacing model

The sequence for transferring data and control between the DSP and the memory is:

1- When the software on the DSP wants to initiate the emulator parameters in the memory, the DSP starts writing these parameters onto the DSP bus. The processor successively writes the block of data onto the bus one word at a time by initiating a sequence of bus write cycles with the corresponding addresses assigned to the parameters registers in the memory block.
2- Then, it sends an event to the ASIC signaling that new parameters are already stored in the memory and begins another task without using the external bus.
3- Once the software on the DSP is ready to receive the results, it performs an operating system call that checks the interrupt flag. If the flag is set, the RTOS call returns immediately. Otherwise, the RTOS waits until the interrupt has been received before returning to the caller. In both cases, the RTOS resets the flag before returning.
4- The DSP reads the data from the memory block over the bus one word at a time by initiating a sequence of successive bus read cycles. The data is read one word at a time using the address assigned to the corresponding register in the memory component.

The sequence for transferring data and control between the ASIC and the memory is:

1- The emulator computing SFSMD waits for the starting signal generates by the DSP then control is given to the Init SFSMD.
2- The Init SFSMD performs read of parameters from the external memory then it gives control to the emulation computing behavior.
3- At each Ts period, the Storage SFSMD is activated in order to perform writing of new results.
4- At the end of the emulation procedure, the emulator sends an event to the DSP signaling that results are ready in the memory block.

Note that for each message, the processor transfers the data items sequentially over the bus one word at a time in a predefined, fixed order. Hence, at each step in the sequence of data transfers it is implicitly defined which word of which data item is currently being transferred. The custom hardware and the DSP keep track of the sequence of data transfers, and according to their internal sides determine how to process the transferred item.

6.2.3 Results

A final simulation of these communication models including interrupt handling, external data transfers, and so on, was done. As described before, the communication model is a bus functional model. The behaviors in the

component are simulated on a functional level in native C language annotated with estimated delays. On the other hand, communication between components is modeled accurately over actual wires of the processor bus.

In the backend, these models will be synthesized into a structural view of all components in the system architecture. The functionality of each component will be implemented on top of the component's RTL or instruction-set micro-architecture. In the process, timing will be refined down to the level of individual clock cycles based on each component's clock period.

7 Example of a DC System

To validate this new approach of emulator design, based on SpecC methodology, we describe in this section the case of a DC system emulator. This system is composed of a DC motor fed by a four-quadrant chopper and associated to a Hall sensor for the current capture and to an Optical Incremental Encoder for the speed capture.

For the computing of this system state, we use simple models for the chopper and the electrical motor. These models are digitized using Runge-Kutta 2 and obtained equations are as follows:

- ***Motor/load***

$$\begin{cases} i_m(k+1) = \alpha.i_m(k) + \beta.\Omega_m(k) + \gamma.V_h(k) \\ \Omega_m(k+1) = \lambda.i_m(k) + \mu.\Omega_m(k) + \nu.sign(\Omega_m(k)) \end{cases} \quad (1)$$

α, β, γ, λ, μ and ν are obtained from system parameters and computing step *hor*.

- ***Chopper***

$$V_h = (C_0 - C_1).V_e \quad (2)$$

V_e is the input voltage, V_h the output voltage and C_0, C_1 control signals.

The Hall sensor output is a voltage magnitude that represents the current value. This output voltage depends on the Hall sensor characteristics and the current value. In our case this dependency is represented by a simple model as described by equation1: Vout=im*5/15 volts. A more sophisticated model (including influence of noises, temperature, wear,...) can be specified in this behavior in order to reproduce more precisely the Hall sensor output.
The optical incremental encoder generates two quadrature square wave signals (S0 and S1) with the same frequency (proportional to the motor frequency) as represented by the figure 25.

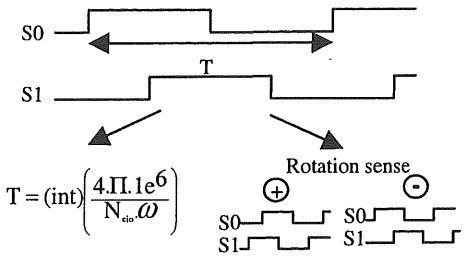


Figure 25: EIO sensor specifications

In our project we studied two solutions according to the used data format. The first one used Floating-point format (floating emulator) and the second one used integer format (integer emulator).

For the integer emulator, models equations used for computing are converted into integer form. This translation is made according to two main considerations in order to have the best computing precision:

- We take the same precision level of the system parameters when coding them in integer form ($1e^{-6}$);
- We used Δi_m and Δw_m instead of I_m and Ω_m, which represent variations of i_m and w_m where:

$$\begin{cases} i_m = \frac{1}{1024} I_m \\ \omega_m = \frac{1}{1024} \Omega_m \end{cases}$$

This allows obtaining a good precision of computing.
According to these considerations, we obtain following equations:

$$\begin{cases} 128.\Delta i_m(k+1) = \frac{a.i_m(k)}{1024} + \frac{b.\frac{\omega_m(k)}{8}}{1024} + g_v(C_0(k) - C_1(k)) \\ \Delta W_m(k+1) = l.i_m(k) + m.\omega_m(k) + n.signe(\omega_m(k)) \end{cases} \quad 3)$$

where:

$$\begin{cases} a = 128.1024.(\alpha - 1) \\ b = 1024.1024.\beta \\ g_v = 1024.128.\gamma.V_e \end{cases}$$

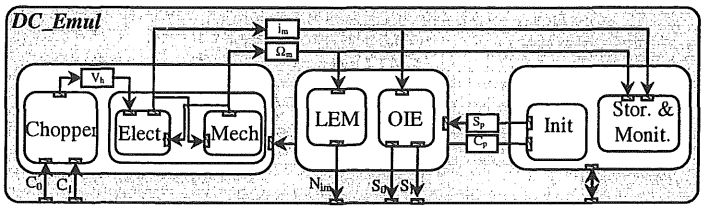
and:

$$\begin{cases} l = 1024.1024.\lambda \\ m = 1024.1024.(\mu - 1) \\ n = 1024.1024.\nu \\ \Delta W_m(k+1) = 1024.1024.\Delta\omega_m(k+1) \end{cases}$$

a, *b*, g_v, *l*, *m* and *n* will be the new parameters of this system.
The main advantage of the integer form is to simplify the computing unit.

7.1 Specification Model

According to the previous description, the specification model of the emulator can be represented as shown on figure 26.



S_p Sensors parameters / C_p Computing parameters

Figure 26: Emulator specification (case of a DC system)

The execution of these different behaviors is well scheduled by using different clock behaviors. Note that for simplification reason, clock generator behaviors are not represented in this figure (Figure 26).

The *init* behavior is executed only one time at the beginning of this model execution. It performs the computing of different parameters and the initialization of intermediate variables. Parameters are obtained, as specified in previous equations, according to the system characteristics, to the digitizing method and to the retained computing step.

All the other behaviors are executed in a periodic manner with different periods. These periods are specified in the clock generator behaviors by using the instruction **waitfor**(*Time_delay*). When the *Time_delay* is reached, the clock behavior generates an event that will activate corresponding computing behavior.

Note that at this stage, there is not any notion of time. Therefore, values (*Time_delay*) used with these clock behaviors serve only to the scheduling of behaviors execution. Execution procedure is supposed to be done with 0 time_delay.

These behaviors execution is scheduled as following:

- The storage behavior is used, in our example, only for the capture of emulator results (no monitoring task is employed). Therefore it is executed at each $T_s=X_s.hor$ period ($X_s=10$).
- The computing core of the system state (chopper & motor/load) is executed at each $Tc=hor$ period. It performs the computing of the needed System State magnitudes, which are in our case i_m *(current)* and Ω_m *(speed)*.
- The LEM behavior is used to emulate the LEM running, which generates a voltage corresponding to the current magnitude. This behavior will be associated with a DAC component. So, it generates the corresponding integer N_{im} according to the obtained current value im and to the resolution of the used DAC. According to the high dynamic of the electrical mode, the execution period of this behavior must be fast enough to cover the current variation with high precision. However, it is limited by the temporal characteristics of the DAC (time needed for conversion). In our case we use a delay equal to 5 ($X_i=5$).
- The OIE behavior is used to generate two digital signals identical to those obtained by the OIE sensor and according to the speed value Ω_m. Since, the variation of the speed magnitude is very slow, we usually use an important delay time for the execution of this module. In our case we used $x_\Omega=1000$.

The obtained specification model is validated by simulation. Figure 27 represents the obtained results for the current magnitude im.

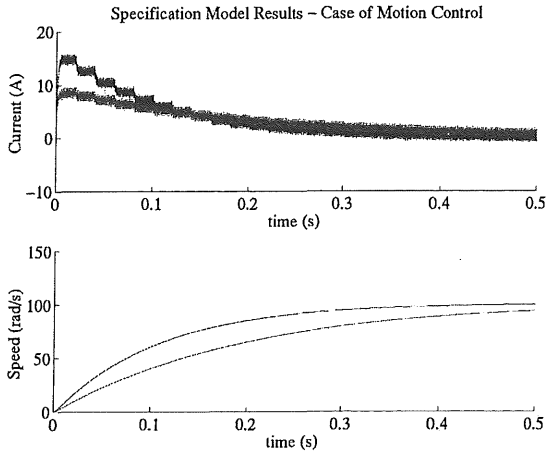


Figure 27: Specification model results

Note: The specification model represent different modules in parallel like the physical system. However, the execution is done in a sequential manner !!!

7.2 Architecture Model

The Architecture exploration has been done following the described steps and transformations of section 5. Therefore, two refined architecture models are retained and validated for each of the floating emulator and the integer emulator.

7.3 Communication Model

The two studied structures, described in section 6, are applied to the case of the DC system. So for each of the floating emulator and integer emulator, we designed a communication model for the monitored structure and another for the autonomous structure.

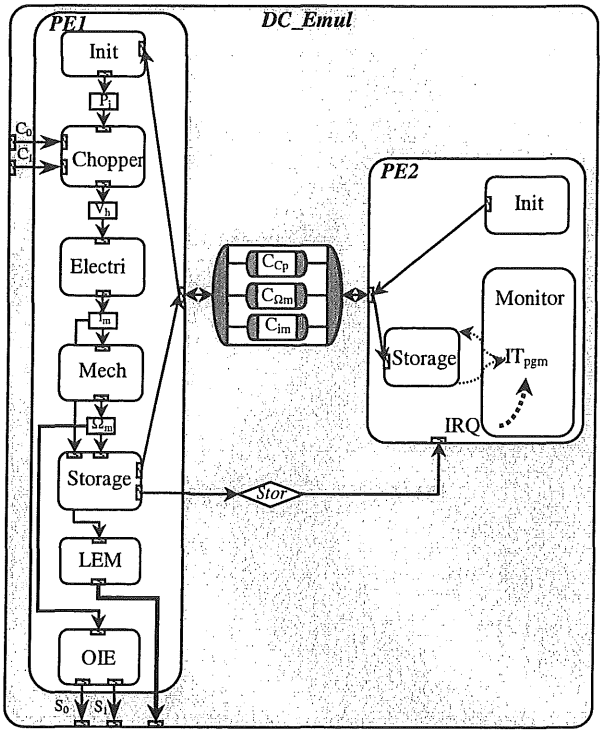


Figure 28: Architecture refined model of the monitored emulator (case of DC system)

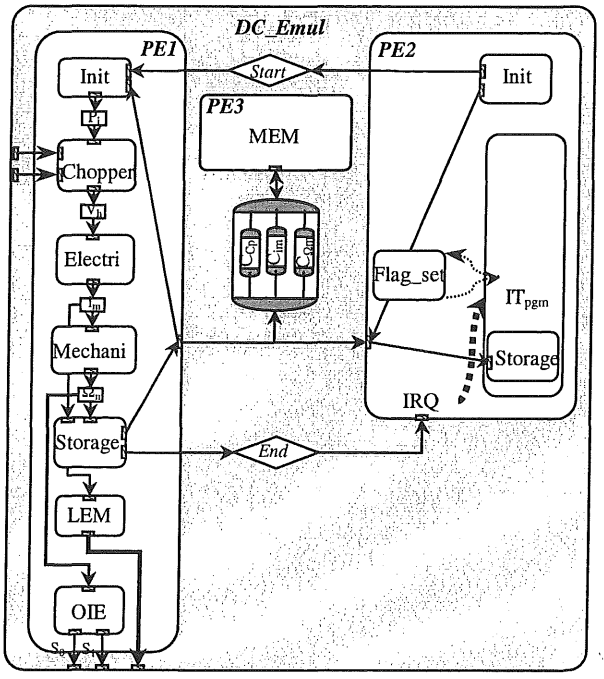


Figure 29: Architecture model of the autonomous emulator (case of DC system)

For this design, we used the DSP56600 as a software component and for the autonomous structure we add a block of High Speed Static RAM (32Kx8 bit) type KM68257C (Samsung). The communication exploration is done according the transformations described in section 6.

In this example, we used the DSP external bus protocol as the system bus protocol and we suppose that the hardware and software protocols are compatible. The memory, as specified in section 6, is fast enough and it doesn't require any other interface. Therefore, we didn't include transducers in the communication model (figure 30).

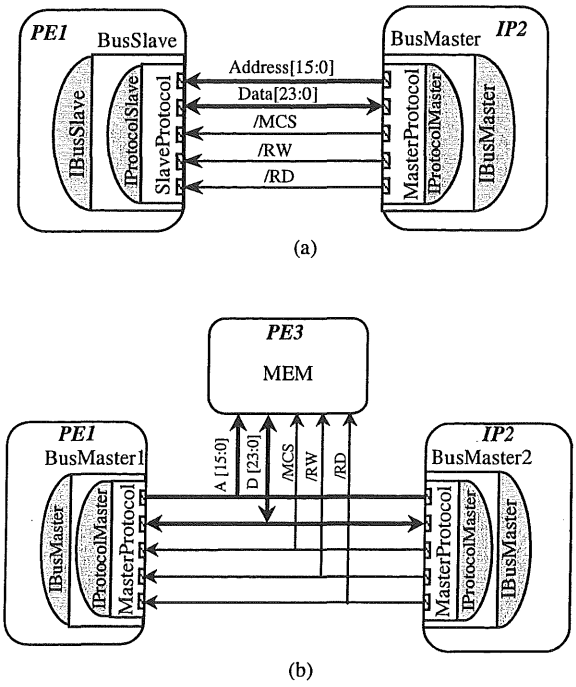


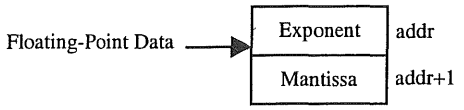
Figure 30: Communication refined model (a-monitored emulator, b- autonomous emulator)

However, actually we should determine the HW SFSMD in order to study accurately the compatibility with the system bus protocol. Otherwise, transducer must be included and synthesized.

Communication and synchronization are specified as mentioned in section 6. the Data transfer is performed in a vector type by the application layer. This layer performs several tasks like synchronization, addressing of data on the bus, slicing of abstract data types into bus words... Especially, in our example and in the case of the floating emulator, this application layer performs the conversion of floating data to words in order to transfer them from a component to another over the system bus. This conversion is done according to the DSP56600 C compiler specifications and to the floating format implemented on the hardware. In this example we

suppose that the hardware use the same format as described on the DSP56600 C compiler.

In DSP56600, the C data float and double are both implemented as single precision in a format different to the IEEE STD 754-1985 standard format for binary floating-point arithmetic. Figure 31 represents some specifications of this format. For more details, refer to Motorola technical books.



Characteristic	*DSP56KCC Format*	*IEEE Format*
Mantissa precision	23 bits	24 bits
Hidden leading One	No	Yes
Mantissa Format	24 bit Two's Complement Fraction	23 bit Unsigned Magnitude Fraction
Exponent Width	16 bits (14 bits used)	8 bits / 11bits
Format Width	48 bits	32 bits / 64 bits

Figure 31: Specifications of the C compiler floating-point format (DSP56600)

The obtained communication models including interrupt handling are validated by simulation. Therefore they are ready for synthesis in the backend process. A long this process, these models will be transformed into cycle-accurate implementation models according to the following tasks:

- HW component will be synthesized into a netlist of register-transfer level (RTL) components;
- SW will be converted into a C program, compiled into the processor's instruction set and possibly linked against an RTOS;
- The application and protocol layer functionality will be synthesized into a cycle-accurate implementation of the bus protocols on each component. This requires generation of bus interface FSMDs on the hardware side and generation of assembly code for the bus drivers on the software side.

The implementation model will be synthesized in the future.

8 Conclusion

In this report, we present a new design approach of power electronic and electric drive emulators, based on the SpecC methodology. According to this methodology and during the synthesis process, three models are constructed: the specification model, the architecture model and the communication model. For each of them we discussed the mechanism of refinement and we proceeded to a validation by simulation. The correct output demonstrates the correctness of our models.

In this approach, we developed two structures of the emulator (*monitored* and *autonomous*) that differ by their functioning constraints. The *monitored* structure represents an emulator useful only with another computing system that performs initialization and storage of results for monitoring. While the *autonomous* emulator is developed with a shared memory and can be used independently of any other system. For each of these structures we discussed different steps and transformations undergone in order to obtain the best results.

An application to the case of DC system is done and demonstrates the efficiency of this methodology. The development of this emulator is performed very easily using the defined approach and the previous study of the emulator structures.

The used SpecC methodology presents a simplified design process based on well-defined, clear and structured models at each exploration step. This enables quick exploration and synthesis.
The modular structure of the SpecC programs and the clear separation of communication and computation facilitate reuse of system components and enables easy integration of IP. This will give the SpecC methodology a powerful possibilities of extension using libraries of electronic components (memories, processors, communication protocols, ...) and of process modules (motor, converter, sensors, ...).

Using these libraries with specific tools for step automation will facilitate the design process and reduce further more the time-to-market. The user will be able in the near future to design her/his emulator and implement it (using for example FPGA circuits) in few hours without the need of any high qualification.

Development of these tools and libraries represent our main objectives for the future works.

Acknowledgments

The authors would like to thank the Fulbright Scholar Program for supporting this project. Also we would like to thank Andreas Gerstlauer and Rainer Doemer for help by their interesting comments and ideas.

References

[1] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, 2000

[2] A. Gerstlauer, R. Dömer, Junyu Peng, D. Gajski, "System Design: A Practical Guide with SpecC", Kluwer Academic Publishers, 2001

[3] S. Ben Saoud, J.C. Hapiot, "Parallel architectures applied to real time emulation", IECON2K IEEE International Conference on Industrial Electronics, Control and Instrumentation, October 22-28, 2000, pp1719-1724

[4] S. Ben Saoud, J.N. Contensou, J.C. Hapiot, "ASIC dedicated to real time emulation", SDEMPED'99 IEEE International Symposium on Diagnostics for Electrical Machines, Power Electronics and Drives, September 1-3,1999

[5] S. Ben Saoud, B. Dagues, J.C. Hapiot, "Universal emulator of static converter / electrical machines / sensors sets. Test of new control unit", CESA'98 Computational Engineering in Systems Applications, April 1-4,1998

[6] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak, "Design of a GSM Vocoder using SpecC Methodology", University of California, Irvine, Technical Report ICS-TR-99-11, February 1999

[7] Motorola, Inc., Semiconductor Products Sector, DSP Division, DSP 56600 16-bit Digital Signal Processor Family Manual, DSP56600FM/AD, 1996

[8] J. Peng, L. Cai, A. Selka, D. Gajski, "Design of a JBIG Encoder using SpecC Methodology", University of California, Irvine, Technical Report ICS-TR-00-13, June 2000

[9] Samsung Semiconductor Inc., North America, SRAM Products, "KM68257C", 1998

A Specification Model for the DC Emulator (Integer Form)

B Architecture Model for the DC Monitored - Emulator (Integer Form)

C Communication Model for the Monitored - Emulator (Integer Form)

D Architecture Model for the DC Autonomous - Emulator (Integer Form)

E Communication Model for the Autonomous - Emulator (Integer Form)

```
////////////////////////////////////////////////////////////
// SPEC_EMI_MCC1.sc                                       //
//                                                        //
//                  SPECIFICATION MODEL                   //
//                                                        //
//               EMULATOR DESIGN - INTEGER                //
//                                                        //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS      //
//                                                        //
// Slim Ben Saoud                                         //
// CECS-UCI July 2001                                     //
//                                                        //
////////////////////////////////////////////////////////////


#include <stdio.h>
#include <sim.sh>
#include <math.h>
#include <stdlib.h>

#include "typedef.sh"
#include "sync.sc"
#include "clk.sc"
#include "ctl.sc"
#include "master.sc"
#include "emul.sc"


//Testbench//////

behavior Main
    {

    event       SysClk;
    bool        stop;
    bit[1:0]    sc;
    int         gv, a, b, l, m, n, i_int, w_int;
    int         Nim;
    bit[1:0]    S;

    CSync       sync1, sync2;

    Clock       Clock1(SysClk, stop);
    Master      Master1(gv, a, b, l, m, n, stop, i_int, w_int, sync1, sync2);
    Command_rap Command_rap1(stop, SysClk, sc);
    Emulator    Emulator1(gv, a, b, l, m, n, stop, SysClk, sc, i_int, w_int, Nim, S, sync1, sync2);

    int main(void)
        {
        stop=false;

        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);

        puts("starting...");
        printf("Simulation time=%d us \n",sim_time1);

        par {
            Clock1.main();
            Master1.main();
            Command_rap1.main();
            Emulator1.main();
            }

        printf("im=%f \n", i_int/1024.);
        printf("vit=%f \n", w_int/1024.);
        puts("Exiting...");

        return(0);
        }
    };

//EOF

////////////////////////////////////////////////////////////
// MCC_COM_CNST.sh                                        //
//                                                        //
// SPECIFIACTION OF ELECTROMECHANICAL SYSTEM USING SPECC  //
// CASE OF A DC MACHINE WITH CURRENT AND SPEED SENSORS    //
//          WITH CONTROL DEVICE                           //
```



```
//                                                        //
// Slim Ben Saoud                                         //
// CECS-UCI April 2001                                    //
//                                                        //
////////////////////////////////////////////////////////////


// CONSTANTS AND MACROS
//////////////////////////

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  5000
#define CYCLESAVE   10
#define ALPHA       200

// Electromechanical system parameters

#define CYCLESEN1   5           // cycle de rafraichissement du courant
#define CYCLESEN2   1000        // cycle de rafraichissement de la vitesse
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30


FILE    *pTrace;

//Simulation time

int sim_time1=5000;


////////////////////////////////////////////////////////////
//  SYNC.sc                                               //
//                                                        //
//              SPEC_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////


//Interfaces and Channels for synchronization
////////////////////////////////////////////////

interface ISyncIn
    {
    void recv();
    };

interface ISyncOut
    {
    void send();
    };

channel CSync() implements ISyncIn, ISyncOut
    {
    bool    valid=false;
    event   e;

    void send()
        {
        valid=true;
        notify (e);
        }

    void recv()
        {
        if (!valid) wait(e);
        valid=false;
        }

    };

////////////////////////////////////////////////////////////
//  CLK.sc                                                //
//                                                        //
//              SPEC_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////
```

```
// SIMULATION CLOCK
//////////////////////

behavior Clock(out event clk, out bool stop)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while ((int)now()<sim_time1);

        stop=true;
        notify(clk);
        }
    };

////////////////////////////////////////////////////////////////
// CTL.sc                                                     //
//                                                            //
//              SPEC_EMI_MCC (July 2001)                      //
////////////////////////////////////////////////////////////////

behavior Command_rap(in bool stop, in event clk, out bit[1:0] cd)
    {
    int i=0;
    int alph=180;

    void main(void)
        {
        do
            {
            wait clk;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i>284) i=0;
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (!stop);
        }
    };


////////////////////////////////////////////////////////////////
// MASTER.sc                                                  //
//                                                            //
//              SPEC_EMI_MCC1 (July 2001)                     //
////////////////////////////////////////////////////////////////

//INITIALISATION
//////////////////

behavior Init(out int gv,out int a, out int b, out int l, out int m, out int n, ISyncOut sync1)
    {
    float   Alpha, Betta, Gamma, Londa, Mu, Psi;

    float   Ve=60.0;

    float   rm=1.0;
    float   lm=0.00400;
    float   km=0.184;

    float   fm=0.000180;
    float   jm=0.00160;
    float   fc=0.000180;
    float   jc=0.0016;
    float   Cs=0.16;

    float   ft, jt;
    float   hor=1e-6;
    bool    init_f=true;

    void main(void)
        {
```



```
        if (init_f)
            {
            ft=fm+fc;
            jt=jm+jc;
            Alpha=1.-hor*rm/lm*(1.-hor/2.*rm/lm);
            Betta=-hor*km/lm*(1.-hor/2.*rm/lm);
            Gamma=hor/lm*(1.-hor/2.*rm/lm);
            Londa=hor*km/jt*(1.-hor/2.*ft/jt);
            Mu=1.-hor*ft/jt*(1.-hor/2.*ft/jt);
            Psi=-Cs*hor/jt*(1.-hor/2.*ft/jt);


            a=(int)(1024*1024*(Alpha-1));
            b=(int)(1024*1024*Betta);
            gv=(int)(1024*128*Gamma*Ve);
            l=(int)(1024*1024*Londa);
            m=(int)(1024*1024*(Mu-1));
            n=(int)(1024*128*Psi);

            sync1.send();

            pTrace=fopen("R_spec_emi_mcc1.dat","w");

            init_f=false;
            }

        }
    };


behavior Storage(in bool stop, in event clk, in int i_int, in int w_int, ISyncIn sync2)
    {
    float   im, vit;

    void main(void)
        {
        do
            {
            sync2.recv();
            im=i_int/1024.;
            vit=w_int/1024.;
            fprintf(pTrace," %f  %f   \n",  im, vit);
            }while (!stop);

        fclose(pTrace);
        }
    };


//Master behavior
//////////////////

behavior Master(out int gv,out int a,out int b,out int l,out int m,out int n,
                in bool stop, in int i_int, in int w_int, ISyncOut sync1, ISyncIn sync2)
    {
    event       clk_s;

    Init        Init1(gv, a, b, l, m, n, sync1);
    Storage     Storage1(stop, clk_s, i_int, w_int, sync2);

    void main(void)
        {
        Init1.main();
        Storage1.main();
        }
    };


////////////////////////////////////////////////////////////////
// EMUL.sc                                                    //
//                                                            //
//              SPEC_EMI_MCC (July 2001)                      //
////////////////////////////////////////////////////////////////

#include    "motor.sc"
#include    "sensors.sc"
#include    "stor_send.sc"


// Electromechanical system global module
```

```
behavior Emulator(in int gv,in int a,in int b,in int 1,in int m, in int n,
                  in bool stop, in event clk, in bit[1:0] cd,
                  inout int i_int, inout int w_int, out int Nim, out bit[1:0] S, ISyncIn sync1, ISyncO
ut sync2)
    {
    Motor           Motor1(gv, a, b, 1, m, cd, i_int, w_int);
    Sen_cur         Sen_cur1(i_int, Nim);
    Sen_speed       Sen_speed1(w_int, S);
    Storage_send    Storage_send1(sync2);

    void main(void)
        {
        i_int= w_int= 0;
        sync1.recv();

        do
            {
            wait clk;
            Motor1.main();
            Sen_cur1.main();
            Sen_speed1.main();
            Storage_send1.main();
            }while (!stop);
        }
    };

////////////////////////////////////////////////////////////
// MOTOR.sc                                                //
//                                                         //
//              SPEC_EMI_MCC (July 2001)                   //
////////////////////////////////////////////////////////////


//ELECTROMECHANICAL SYSTEM
/////////////////////////////

behavior Converter(in int gv, in bit[1:0] cd, out int Vout)
    {
    int a,b;
    void main(void)
        {
        a=cd[0];
        b=cd[1];
        Vout=gv*(a-b);
        }
    };

behavior Electric(in int a,in int b,in int Vout, in int w_int, inout int i_int)
    {
    int di_int=0;

    void main(void)
        {
        di_int=(((a*(i_int>>3))>>10)+((b*(w_int>>3))>>10)+Vout);
        i_int=i_int+di_int/128;
        }
    };

behavior Mecanic(in int 1,in int m,in int i_int, inout int w_int)
    {
    int dw_int=0;
    int dw_int1=0;

    void main(void)
        {
        dw_int=(1*i_int+m*w_int)+dw_int;
        dw_int1=dw_int>>20;
        dw_int=dw_int-(dw_int1<<20);
        w_int=w_int+dw_int1;
        }
    };


behavior Motor(in int gv,in int a,in int b,in int 1,in int m, in bit[1:0] cd,
               inout int i_int, inout int w_int)
    {
    int Vout;

    Converter Converter1(gv, cd, Vout);
```



```
    Electric Electric1(a, b, Vout, w_int, i_int);
    Mecanic Mecanic1(1, m, i_int, w_int);

    void main(void)
        {
        Converter1.main();
        Electric1.main();
        Mecanic1.main();
        }
    };

////////////////////////////////////////////////////////////
// SENSORS.sc                                              //
//                                                         //
//              SPEC_EMI_MCC (July 2001)                   //
////////////////////////////////////////////////////////////

// SENSORS MODULES
////////////////////

// Current sensor Generator

behavior Sen_cur(in int i_int, out int Nim)
    {
    int sens1_i=0;
    void main(void)
        {
        sens1_i=sens1_i+1;
        if (sens1_i==CYCLESEN1)
            {
            Nim=(int)(((p/1024)*i_int+p*imax)/(2*imax));
            sens1_i=0;
            }
        }
    };


behavior Sen_speed(in int w_int, out bit[1:0] S)
    {
    int w_inti;
    int sens2_i=0;
    int i=0;

    int T4_vit=24000;
    bool s_vit=true;

    void main(void)
        {
        sens2_i=sens2_i+1;

        if (sens2_i==CYCLESEN2)
            {
            w_inti=w_int;
            if (w_inti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(w_inti)<512) w_inti=512;          //Limite basse de vitesse a reproduire

            T4_vit=(int)((pi*1e6*1024)/(n_cio*w_inti));     //quart de periode en us

            sens2_i=0;
            }

        if (i <= T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            if (T4_vit < i && i <= 2*T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (2*T4_vit < i && i  <= 3*T4_vit)
                {
                S[0]=0;
```

```
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (3*T4_vit < i && i  <= 4*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            i=i+1;
            if (i>4*T4_vit) i=0;
        }
    };


////////////////////////////////////////////////////////////
// STOR_SEND.sc                                           //
//                                                        //
//              SPEC_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////

// Storage_send behavior

behavior Storage_send(ISyncOut sync2)
    {
    int stor_i=0;

    void main(void)
        {
        stor_i=stor_i+1;
        if (stor_i==CYCLESAVE)
            {
            sync2.send();
            stor_i=0;
            }
        }
    };
```

```
////////////////////////////////////////////////////////////////
// ARCH1_EMI_MCC.sc                                           //
//                                                            //
//                    ARCHITECTURE MODEL                      //
//                                                            //
//                EMULATOR DESIGN _ INTEGER                   //
//INTERRUPT OF MASTER performing only INIT/STORAGE Processes//
//                                                            //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS          //
//                                                            //
// Slim Ben Saoud                                             //
// CECS-UCI July 2001                                         //
//                                                            //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <sim.sh>
#include <math.h>
#include <stdlib.h>
#include <memory.h>


#include "typedef.sh"
#include "sync.sc"
#include "channels.sc"
#include "bus.sc"

#include "clk.sc"

#include "ctl.sc"
#include "master.sc"
#include "emul.sc"


//Testbench//////

behavior Main
    {

    event       SysClk;
    bool        stop;
    bit[1:0]    sc;
    int         Nim;
    bit[1:0]    S;

    Bus0        bus0;

    Clock       Clock1(SysClk, stop);
    Master      Master1(stop, bus0);
    Command_rap Command_rap1(stop, SysClk, sc);
    Emulator    Emulator1(stop, SysClk, sc, Nim, S, bus0);

    int main(void)
        {
        stop=false;

        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);

        puts("starting...");
        printf("Simulation time=%d us \n",sim_time1);

        par {
            Clock1.main();
            Master1.main();
            Command_rap1.main();
            Emulator1.main();
            }


        puts("Exiting...");

        return(0);
        }
    };

//EOF


////////////////////////////////////////////////////////////////
```



```
// TYPEDEF.sh
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////


// CONSTANTS AND MACROS
//////////////////////////

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  5000
#define CYCLESAVE   10
#define ALPHA       200

// Electromechanical system parameters

#define CYCLESEN1   5           // cycle de rafraichissement du courant
#define CYCLESEN2   1000        // cycle de rafraichissement de la vitesse
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30


FILE    *pTrace;

// Simulation time

int sim_time1;


////////////////////////////////////////////////////////////////
//  SYNC.sc                                                   //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////


//Interfaces and Channels for synchronization
////////////////////////////////////////////////

interface ISyncIn
    {
    void recv();
    };

interface ISyncOut
    {
    void send();
    };

channel CSync() implements ISyncIn, ISyncOut
    {
    bool    valid=false;
    event   e;

    void send()
        {
        valid=true;
        notify (e);
        }

    void recv()
        {
        if (!valid) wait(e);
        valid=false;
        }

    };

////////////////////////////////////////////////////////////////
// CHANNELS.sc                                                //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////
```

```
interface ISendWord
{
  void send(int data);
};

interface IRecvWord
{
  void recv(int* data);
};


channel CWord()
  implements ISendWord, IRecvWord
{
  int  buf;
  bool    valid = false;
  event   ev;


  void send(int data)
  {
    buf = data;
    valid = true;
    notify (ev);
  }

  void recv(int* data)
  {
    if (!valid) wait (ev);
    *data = buf;
    valid = false;
  }
};



/* ------------------------------------------------------------------ */

interface ISendWordP
{
  void send(int* data, int len);
};

interface IRecvWordP
{
  void recv(int* data, int len);
};


channel CWordP()
  implements ISendWordP, IRecvWordP
{
  int* buf = 0;
  event   ev;


  void send(int* data, int len)
  {
    buf = (int*)malloc(sizeof(int) * len);
    memcpy(buf, data, sizeof(int) * len);
    notify (ev);
  }

  void recv(int* data, int len)
  {
    if (!buf) wait (ev);
    memcpy(data, buf, sizeof(int) * len);
    free(buf);
    buf = 0;
  }
};


////////////////////////////////////////////////////////////////
// BUS.sc                                                     //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////
```



```
interface IPE0bus0
    {
    void recv_param(int param[6]);
    void send_result(int result[2]);
    };

interface IPE1bus0
    {
    void send_param(int param[6]);
    void recv_result(int result[2]);
    };


channel Bus0() implements   IPE0bus0, IPE1bus0
    {
    CWordP  Cparam;
    CWordP  Cresult;

    void send_param(int param[6]) {Cparam.send(param, 6);}
    void recv_param(int param[6]) {Cparam.recv(param, 6);}
    void send_result(int result[2]) {Cresult.send(result, 2);}
    void recv_result(int result[2]) {Cresult.recv(result, 2);}
    };


////////////////////////////////////////////////////////////////
//  CLK.sc                                                    //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////

// SIMULATION CLOCK
//////////////////////

behavior Clock(out event clk, out bool stop)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while ((int)now()<sim_time1);

        stop=true;
        notify(clk);
        }
    };

////////////////////////////////////////////////////////////////
// CTL.sc                                                     //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////

behavior Command_rap(in bool stop, in event clk, out bit[1:0] cd)
    {
    int i=0;
    int alph=180;

    void main(void)
        {
        do
            {
            wait clk;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i>284) i=0;
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (!stop);
        }
    };


////////////////////////////////////////////////////////////////
// MASTER.sc                                                  //
```

```
//                                                                  //
//          Arch_Emi_Mcc (July 2001)                                //
//////////////////////////////////////////////////////////////////////


//INITIALISATION
//////////////////

behavior Init(IPE1bus0 bus0)
    {
    float   Alpha, Betta, Gamma, Londa, Mu, Psi;
    int     gv, a, b, l, m, n;

    float   Ve=60.0;

    float   rm=1.0;
    float   lm=0.00400;
    float   km=0.184;

    float   fm=0.000180;
    float   jm=0.00160;
    float   fc=0.000180;
    float   jc=0.0016;
    float   Cs=0.16;

    float   ft, jt;
    float   hor=1e-6;
    bool    init_f=true;

    int     param[6];

    void main(void)
        {
        if (init_f)
            {
            ft=fm+fc;
            jt=jm+jc;
            Alpha=1.-hor*rm/lm*(1.-hor/2.*rm/lm);
            Betta=-hor*km/lm*(1.-hor/2.*rm/lm);
            Gamma=hor/lm*(1.-hor/2.*rm/lm);
            Londa=hor*km/jt*(1.-hor/2.*ft/jt);
            Mu=1.-hor*ft/jt*(1.-hor/2.*ft/jt);
            Psi=-Cs*hor/jt*(1.-hor/2.*ft/jt);


            gv=(int)(1024*128*Gamma*Ve);
            a=(int)(1024*1024*(Alpha-1));
            b=(int)(1024*1024*Betta);
            l=(int)(1024*1024*Londa);
            m=(int)(1024*1024*(Mu-1));
            n=(int)(1024*128*Psi);

            param[0]=gv;
            param[1]=a;
            param[2]=b;
            param[3]=l;
            param[4]=m;
            param[5]=n;

            bus0.send_param(param);

            pTrace=fopen("R_arch_emi_mcc.dat","w");

            init_f=false;
            }

        }
    };


behavior Storage(in bool stop, IPE1bus0 bus0)
    {
    int     result[2];
    float   im, vit;

    void main(void)
        {
        do
            {
            bus0.recv_result(result);
```



```
            im=result[0]/1024.;
            vit=result[1]/1024.;
            fprintf(pTrace," %f  %f   \n",  im, vit);
            }while (!stop);

        fclose(pTrace);
        }
    };


//Master behavior
//////////////////

behavior Master(in bool stop, IPE1bus0 bus0)
    {
    event       clk_s;

    Init        Init1(bus0);
    Storage     Storage1(stop, bus0);

    void main(void)
        {
        Init1.main();
        Storage1.main();
        }
    };


//////////////////////////////////////////////////////////////////
// EMUL.sc                                                      //
//                                                              //
//          Arch_Emi_Mcc (July 2001)                            //
//////////////////////////////////////////////////////////////////

#include    "motor.sc"
#include    "sensors.sc"
#include    "stor_send.sc"

behavior Em_init(IPE0bus0 bus0, out int gv,out int a, out int b, out int l, out int m, out int n)
    {
    int param[6];

    void main(void)
        {
            bus0.recv_param(param);

            gv=param[0];
            a=param[1];
            b=param[2];
            l=param[3];
            m=param[4];
            n=param[5];
        }
    };

// Electromechanical system global module

behavior Emulator(in bool stop, in event clk, in bit[1:0] cd, out int Nim, out bit[1:0] S, IPE0bus0
bus0)
    {
    int i_int, w_int;
    int gv, a, b, l, m, n;

    Em_init         Em_init1(bus0, gv, a, b, l, m, n);
    Motor           Motor1(gv, a, b, l, m, cd, i_int, w_int);
    Sen_cur         Sen_cur1(i_int, Nim);
    Sen_speed       Sen_speed1(w_int, S);
    Storage_send    Storage_send1(i_int, w_int, bus0);

    void main(void)
        {
        i_int= w_int= 0;
        Em_init1.main();

        do
            {
            wait clk;
            Motor1.main();
            Sen_cur1.main();
            Sen_speed1.main();
```

```
            Storage_send1.main();
            }while (!stop);

        printf("im=%f \n", i_int/1024.);
        printf("vit=%f \n", w_int/1024.);
        }
    };

////////////////////////////////////////////////////////////////
// MOTOR.sc                                                   //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////


//ELECTROMECHANICAL SYSTEM
//////////////////////////

behavior Converter(in int gv, in bit[1:0] cd, out int Vout)
    {
    int a,b;
    void main(void)
        {
        a=cd[0];
        b=cd[1];
        Vout=gv*(a-b);
        }
    };

behavior Electric(in int a,in int b,in int Vout, in int w_int, inout int i_int)
    {
    int di_int=0;

    void main(void)
        {
        di_int=(((a*(i_int>>3))>>10)+((b*(w_int>>3))>>10)+Vout);
        i_int=i_int+di_int/128;
        }
    };

behavior Mecanic(in int l,in int m,in int i_int, inout int w_int)
    {
    int dw_int=0;
    int dw_int1=0;

    void main(void)
        {
        dw_int=(l*i_int+m*w_int)+dw_int;
        dw_int1=dw_int>>20;
        dw_int=dw_int-(dw_int1<<20);
        w_int=w_int+dw_int1;
        }
    };


behavior Motor(in int gv,in int a,in int b,in int l,in int m, in bit[1:0] cd,
               inout int i_int, inout int w_int)
    {
    int Vout;

    Converter Converter1(gv, cd, Vout);
    Electric Electric1(a, b, Vout, w_int, i_int);
    Mecanic Mecanic1(l, m, i_int, w_int);

    void main(void)
        {
        Converter1.main();
        Electric1.main();
        Mecanic1.main();
        }
    };

////////////////////////////////////////////////////////////////
// SENSORS.sc                                                 //
//                                                            //
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////

// SENSORS MODULES
//////////////////
```



```
// Current sensor Generator

behavior Sen_cur(in int i_int, out int Nim)
    {
    int sens1_i=0;
    void main(void)
        {
        sens1_i=sens1_i+1;
        if (sens1_i==CYCLESEN1)
            {
            Nim=(int)(((p/1024)*i_int+p*imax)/(2*imax));
            sens1_i=0;
            }
        }
    };


behavior Sen_speed(in int w_int, out bit[1:0] S)
    {
    int sens2_i=0;
    int i=0;
    int w_inti;
    int T4_vit=24000;
    bool s_vit=true;

    void main(void)
        {
        sens2_i=sens2_i+1;

        if (sens2_i==CYCLESEN2)
            {
            w_inti=w_int;
            if (w_inti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(w_inti)<512) w_inti=512;          //Limite basse de vitesse a reproduire

            T4_vit=(int)((pi*1e6*1024)/(n_cio*w_inti));     //quart de periode en us

            sens2_i=0;
            }

        if (i <= T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            if (T4_vit < i && i <= 2*T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (2*T4_vit < i && i  <= 3*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (3*T4_vit < i && i  <= 4*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            i=i+1;
            if (i>4*T4_vit) i=0;
        }
    };


////////////////////////////////////////////////////////////////
// STOR_SEND.sc                                               //
//                                                            //
```

```
//          Arch_Emi_Mcc (July 2001)                          //
////////////////////////////////////////////////////////////////

// Storage_send behavior

behavior Storage_send(in int i_int, in int w_int, IPE0bus0 bus0)
    {
    int stor_i=0;
    int result[2];

    void main(void)
        {
        stor_i=stor_i+1;
        if (stor_i==CYCLESAVE)
            {
            result[0]=i_int;
            result[1]=w_int;
            bus0.send_result(result);
            stor_i=0;
            }
        }
    };
```

```
////////////////////////////////////////////////////////////
// COM1_EMI_MCC1.sc                                       //
//                                                        //
//                  COMMUNICATION MODEL                   //
//                                                        //
//              EMULATOR DESIGN _ INTEGER                 //
//INTERRUPT OF MASTER performing only INIT/STORAGE Processes//
//                                                        //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS      //
//                                                        //
// Slim Ben Saoud                                         //
// CECS-UCI July 2001                                     //
//                                                        //
////////////////////////////////////////////////////////////

#include <stdio.h>
#include <sim.sh>
#include <math.h>
#include <stdlib.h>
#include <memory.h>


#include "typedef.sh"
#include "sync.sc"
#include "clk.sc"
#include "ctl.sc"
#include "master.sc"
#include "emul.sc"


//Testbench//////

behavior Main
    {
    bit[15:0]   A;          // address
    bit[23:0]   D;          // data
    event       MCS;        // chip select
    bool        nRD;        // control lines
    bool        nWR;
    event   intC;


    event       SysClk;
    bool        stop;
    bit[1:0]    sc;
    int         Nim;
    bit[1:0]    S;

    Clock       Clock1(SysClk, stop);
    Master      Master1(stop, A, D, MCS, nRD, nWR, intC);
    Command_rap Command_rap1(stop, SysClk, sc);
    Emulator    Emulator1(stop, SysClk, sc, Nim, S, A, D, MCS, nRD, nWR, intC);

    int main(void)
        {
        stop=false;

        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);

        puts("starting...");
        printf("Simulation time=%d us \n",sim_time1);

        sim_time1=sim_time1*1000;       //new cycletime is given in ns//

        par {
            Clock1.main();
            Master1.main();
            Command_rap1.main();
            Emulator1.main();
            }


        puts("Exiting...");

        return(0);
        }
    };

//EOF
```



```
////////////////////////////////////////////////////////////
// TYPEDEF.sh                                             //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

// CONSTANTS AND MACROS
//////////////////////////

// Time references

#define CYCLETIME   1000
#define CYCLENB     50
#define MAXSIMTIME  50000000
#define CYCLESAVE   10
#define ALPHA       200

// Electromechanical system parameters

#define CYCLESEN1   5           // cycle de rafraichissement du courant
#define CYCLESEN2   1000        // cycle de rafraichissement de la vitesse
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30


FILE    *pTrace;

#define ADDRESS     0xAF00

#define WS 2       // wait states
#define DSP_CLOCK_PERIOD     100/6       /* 60 MHz = 16.67ns */
#define HW_CLOCK_PERIOD      10          /* 100 MHz = 10ns   */

typedef int word24;

//Simulation time

int sim_time1=5000;


////////////////////////////////////////////////////////////
//  SYNC.sc                                               //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////


//Interfaces and Channels for synchronization
////////////////////////////////////////////////

interface ISyncIn
    {
    void recv();
    };

interface ISyncOut
    {
    void send();
    };

channel CSync() implements ISyncIn, ISyncOut
    {
    bool    valid=false;
    event   e;

    void send()
        {
        valid=true;
        notify (e);
        }

    void recv()
        {
        if (!valid) wait(e);
        valid=false;
```

```
        }

    };


////////////////////////////////////////////////////////////
//  INT_CLK.sc                                            //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

// SIMULATION CLOCK
//////////////////////

behavior Clock(out event clk, out bool stop)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while ((int)now()<sim_time1);

        stop=true;
        notify(clk);
        }
    };

////////////////////////////////////////////////////////////
// INT_CTL.sc                                             //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

behavior Command_rap(in bool stop, in event clk, out bit[1:0] cd)
    {
    int i=0;
    int alph=180;

    void main(void)
        {
        do
            {
            wait clk;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i>284) i=0;
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (!stop);
        }
    };

////////////////////////////////////////////////////////////
// MASTER.sc                                              //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

#include "sw_protocol.sc"
#include "sw_bus.sc"


//INITIALISATION
/////////////////

behavior Init(IBusMaster bus0)
    {
    float   Alpha, Betta, Gamma, Londa, Mu, Psi;
    int     gv, a, b, l, m, n;

    float   Ve=60.0;

    float   rm=1.0;
    float   lm=0.00400;
    float   km=0.184;
```



```
    float   fm=0.000180;
    float   jm=0.00160;
    float   fc=0.000180;
    float   jc=0.0016;
    float   Cs=0.16;

    float   ft, jt;
    float   hor=1e-6;
    bool    init_f=true;

    int     param[6];

    void main(void)
        {
        if (init_f)
            {
            ft=fm+fc;
            jt=jm+jc;
            Alpha=1.-hor*rm/lm*(1.-hor/2.*rm/lm);
            Betta=-hor*km/lm*(1.-hor/2.*rm/lm);
            Gamma=hor/lm*(1.-hor/2.*rm/lm);
            Londa=hor*km/jt*(1.-hor/2.*ft/jt);
            Mu=1.-hor*ft/jt*(1.-hor/2.*ft/jt);
            Psi=-Cs*hor/jt*(1.-hor/2.*ft/jt);


            gv=(int)(1024*128*Gamma*Ve);
            a=(int)(1024*1024*(Alpha-1));
            b=(int)(1024*1024*Betta);
            l=(int)(1024*1024*Londa);
            m=(int)(1024*1024*(Mu-1));
            n=(int)(1024*128*Psi);

            param[0]=gv;
            param[1]=a;
            param[2]=b;
            param[3]=l;
            param[4]=m;
            param[5]=n;

            bus0.Master_sendBW(param, 6, ADDRESS);

            pTrace=fopen("R_com_emi_mcc1.dat","w");

            init_f=false;
            }

        }
    };


behavior Storage(in bool stop, IBusMaster bus0)
    {
    int     result[2];
    float   im, vit;

    void main(void)
        {
        do
            {
            bus0.Master_recvBW(result, 2, ADDRESS);
            im=result[0]/1024.;
            vit=result[1]/1024.;
            fprintf(pTrace," %f  %f   \n",  im, vit);
            }while (!stop);

        fclose(pTrace);
        }
    };


behavior SwMain(in bool stop, IBusMaster bus0)
    {
    word24      donnee=0x34A7, donnee1=0xFFE4;
    int         donnee2, donnee3;

    Init    Init1(bus0);
    Storage Storage1(stop, bus0);
```

```
    void main(void)
        {
        Init1.main();
        Storage1.main();
        }
    };


//Internal generation of synchronisation signal
////////////////////////////////////////////////

behavior IntHandler(ISyncOut ev)
    {
    void main(void)
        {
        ev.send();
        }
    };

//Master behavior
/////////////////

behavior Master(in bool stop,
                bit[15:0]   A,          // address
                bit[23:0]   D,          // data
                event       MCS,        // chip select
                bool        nRD,        // control lines
                bool        nWR,
                in event    intC)
    {
    CSync       intCflag;
    IntHandler  IntChandler(intCflag);

    MasterBus   bus0(A, D, MCS, nRD, nWR, intCflag);
    SwMain      SwMain1(stop, bus0);


    void main(void)
        {
        try
            {
            SwMain1.main();
            }
        interrupt(intC)
            {
            IntChandler.main();
            }
        }
    );

////////////////////////////////////////////////////////////////
//  SW_PROTOCOL.sc                                            //
//                                                            //
//              COM_EMI_MCC1 (July 2001)                      //
////////////////////////////////////////////////////////////////


//PROTOCOL CHANNAEL
/////////////////////

interface IProtocolMaster
    {
    word24 Master_read(bit[15:0] addr);
    void Master_write(bit[15:0] addr, word24 data);
    };


/*----------------- I/O instructions of DSP56600 (MOVEM) ---------------- */

channel MasterProtocol( bit[15:0]   A,          // address
                        bit[23:0]   D,          // data
                        event       MCS,        // chip select
                        bool        nRD,        // control lines
                        bool        nWR)

  implements IProtocolMaster
    {
    word24 Master_read(bit[15:0] addr)
```



```
        {
        word24 data;

        waitfor(5 * DSP_CLOCK_PERIOD / 2);
        A = addr;
        notify(MCS);
        waitfor(DSP_CLOCK_PERIOD / 2);              // 0.5T-4.0 = (4.3ns,-)
        nRD = 0;
        waitfor((2*WS + 1) * DSP_CLOCK_PERIOD / 2); // (WS+0.5)T-8.5 = (16.5ns,-)
        data = D;
        waitfor(DSP_CLOCK_PERIOD / 2);
        nRD = 1;                                    // nRD pulse width: (WS+0.25)T-3.8 = (17ns,-)
        waitfor(WS * DSP_CLOCK_PERIOD);

        return data;
        }

    void Master_write(bit[15:0] addr, word24 data)
        {
        waitfor(5 * DSP_CLOCK_PERIOD / 2);
        A = addr;
        notify(MCS);
        waitfor(DSP_CLOCK_PERIOD / 4);          // 0.25T-3.7 = (0.5ns,-)
        nWR = 0;
        waitfor(3 * DSP_CLOCK_PERIOD / 4);      // 0.75T-3.7 = (8.8ns,-)
        D = data;
        waitfor((WS + 1) * DSP_CLOCK_PERIOD);
        nWR = 1;                                // nWR pulse width: 1.5T-5.7 = (19.3ns,-)
        waitfor(WS * DSP_CLOCK_PERIOD / 2);
    }

};

////////////////////////////////////////////////////////////////
//  SW_BUS.sc                                                 //
//                                                            //
//              COM_EMI_MCC1 (July 2001)                      //
////////////////////////////////////////////////////////////////

//APPLICATION LAYER
///////////////////////

interface IBusMaster
    {
    void    Master_sendW(word24 data, int   addr);
    void    Master_sendBW(word24* data,int len, int addr);
    void    Master_recvW(word24* data, int addr);
    void    Master_recvBW(word24* data,int len, int addr);
    };

channel MasterBus(  bit[15:0]   A,          // address
                    bit[23:0]   D,          // data
                    event       MCS,        // chip select
                    bool        nRD,        // control lines
                    bool        nWR,
                    ISyncIn     intC)

    implements IBusMaster

    {
    MasterProtocol  protocol(A, D, MCS, nRD, nWR);
    void Master_sendW(word24 data, int addr)
        {
        intC.recv();
        protocol.Master_write(addr, data);
        }

    void Master_sendBW(word24* data, int len, int addr)
        {
        int i;
        for(i=0; i<len; i++)
            {
            Master_sendW(data[i], addr);
            }
        }

    void Master_recvW(word24* data, int addr)
        {
        intC.recv();
        *data=protocol.Master_read(addr);
```

```
        }

    void Master_recvBW(word24* data, int len, int addr)
        {
        int i;
        for(i=0; i<len; i++)
            {
            Master_recvW(&data[i], addr);
            }
        }

    };

////////////////////////////////////////////////////////////
// EMUL.sc                                                //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

#include    "hw_protocol.sc"
#include    "hw_bus.sc"
#include    "motor.sc"
#include    "sensors.sc"

behavior Em_init(IBusSlave bus0, out int gv,out int a, out int b, out int l, out int m, out int n)
    {
    int param[6];

    void main(void)
        {
            bus0.Slave_recvBW(param, 6, ADDRESS);

            gv=param[0];
            a=param[1];
            b=param[2];
            l=param[3];
            m=param[4];
            n=param[5];
        }
    };

// Storage_send behavior

behavior Storage_send(in int i_int, in int w_int, IBusSlave bus0)
    {
    int stor_i=0;
    int result[2];

    void main(void)
        {
        stor_i=stor_i+1;
        if (stor_i==CYCLESAVE)
            {
            result[0]=i_int;
            result[1]=w_int;
            bus0.Slave_sendBW(result, 2, ADDRESS);
            stor_i=0;
            }
        }
    };

// Electromechanical system global module

behavior Emulator(in bool stop, in event clk, in bit[1:0] cd, out int Nim, out bit[1:0] S,
                  bit[15:0]   A,          // address
                  bit[23:0]   D,          // data
                  event       MCS,        // chip select
                  bool        nRD,        // control lines
                  bool        nWR,
                  out event   intC)
    {
    int i_int, w_int;
    int gv, a, b, l, m, n;

    SlaveBus    bus0(A, D, MCS, nRD, nWR, intC);

    Em_init         Em_init1(bus0, gv, a, b, l, m, n);
    Motor           Motor1(gv, a, b, l, m, cd, i_int, w_int);
    Sen_cur         Sen_cur1(i_int, Nim);
```



```
    Sen_speed       Sen_speed1(w_int, S);
    Storage_send    Storage_send1(i_int, w_int, bus0);

    void main(void)
        {
        i_int= w_int= 0;
        Em_init1.main();

        do
            {
            wait clk;
            Motor1.main();
            Sen_cur1.main();
            Sen_speed1.main();
            Storage_send1.main();
            }while (!stop);

        printf("im=%f \n", i_int/1024.);
        printf("vit=%f \n", w_int/1024.);
        }
    };

////////////////////////////////////////////////////////////
//  hw_protocol.sc                                        //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

//PROTOCOL CHANNAEL
//////////////////////

interface IProtocolSlave
    {
    word24 Slave_read(bit[15:0] addr);
    void Slave_write(bit[15:0] addr, word24 data);
    };

channel SlaveProtocol(  bit[15:0]   A,          // address
                        bit[23:0]   D,          // data
                        event       MCS,        // chip select
                        bool        nRD,        // control lines
                        bool        nWR)

  implements IProtocolSlave
    {
    word24 Slave_read(bit[15:0] addr)
        {
        word24 data;

        t1: wait(MCS);
        if (A != addr) goto t1;
        waitfor(20);
        if (nWR) goto t1;
        data = D;

        return data;
        }

    void Slave_write(bit[15:0] addr, word24 data)
        {
        t1: wait(MCS);
        if (A != addr) goto t1;
        waitfor(15);
        if (nRD) goto t1;
        D = data;
        }
};

////////////////////////////////////////////////////////////
//  hw_bus.sc                                             //
//                                                        //
//              COM_EMI_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////

interface IBusSlave
    {
    void    Slave_sendW(word24 data, int    addr);
    void    Slave_sendBW(word24* data, int len, int addr);
    void    Slave_recvW(word24* data, int addr);
    void    Slave_recvBW(word24* data, int len, int addr);
```

```
    };
channel SlaveBus(   bit[15:0]   A,          // address
                    bit[23:0]   D,          // data
                    event       MCS,        // chip select
                    bool        nRD,        // control lines
                    bool        nWR,
                    out event   intC)

    implements IBusSlave
    {
    SlaveProtocol   protocol(A, D, MCS, nRD, nWR);

    void Slave_sendW(word24 data, int addr)
        {
        notify(intC);
        protocol.Slave_write(addr, data);
        }


    void Slave_sendBW(word24* data, int len, int addr)
        {
        int i;
        for(i=0; i<len; i++)
            {
            Slave_sendW(data[i], addr);
            }
        }

    void Slave_recvW(word24* data, int addr)
        {
        notify(intC);
        *data=protocol.Slave_read(addr);
        }


    void Slave_recvBW(word24* data, int len, int addr)
        {
        int i;
        for(i=0; i<len; i++)
            {
            Slave_recvW(&data[i], addr);
            }
        }

    };

////////////////////////////////////////////////////////////////
// MOTOR.sc                                                   //
//                                                            //
//              COM_EMI_MCC1 (July 2001)                      //
////////////////////////////////////////////////////////////////


//ELECTROMECHANICAL SYSTEM
/////////////////////////////

behavior Converter(in int gv, in bit[1:0] cd, out int Vout)
    {
    int a,b;
    void main(void)
        {
        a=cd[0];
        b=cd[1];
        Vout=gv*(a-b);
        }
    };

behavior Electric(in int a,in int b,in int Vout, in int w_int, inout int i_int)
    {
    int di_int=0;

    void main(void)
        {
        di_int=(((a*(i_int>>3))>>10)+((b*(w_int>>3))>>10)+Vout);
        i_int=i_int+di_int/128;
        }
    };

behavior Mecanic(in int l,in int m,in int i_int, inout int w_int)
    {
```



```
    int dw_int=0;
    int dw_int1=0;

    void main(void)
        {
        dw_int=(l*i_int+m*w_int)+dw_int;
        dw_int1=dw_int>>20;
        dw_int=dw_int-(dw_int1<<20);
        w_int=w_int+dw_int1;
        }
    };


behavior Motor(in int gv,in int a,in int b,in int l,in int m, in bit[1:0] cd,
               inout int i_int, inout int w_int)
    {
    int Vout;

    Converter Converter1(gv, cd, Vout);
    Electric Electric1(a, b, Vout, w_int, i_int);
    Mecanic Mecanic1(l, m, i_int, w_int);

    void main(void)
        {
        Converter1.main();
        Electric1.main();
        Mecanic1.main();
        }
    };

////////////////////////////////////////////////////////////////
// SENSORS.sc                                                 //
//                                                            //
//              COM_EMI_MCC1 (July 2001)                      //
////////////////////////////////////////////////////////////////

// SENSORS MODULES
////////////////////

// Current sensor Generator

behavior Sen_cur(in int i_int, out int Nim)
    {
    int sens1_i=0;
    void main(void)
        {
        sens1_i=sens1_i+1;
        if (sens1_i==CYCLESEN1)
            {
            Nim=(int)(((p/1024)*i_int+p*imax)/(2*imax));
            sens1_i=0;
            }
        }
    };


behavior Sen_speed(in int w_int, out bit[1:0] S)
    {
    int sens2_i=0;
    int i=0;
    int w_inti;
    int T4_vit=24000;
    bool s_vit=true;

    void main(void)
        {
        sens2_i=sens2_i+1;

        if (sens2_i==CYCLESEN2)
            {
            w_inti=w_int;
            if (w_inti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(w_inti)<512) w_inti=512;           //Limite basse de vitesse a reproduire

            T4_vit=(int)((pi*1e6*1024)/(n_cio*w_inti));     //quart de periode en us

            sens2_i=0;
            }
```

```
        if (i <= T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            if (T4_vit < i && i <= 2*T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (2*T4_vit < i && i  <= 3*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (3*T4_vit < i && i  <= 4*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            i=i+1;
            if (i>4*T4_vit) i=0;
        }
    };
```

```
////////////////////////////////////////////////////////////////
// ARCH2_EMI_MCC.sc                                           //
//                                                            //
//                  ARCHITECTURE MODEL MODEL                  //
//                                                            //
//               EMULATOR DESIGN _ INTEGER                    //
//          AUTONOMOUS STRUCTURE (INCLUDING MEMORY)           //
//                                                            //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS          //
//                                                            //
// Slim Ben Saoud                                             //
// CECS-UCI July 2001                                         //
//                                                            //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <sim.sh>
#include <math.h>
#include <stdlib.h>
#include <memory.h>


#include "typedef.sh"
#include "sync.sc"
#include "channels.sc"
//#include "bus.sc"
#include "clk.sc"
#include "memory.sc"
#include "ctl.sc"
#include "master.sc"
#include "emul.sc"


//Testbench//////

behavior Main
    {

    event       SysClk;
    bool        stop;
    bit[1:0]    sc;
    int         Nim;
    bit[1:0]    S;
    bool        done;

    CSync       HS_sync1, HS_sync2;
    Mem_Access  m_access;

    Clock       Clock1(SysClk, stop);
    Master      Master1(HS_sync1, HS_sync2, m_access, done);
    Command_rap Command_rap1(stop, SysClk, sc);
    Emulator    Emulator1(HS_sync1, HS_sync2, m_access, stop, SysClk, sc, Nim, S);
    memory      memory1(m_access, done);

    int main(void)
        {
        stop=false;
        done=false;

        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);

        puts("starting...");
        printf("Simulation time=%d us \n",sim_time1);

        par {
            Clock1.main();
            Master1.main();
            Command_rap1.main();
            Emulator1.main();
            memory1.main();
            }


        puts("Exiting...");

        return(0);
        }
    };
```



```
//EOF


////////////////////////////////////////////////////////////////
// typedef.sh                                                 //
//                  ARCH2_EMI_MCC (July 2001)                 //
////////////////////////////////////////////////////////////////


// CONSTANTS AND MACROS
//////////////////////////

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  5000
#define CYCLESAVE   10
#define ALPHA       200

// Electromechanical system parameters

#define CYCLESEN1   5           // cycle de rafraichissement du courant
#define CYCLESEN2   1000        // cycle de rafraichissement de la vitesse
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30


FILE    *pTrace;

//Simulation time

int sim_time1=5000;


////////////////////////////////////////////////////////////////
//  SYNC.sc                                                   //
//                  ARCH2_EMI_MCC (July 2001)                 //
////////////////////////////////////////////////////////////////

//Interfaces and Channels for synchronization
////////////////////////////////////////////////

interface ISyncIn
    {
    void recv();
    };

interface ISyncOut
    {
    void send();
    };

channel CSync() implements ISyncIn, ISyncOut
    {
    bool    valid=false;
    event   e;

    void send()
        {
        valid=true;
        notify (e);
        }

    void recv()
        {
        if (!valid) wait(e);
        valid=false;
        }

    };

////////////////////////////////////////////////////////////////
// MEM_CHANNELS.sc                                            //
//                  ARCH2_EMI_MCC (July 2001)                 //
////////////////////////////////////////////////////////////////
```

```
interface PE_part
{
  void write_send(int data, int addr);       //Memory write
  void read_send(int*data, int addr);        //Memory read
};

interface MEM_part
{
  void recv_addr_write(int* addr, int* rw);
  void recv_data_write(int* data);
  void send_data_read(int data);
};


channel Mem_Access()
  implements PE_part, MEM_part
{
  bool  mem_valid=false;
  event send_addr, recv_addr, send_data, recv_data;
  int buf_mem;
  int add_mem;
  int rw_mem;


  void write_send(int data, int addr)
  {
    mem_valid=true;
    buf_mem = data;
    add_mem=addr;
    rw_mem=1;

    notify(send_addr);
    if (mem_valid) wait (recv_addr);

    mem_valid=true;
    notify(send_data);
    if (mem_valid) wait (recv_data);
  }

  void read_send(int* data, int addr)
  {
    mem_valid=true;
    add_mem=addr;
    rw_mem=0;

    notify(send_addr);
    if (mem_valid) wait (recv_addr);

    mem_valid=true;
    notify(send_data);
    if (mem_valid) wait (recv_data);

    *data=buf_mem;
  }

  void recv_addr_write(int* addr, int* rw)
  {
    if (!mem_valid) wait(send_addr);

    *addr=add_mem;
    *rw=rw_mem;

    notify(recv_addr);
    mem_valid=false;
  }

  void recv_data_write(int* data)
  {
    if (!mem_valid) wait(send_data);
    notify(recv_data);
    mem_valid=false;
    *data=buf_mem;
  }

  void send_data_read(int data)
  {
    if (!mem_valid) wait(send_data);
    notify(recv_data);
    mem_valid=false;
    buf_mem=data;
```



```
  }
};

////////////////////////////////////////////////////////////////
//  CLK.sc                                                    //
//                ARCH2_EMI_MCC (July 2001)                   //
////////////////////////////////////////////////////////////////


// SIMULATION CLOCK
//////////////////////

behavior Clock(out event clk, out bool stop)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while ((int)now()<sim_time1);

        stop=true;
        notify(clk);
        }
    };

////////////////////////////////////////////////////////////////
// MEMORY.sc                                                  //
//                ARCH2_EMI_MCC (July 2001)                   //
////////////////////////////////////////////////////////////////

behavior memory(MEM_part msync, in bool done)
    {
    int rw;
    int addr;
    int data;
    int mem[30000];

    void main(void)
        {
        do
            {
            msync.recv_addr_write(&addr, &rw);
            if (rw==1)
                {
                msync.recv_data_write(&data);
                mem[addr]=data;
                }
            if (rw==0)
                {
                data=mem[addr];
                msync.send_data_read(data);
                }
            }while(!done);

        }

    };


////////////////////////////////////////////////////////////////
// CTL.sc                                                     //
//                ARCH2_EMI_MCC (July 2001)                   //
////////////////////////////////////////////////////////////////

behavior Command_rap(in bool stop, in event clk, out bit[1:0] cd)
    {
    int i=0;
    int alph=180;

    void main(void)
        {
        do
            {
            wait clk;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
```

```
                if (i>284) i=0;
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (!stop);
        }
    };


////////////////////////////////////////////////////////////
// MEM_MASTER.sc                                          //
//                ARCH2_EMI_MCC (July 2001)               //
////////////////////////////////////////////////////////////

//INITIALISATION
//////////////////

behavior Init(PE_part ma_sync)
    {
    float   Alpha, Betta, Gamma, Londa, Mu, Psi;
    int     gv, a, b, l, m, n;

    float   Ve=60.0;

    float   rm=1.0;
    float   lm=0.00400;
    float   km=0.184;

    float   fm=0.000180;
    float   jm=0.00160;
    float   fc=0.000180;
    float   jc=0.0016;
    float   Cs=0.16;

    float   ft, jt;
    float   hor=1e-6;
    bool    init_f=true;

    int     param[6];

    void main(void)
        {
        if (init_f)
            {
            ft=fm+fc;
            jt=jm+jc;
            Alpha=1.-hor*rm/lm*(1.-hor/2.*rm/lm);
            Betta=-hor*km/lm*(1.-hor/2.*rm/lm);
            Gamma=hor/lm*(1.-hor/2.*rm/lm);
            Londa=hor*km/jt*(1.-hor/2.*ft/jt);
            Mu=1.-hor*ft/jt*(1.-hor/2.*ft/jt);
            Psi=-Cs*hor/jt*(1.-hor/2.*ft/jt);


            gv=(int)(1024*128*Gamma*Ve);
            a=(int)(1024*1024*(Alpha-1));
            b=(int)(1024*1024*Betta);
            l=(int)(1024*1024*Londa);
            m=(int)(1024*1024*(Mu-1));
            n=(int)(1024*128*Psi);

            ma_sync.write_send(gv, 0);
            ma_sync.write_send(a, 1);
            ma_sync.write_send(b, 2);
            ma_sync.write_send(l, 3);
            ma_sync.write_send(m, 4);
            ma_sync.write_send(n, 5);

            pTrace=fopen("R_arch2_emi_mcc.dat","w");

            init_f=false;
            }

        }
    };


behavior Storage(PE_part ma_sync, out bool done)
    {
    int     addr=6;
```



```
    float   im, vit;
    int     i_int, w_int;
    int     i;

    void main(void)
        {
        done=false;

        for(i=0; i<500; i++)
            {
            ma_sync.read_send(&i_int, addr);
            addr=addr+1;

            if (i==499) done=true;

            ma_sync.read_send(&w_int, addr);
            addr=addr+1;

            im=i_int/1024.;
            vit=w_int/1024.;
            fprintf(pTrace," %f  %f   \n",  im, vit);
            }

        fclose(pTrace);
        }
    };


//Master behavior
//////////////////

behavior Master(ISyncOut sync_1, ISyncIn sync_2, PE_part ma_sync, out bool done)
    {
    event       clk_s;

    Init        Init1(ma_sync);
    Storage     Storage1(ma_sync, done);

    void main(void)
        {
        Init1.main();
        sync_1.send();

        sync_2.recv();
        Storage1.main();
        }
    };


////////////////////////////////////////////////////////////
// MEM_EMUL.sc                                            //
//                ARCH2_EMI_MCC (July 2001)               //
////////////////////////////////////////////////////////////

#include    "motor.sc"
#include    "sensors.sc"

behavior Em_init(PE_part em_sync, out int gv,out int a, out int b, out int l, out int m, out int n)
    {
    void main(void)
        {
        em_sync.read_send(&gv, 0);
        em_sync.read_send(&a, 1);
        em_sync.read_send(&b, 2);
        em_sync.read_send(&l, 3);
        em_sync.read_send(&m, 4);
        em_sync.read_send(&n, 5);
        }
    };


// Storage_send behavior

behavior Storage_send(in int i_int, in int w_int, PE_part em_sync)
    {
    int stor_i=0;
    int addr=6;

    void main(void)
        {
```

```
        stor_i=stor_i+1;
        if (stor_i==CYCLESAVE)
            {
            em_sync.write_send(i_int, addr);
            addr=addr+1;
            em_sync.write_send(w_int, addr);
            addr=addr+1;
            stor_i=0;
            }
        }
    };

// Electromechanical system global module

behavior Emulator(ISyncIn sync_1, ISyncOut sync_2, PE_part em_sync,
                  in bool stop, in event clk, in bit[1:0] cd, out int Nim, out bit[1:0] S)
    {
    int i_int, w_int;
    int gv, a, b, l, m, n;

    Em_init         Em_init1(em_sync, gv, a, b, l, m, n);
    Motor           Motor1(gv, a, b, l, m, cd, i_int, w_int);
    Sen_cur         Sen_cur1(i_int, Nim);
    Sen_speed       Sen_speed1(w_int, S);
    Storage_send    Storage_send1(i_int, w_int, em_sync);

    void main(void)
        {
        i_int= w_int= 0;

        sync_1.recv();
        Em_init1.main();

        do
            {
            wait clk;
            Motor1.main();
            Sen_cur1.main();
            Sen_speed1.main();
            Storage_send1.main();
            }while (!stop);

        sync_2.send();

        printf("im=%f \n", i_int/1024.);
        printf("vit=%f \n", w_int/1024.);
        }
    };


////////////////////////////////////////////////////////////
// MOTOR.sc                                               //
//                  ARCH2_EMI_MCC (July 2001)             //
////////////////////////////////////////////////////////////

//ELECTROMECHANICAL SYSTEM
////////////////////////////

behavior Converter(in int gv, in bit[1:0] cd, out int Vout)
    {
    int a,b;
    void main(void)
        {
        a=cd[0];
        b=cd[1];
        Vout=gv*(a-b);
        }
    };

behavior Electric(in int a,in int b,in int Vout, in int w_int, inout int i_int)
    {
    int di_int=0;

    void main(void)
        {
        di_int=(((a*(i_int>>3))>>10)+((b*(w_int>>3))>>10)+Vout);
        i_int=i_int+di_int/128;
        }
    };
```



```
behavior Mecanic(in int l,in int m,in int i_int, inout int w_int)
    {
    int dw_int=0;
    int dw_int1=0;

    void main(void)
        {
        dw_int=(l*i_int+m*w_int)+dw_int;
        dw_int1=dw_int>>20;
        dw_int=dw_int-(dw_int1<<20);
        w_int=w_int+dw_int1;
        }
    };


behavior Motor(in int gv,in int a,in int b,in int l,in int m, in bit[1:0] cd,
               inout int i_int, inout int w_int)
    {
    int Vout;

    Converter Converter1(gv, cd, Vout);
    Electric Electric1(a, b, Vout, w_int, i_int);
    Mecanic Mecanic1(l, m, i_int, w_int);

    void main(void)
        {
        Converter1.main();
        Electric1.main();
        Mecanic1.main();
        }
    };

////////////////////////////////////////////////////////////
// SENSORS.sc                                             //
//                  ARCH2_EMI_MCC (July 2001)             //
////////////////////////////////////////////////////////////

// SENSORS MODULES
//////////////////

// Current sensor Generator

behavior Sen_cur(in int i_int, out int Nim)
    {
    int sens1_i=0;
    void main(void)
        {
        sens1_i=sens1_i+1;
        if (sens1_i==CYCLESEN1)
            {
            Nim=(int)(((p/1024)*i_int+p*imax)/(2*imax));
            sens1_i=0;
            }
        }
    };


behavior Sen_speed(in int w_int, out bit[1:0] S)
    {
    int sens2_i=0;
    int i=0;
    int w_inti;
    int T4_vit=24000;
    bool s_vit=true;

    void main(void)
        {
        sens2_i=sens2_i+1;

        if (sens2_i==CYCLESEN2)
            {
            w_inti=w_int;
            if (w_inti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(w_inti)<512) w_inti=512;          //Limite basse de vitesse a reproduire

            T4_vit=(int)((pi*1e6*1024)/(n_cio*w_inti));     //quart de periode en us

            sens2_i=0;
```

```
            }

        if (i <= T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            if (T4_vit < i && i <= 2*T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (2*T4_vit < i && i  <= 3*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (3*T4_vit < i && i  <= 4*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            i=i+1;
            if (i>4*T4_vit) i=0;
        }
    };
```

```
////////////////////////////////////////////////////////////////
// COM2_EMI_MCC.sc                                            //
//                                                            //
//                    COMMUNICATION MODEL                     //
//                                                            //
//                EMULATOR DESIGN _ INTEGER                   //
//          AUTONOMOUS STRUCTURE (INCLUDING MEMORY)           //
//                                                            //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS          //
//                                                            //
// Slim Ben Saoud                                             //
// CECS-UCI July 2001                                         //
//                                                            //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <sim.sh>
#include <math.h>
#include <stdlib.h>
#include <memory.h>

#include "typedef.sh"
#include "clk.sc"
#include "ctl.sc"
#include "memory.sc"
#include "master.sc"
#include "emul.sc"


//Testbench//////

behavior Main
    {
    int A;
    int D;
    event MCS;
    bool nRD=true;
    bool nWR=true;

    event       intC, intC1;

    event       SysClk;
    bool        stop;
    bit[1:0]    sc;
    int         Nim;
    bit[1:0]    S;
    bool        done;

    Clock       Clock1(SysClk, stop);
    Master      Master1(intC, intC1, A, D, MCS, nRD, nWR, done);
    Command_rap Command_rap1(stop, SysClk, sc);
    Emulator    Emulator1(intC, intC1, A, D, MCS, nRD, nWR, stop, SysClk, sc, Nim, S);
    memory      memory1(A, D, MCS, nRD, nWR, done);

    int main(void)
        {
        stop=false;
        done=false;

        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);

        puts("starting...");
        printf("Simulation time=%d us \n",sim_time1);

        sim_time1=sim_time1*1000;       //new cycletime is given in ns//

        par {
            Clock1.main();
            Master1.main();
            Command_rap1.main();
            Emulator1.main();
            memory1.main();
            }


        puts("Exiting...");

        return(0);
        }
```



```
    };

//EOF

////////////////////////////////////////////////////////////////
// TYPEDEF.sh                                                 //
//                COM2_EMI_MCC (July 2001)                    //
////////////////////////////////////////////////////////////////


// Time references

#define CYCLETIME   1000
#define CYCLENB     50
#define MAXSIMTIME  5000000
#define CYCLESAVE   10
#define ALPHA       200

// Electromechanical system parameters

#define CYCLESEN1   5           // cycle de rafraichissement du courant
#define CYCLESEN2   1000        // cycle de rafraichissement de la vitesse
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30


FILE    *pTrace;

// Simulation time

int sim_time1=5000;


////////////////////////////////////////////////////////////////
//  CLK.sc                                                    //
//                COM2_EMI_MCC (July 2001)                    //
////////////////////////////////////////////////////////////////

// SIMULATION CLOCK
//////////////////////

behavior Clock(out event clk, out bool stop)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while ((int)now()<sim_time1);

        stop=true;
        notify(clk);
        }
    };

////////////////////////////////////////////////////////////////
// CTL.sc                                                     //
//                COM2_EMI_MCC (July 2001)                    //
////////////////////////////////////////////////////////////////


behavior Command_rap(in bool stop, in event clk, out bit[1:0] cd)
    {
    int i=0;
    int alph=180;

    void main(void)
        {
        do
            {
            wait clk;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i>284) i=0;
```

```
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (!stop);
        }
    };

////////////////////////////////////////////////////////////
// MEMORY.sc                                              //
//                COM2_EMI_MCC (July 2001)                //
////////////////////////////////////////////////////////////


interface IProtocolMEM
    {
    void Mem_answer(int MEMOIRE[2000]);
    };

channel MEM_Protocol(int A, int D, event MCS, bool nRD, bool nWR)
    implements IProtocolMEM
    {
    void Mem_answer(int MEMOIRE[2000])
        {
        int addr;
        t1: wait(MCS);
        addr=A;

        while (nWR==1 && nRD==1)
            {
            waitfor(1);
            }

        if (nRD==0)
            {
            waitfor(6);
            D=MEMOIRE[addr];
            while (nRD==0)
                {
                waitfor(1);
                }
            }

        if (nWR==0)
            {
            waitfor(9);
            while (nWR==0)
                {
                waitfor(1);
                }

            MEMOIRE[addr]=D;
            }
         }
    };

behavior memory(int A, int D, event MCS, bool nRD, bool nWR, in bool done)
    {
    int MEMOIRE[2000];
    MEM_Protocol    mem_p(A, D, MCS, nRD, nWR);

    void main(void)
        {
        do
            {
            mem_p.Mem_answer(MEMOIRE);
            }while(!done);

        }

    };


////////////////////////////////////////////////////////////
// MASTER.sc                                              //
//                COM2_EMI_MCC (July 2001)                //
////////////////////////////////////////////////////////////

//PROTOCOL CHANNEL
//////////////////
```



```
interface IProtMaster
    {
    void    PE_read(int* data, int addr);
    void    PE_write(int data, int addr);
    };

/*------------------ I/O instructions of DSP56600 (MOVEM) ---------------- */

channel Master_Prot(int A, int D, event MCS, bool nRD,  bool nWR)
  implements IProtMaster
    {
/*----------MASTER-----------*/
  void PE_read(int* data, int addr)
  {
    waitfor(5);
    A = addr;
    notify(MCS);
    waitfor(5);
    nRD = 0;
    waitfor(16);
    *data = D;
    waitfor(3);
    nRD = 1;
    waitfor(4.2);
  }

  void PE_write(int data, int addr)
  {
    waitfor(5);
    A = addr;
    notify(MCS);
    waitfor(5);
    nWR = 0;
    waitfor(10);
    D = data;
    waitfor(8);
    nWR = 1;
    waitfor(5);
  }

};

//INITIALISATION
////////////////

behavior Init(IProtMaster Master_p)
    {
    float   Alpha, Betta, Gamma, Londa, Mu, Psi;
    int     gv, a, b, l, m, n;

    float   Ve=60.0;

    float   rm=1.0;
    float   lm=0.00400;
    float   km=0.184;

    float   fm=0.000180;
    float   jm=0.00160;
    float   fc=0.000180;
    float   jc=0.0016;
    float   Cs=0.16;

    float   ft, jt;
    float   hor=1e-6;
    bool    init_f=true;

    int     param[6];

    void main(void)
        {
        if (init_f)
            {
            ft=fm+fc;
            jt=jm+jc;
            Alpha=1.-hor*rm/lm*(1.-hor/2.*rm/lm);
            Betta=-hor*km/lm*(1.-hor/2.*rm/lm);
            Gamma=hor/lm*(1.-hor/2.*rm/lm);
            Londa=hor*km/jt*(1.-hor/2.*ft/jt);
            Mu=1.-hor*ft/jt*(1.-hor/2.*ft/jt);
            Psi=-Cs*hor/jt*(1.-hor/2.*ft/jt);
```

```
            gv=(int)(1024*128*Gamma*Ve);
            a=(int)(1024*1024*(Alpha-1));
            b=(int)(1024*1024*Betta);
            l=(int)(1024*1024*Londa);
            m=(int)(1024*1024*(Mu-1));
            n=(int)(1024*128*Psi);

            Master_p.PE_write(gv, 0);
            Master_p.PE_write(a, 1);
            Master_p.PE_write(b, 2);
            Master_p.PE_write(l, 3);
            Master_p.PE_write(m, 4);
            Master_p.PE_write(n, 5);

            pTrace=fopen("R_com2_emi_mcc.dat","w");

            init_f=false;
            }

        }
    };


behavior Storage(IProtMaster Master_p, out bool done)
    {
    int     addr=6;
    float   im, vit;
    int     i_int, w_int;
    int     i;

    void main(void)
        {
        done=false;

        for(i=0; i<500; i++)
            {
            Master_p.PE_read(&i_int, addr);
            addr=addr+1;

            if (i==499) done=true;

            Master_p.PE_read(&w_int, addr);
            addr=addr+1;

            im=i_int/1024.;
            vit=w_int/1024.;
            fprintf(pTrace," %f  %f   \n",  im, vit);
            }

        fclose(pTrace);
        }
    };


//Master behavior
//////////////////

behavior Master(event intC, event intC1, int A, int D, event MCS, bool nRD, bool nWR, out bool done)
    {
    event       clk_s;

    Master_Prot Master_p(A, D, MCS, nRD, nWR);

    Init        Init1(Master_p);
    Storage     Storage1(Master_p, done);

    void main(void)
        {
        Init1.main();
        notify(intC);

        wait(intC1);
        Storage1.main();
        }
    };


////////////////////////////////////////////////////////////////
```



```
// EMUL.sc
//                  COM2_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////////

#include    "motor.sc"
#include    "sensors.sc"

//PROTOCOL CHANNEL
//////////////////////

interface IProtEmul
    {
    void    PE_read(int* data, int addr);
    void    PE_write(int data, int addr);
    };

/*------------------- I/O instructions of DSP56600 (MOVEM) ---------------- */

channel Emul_Prot(int   A, int D, event MCS, bool nRD,  bool nWR)
  implements IProtEmul
    {
/*----------MASTER-----------*/
  void PE_read(int* data, int addr)
  {
    waitfor(5);
    A = addr;
    notify(MCS);
    waitfor(5);
    nRD = 0;
    waitfor(16);
    *data = D;
    waitfor(3);
    nRD = 1;
    waitfor(4.2);
  }

  void PE_write(int data, int addr)
  {
    waitfor(5);
    A = addr;
    notify(MCS);
    waitfor(5);
    nWR = 0;
    waitfor(10);
    D = data;
    waitfor(8);
    nWR = 1;
    waitfor(5);
  }

};

behavior Em_init(IProtEmul Emul_p, out int gv,out int a, out int b, out int l, out int m, out int n)
    {
    void main(void)
        {
        Emul_p.PE_read(&gv, 0);
        Emul_p.PE_read(&a, 1);
        Emul_p.PE_read(&b, 2);
        Emul_p.PE_read(&l, 3);
        Emul_p.PE_read(&m, 4);
        Emul_p.PE_read(&n, 5);
        }
    };


// Storage_send behavior

behavior Storage_send(in int i_int, in int w_int, IProtEmul Emul_p)
    {
    int stor_i=0;
    int addr=6;

    void main(void)
        {
        stor_i=stor_i+1;
        if (stor_i==CYCLESAVE)
            {
            Emul_p.PE_write(i_int, addr);
            addr=addr+1;
```

```
            Emul_p.PE_write(w_int, addr);
            addr=addr+1;
            stor_i=0;
            }
        }
    };

// Electromechanical system global module

behavior Emulator(event intC, event intC1, int A, int D, event MCS, bool nRD, bool nWR,
                  in bool stop, in event clk, in bit[1:0] cd, out int Nim, out bit[1:0] S)
    {
    int i_int, w_int;
    int gv, a, b, l, m, n;

    Emul_Prot       Emul_p(A, D, MCS, nRD, nWR);

    Em_init         Em_init1(Emul_p, gv, a, b, l, m, n);
    Motor           Motor1(gv, a, b, l, m, cd, i_int, w_int);
    Sen_cur         Sen_cur1(i_int, Nim);
    Sen_speed       Sen_speed1(w_int, S);
    Storage_send    Storage_send1(i_int, w_int, Emul_p);

    void main(void)
        {
        i_int= w_int= 0;

        wait(intC);
        Em_init1.main();

        do
            {
            wait clk;
            Motor1.main();
            Sen_cur1.main();
            Sen_speed1.main();
            Storage_send1.main();
            }while (!stop);

        notify(intC1);

        printf("im=%f \n", i_int/1024.);
        printf("vit=%f \n", w_int/1024.);
        }
    };

////////////////////////////////////////////////////////////////
// MOTOR.sc                                                   //
//                  COM2_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////////

//ELECTROMECHANICAL SYSTEM
/////////////////////////

behavior Converter(in int gv, in bit[1:0] cd, out int Vout)
    {
    int a,b;
    void main(void)
        {
        a=cd[0];
        b=cd[1];
        Vout=gv*(a-b);
        }
    };

behavior Electric(in int a,in int b,in int Vout, in int w_int, inout int i_int)
    {
    int di_int=0;

    void main(void)
        {
        di_int=(((a*(i_int>>3))>>10)+((b*(w_int>>3))>>10)+Vout);
        i_int=i_int+di_int/128;
        }
    };

behavior Mecanic(in int l,in int m,in int i_int, inout int w_int)
    {
    int dw_int=0;
    int dw_int1=0;
```



```
    void main(void)
        {
        dw_int=(l*i_int+m*w_int)+dw_int;
        dw_int1=dw_int>>20;
        dw_int=dw_int-(dw_int1<<20);
        w_int=w_int+dw_int1;
        }
    };


behavior Motor(in int gv,in int a,in int b,in int l,in int m, in bit[1:0] cd,
               inout int i_int, inout int w_int)
    {
    int Vout;

    Converter Converter1(gv, cd, Vout);
    Electric Electric1(a, b, Vout, w_int, i_int);
    Mecanic Mecanic1(l, m, i_int, w_int);

    void main(void)
        {
        Converter1.main();
        Electric1.main();
        Mecanic1.main();
        }
    };

////////////////////////////////////////////////////////////////
// SENSORS.sc                                                 //
//                  COM2_EMI_MCC (July 2001)                  //
////////////////////////////////////////////////////////////////


// SENSORS MODULES
////////////////////


// Current sensor Generator

behavior Sen_cur(in int i_int, out int Nim)
    {
    int sens1_i=0;
    void main(void)
        {
        sens1_i=sens1_i+1;
        if (sens1_i==CYCLESEN1)
            {
            Nim=(int)(((p/1024)*i_int+p*imax)/(2*imax));
            sens1_i=0;
            }
        }
    };


behavior Sen_speed(in int w_int, out bit[1:0] S)
    {
    int sens2_i=0;
    int i=0;
    int w_inti;
    int T4_vit=24000;
    bool s_vit=true;

    void main(void)
        {
        sens2_i=sens2_i+1;

        if (sens2_i==CYCLESEN2)
            {
            w_inti=w_int;
            if (w_inti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(w_inti)<512) w_inti=512;          //Limite basse de vitesse a reproduire

            T4_vit=(int)((pi*1e6*1024)/(n_cio*w_inti));     //quart de periode en us

            sens2_i=0;
            }
```

```
        if (i <= T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            if (T4_vit < i && i <= 2*T4_vit)
                {
                S[0]=1;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (2*T4_vit < i && i  <= 3*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=1;
                else S[1]=0;
                }

            if (3*T4_vit < i && i  <= 4*T4_vit)
                {
                S[0]=0;
                if (s_vit) S[1]=0;
                else S[1]=1;
                }

            i=i+1;
            if (i>4*T4_vit) i=0;
        }
    };
```