SpecC Methodology applied to the Design of Control systems for Power Electronics and Electric Drives

Slim Ben Saoud, Daniel D. Gajski

Technical Report ICS-01-45
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

SpecC Methodology applied to the Design of Control systems for Power Electronics and Electric Drives

Slim Ben Saoud, Daniel D. Gajski

Technical Report ICS-01-45
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

Today, control algorithms are being more and more sophisticated due to the customer and governments demands for lower cost, greater reliability, greater accuracy and environment requirements (power consumption, emitted radiation,...). Then, real-time implementation of these algorithms becomes a difficult task and needs more and more specific hardware systems with dedicated processors and usually systems-on-chip (SOCs).
With the ever-increasing complexity and time-to-market pressures in the design of these specific control systems, a well design methodology is more than even necessary.

In this report we describe the application of the SpecC system-level design methodology (developed at the CAD Lab, UC Irvine) to the design of control systems for power electronics and electric drives. We first begin with an executable specification model in SpecC and then discuss the refinement of this model into architecture model, which accurately reflects the system architecture. At this stage, we discuss different solutions according to the application complexity and constraints. Based on the studied architecture models, communication protocols between the system components are defined and communication models are developed.
In this report, we discuss the case of a DC system Control and describe in details different stages undergone. Generalization to others systems can be done easily using the same steps and transformations.

Contents

List of Figures

SpecC Methodology applied to the Design of Control systems for Power Electronics and Electric Drives

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

ABSTRACT

Today, control algorithms are being more and more sophisticated due to the customer and governments demands for lower cost, greater reliability, greater accuracy and environment requirements (power consumption, emitted radiation,...). Then, real-time implementation of these algorithms becomes a difficult task and needs more and more specific hardware systems with dedicated processors and usually systems-on-chip (SOCs).
With the ever-increasing complexity and time-to-market pressures in the design of these specific control systems, a well design methodology is more than even necessary.

In this report we describe the application of the SpecC system-level design methodology (developed at the CAD Lab, UC Irvine) to the design of control systems for power electronics and electric drives. We first begin with an executable specification model in SpecC and then discuss the refinement of this model into architecture model, which accurately reflects the system architecture. At this stage, we discuss different solutions according to the application complexity and constraints. Based on the studied architecture models, communication protocols between the system components are defined and communication models are developed.
In this report, we discuss the case of a DC system Control and describe in details different stages undergone. Generalization to others systems can be done easily using the same steps and transformations.

1 Introduction

The goal of this work is to introduce the SpecC methodology to the design of electric drives. In this project, we present and discuss the case of a DC motor control. The control algorithm used is very simple and can be implemented using standard micro-controller. So, the objective of this work is not really to design the control device but to introduce the SpecC methodology and to discuss its application to the electric drive controller design. A generalization of this study to any other system control can be done easily using the same steps discussed in the following sections.

The control device was described in four models, which represent four different levels of abstraction in the SpecC methodology [1,2]. All these models are executable and validated by simulation.

The rest of the report is organized as follows: We first begin with a brief presentation of the SpecC Methodology. Then we describe an executable specification model (in SpecC) of the control system and we discuss the refinement of this model into architecture model, which accurately reflects the system architecture. At this stage, we discuss different solutions according to the application complexity and constraints. Based on the studied architecture models, communication protocols between the system components are defined and communication models are developed.

2 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 1). 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 1. 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.

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.

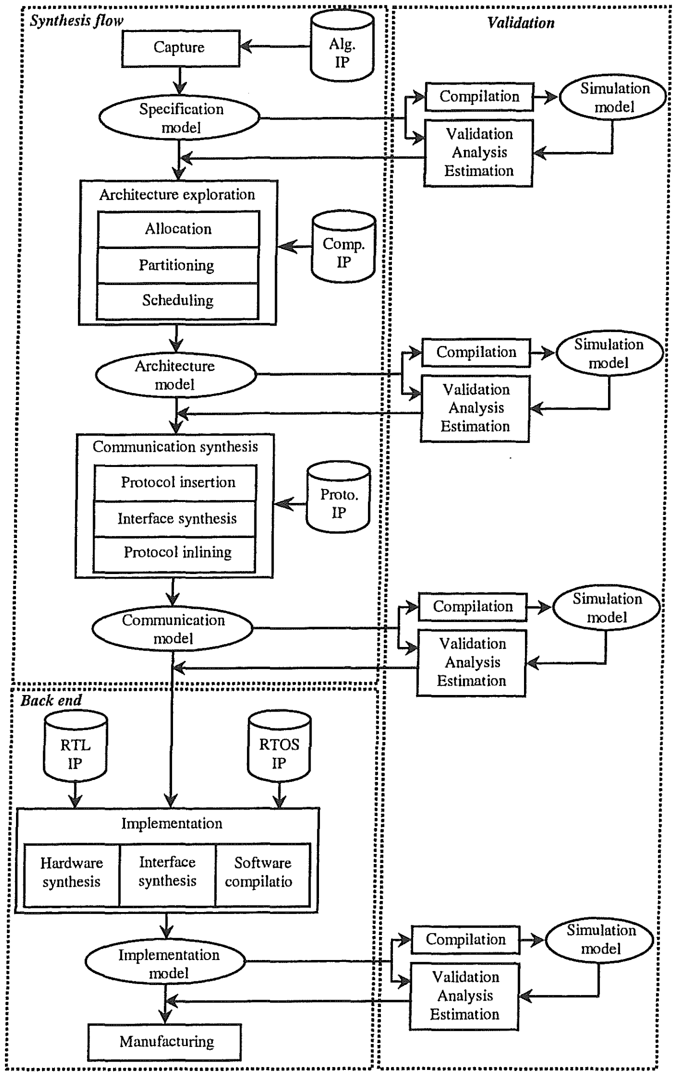


Figure 1: SpecC methodology

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 1. 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.

3 Specification

The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to architecture exploration, the first step of the system design process, and therefore defines the basis for all synthesis and exploration. For example, the specification model defines the granularity for exploration through the size of the leaf behaviors, it exposes the available parallelism, uses hierarchy to group related functionality and manage complexity, separates communication from computation, and so on.
The specification model is a purely functional, abstract model that is free of any implementation details. The hierarchy of behaviors in the specification model solely reflects the system functionality without implying anything about the system architecture to be implemented. For example, parallel parts in the specification model describes independent groups of functions that can run concurrently but does not make any premature assumptions about an implementation on concurrent processing elements.
The specification model is free of any notion of time. The model executes in zero logical (simulation) time. Events in the specification model are used by the designer for synchronization only in order to specify causality and thus establish a partial ordering among these behaviors [2].

3.1 Control Device Specification

The DC control device has been specified using SpecC language in previous work. The control algorithm and the I/O modules composing the control device was represented in a formal, executable, specification model that has been validated by simulation. This obtained model is shown on figure 2.

The used control algorithm is composed of two control loops: an outer motion loop and an inner current loop. Each of them is specified in a separate sub-behavior and associated to a clock-behavior that generated the synchronization event to activate the corresponding control loop at the predefined periodic step. For the current control loop, we used a period of 284µs and for the motion one, we used a period of 20ms. However, in the specification model, there's no notion of time, so in all our specification model, we consider that basic cycle of time is 1µs and then we used for the Clk_i behavior the *waitfor* statements: a *waitfor(284)* for the Clk_i behavior and a *waitfor(20000)* for the Clk_Ω behavior.

The I/O modules necessary for the control device functioning are specified in two behaviors: the *PWM* behavior represents the PWM module functioning while the *ACQ* behavior represents the information acquisition modules. Each of these modules is specified by a specific sub-behavior associated to a clock-behavior that generates the synchronization event necessary for its activation. These I/O modules are independent and they usually use different clock periods.

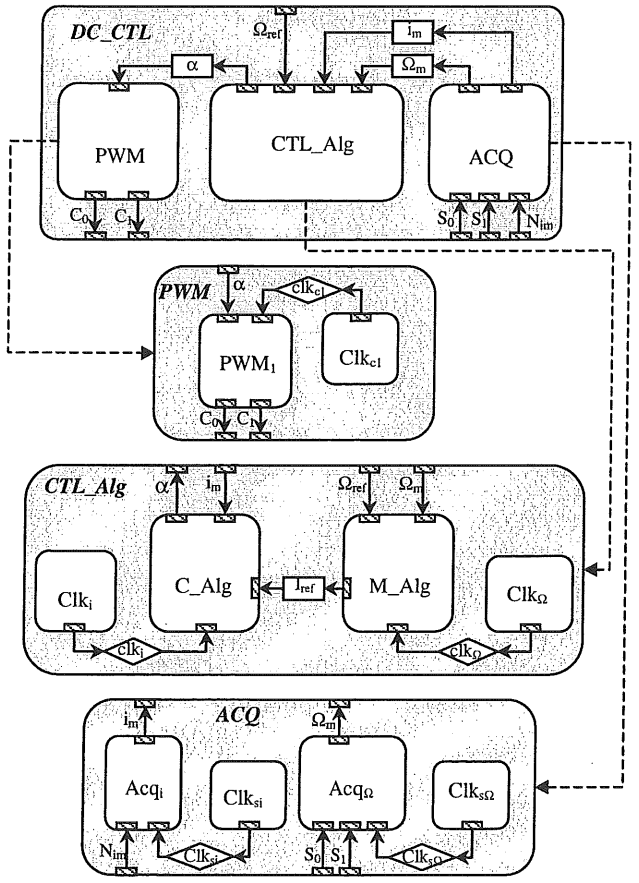


Figure 2: Specification model of the control device

The PWM behavior generates two complementary signals C0 and C1 with the same frequency as the current control module clock and according to the pulse width value α (for C0) obtained by the current control behavior.
The current acquisition behavior captures the current value (N_{im} obtained from the used ADC component) and computes its average value over the current control period (i_m). While the speed acquisition behavior computes the speed value (Ω_m) from the two signals S0 and S1 generated by the optical incremental encoder (sensor used on the process under control).

As shown on figure 2, the SpecC specification describes the control device functionality in a clear and precise manner.

3.2 Control Device Constraints

Usually, the main constraints of control devices are:

- Time: especially the time execution of the control algorithm and the time needed for conversion of analog information to digital form.
- Precision: especially the resolution of the I/O modules like the resolution of the used ADC components (number of bits), the resolution of position/speed acquisition module (clock and number of bits) and the resolution of the PWM module... These characteristics are usually dependent on the used processor data bus.

In our application, and since we use a simple control algorithm, these constraints are not really severe. As an example, we used the following specifications:

- a period Tc=284 µs for the current control loop;
- a period of Tm=20ms for the speed control loop;
- a resolution of 10 bits for the ADC component, and a period of 5 µs for the current acquisition;
- a period of 1ms for the speed acquisition;
- a clock of 1µs for the PWM module ....

These values are only used as an example and then will be adapted by the user according to his application.

4 Architecture Exploration

Architecture exploration is the first part of the system synthesis process that develops a system architecture from the specification model. The purpose of architecture exploration is to map the computational parts of the specification, represented by the behaviors in the specification model, onto the components of a system architecture. The steps involved in this process are:

allocation of a set of system components (Processing elements PEs and memories), partitioning of behaviors onto the processing elements, mapping of variables into memories and scheduling of behaviors on the inherently sequential PEs. Through this process, the specification model is gradually refined into the architecture model.

Note that in general, exploration is an iterative process. The different tasks can be executed repeatedly and in each iteration the task can be done generally in any order or even simultaneously.
In order to perform architecture exploration, it is crucial to obtain accurate information about the design in a short amount of time. Therefore, the task of estimation is central to the whole exploration process. Estimation tools determine design metrics such as performance (execution time) and memory requirements (code and data size) for each part of the specification with respect to the allocated components.

Usually, to get the better trade-off between the performance and cost, HW/SW partition is performed, which involves the estimation of the different partitions. Based on the estimation the partition of the system can be done [3].

Knowing the HW/SW performance of each block, we could consider different partitioning solutions. For each partition, we could compute the number of clock cycles required for the HW block(s), the number of clock cycles required for the SW block(s) and hence the total number of clock cycles.

Naturally, the more the functionality was put into HW, the less was the required number of clock cycles. However each partition was associated with a communication overhead in terms of the amount of data transferred at the interface. Based on the communication overhead, certains decisions regarding local HW memory and shared memory will be made.

In our application, the motion control loop does not present severe temporal constraints and it is usually used as the main program in which, we integrate the communication with the user for configuration and monitoring. So this block is usually preferred as a software one. On the other hand the main part of the I/O modules require time management (timers) so they are implemented on hardware.
Therefore, the study concerns in the most of cases the current control loop because it presents the most severe temporal constraint.

However, in this study, our objective is to introduce the SpecC methodology to the case of control device design. So we use a simple application for which constraints are not severe. We present two architectures models (according to the current control implementation), that seem to be the most useful for our type of application. In the following, we will show the step-by-step process applied to the specification model developed in previous section, in order to obtain two different architecture models.

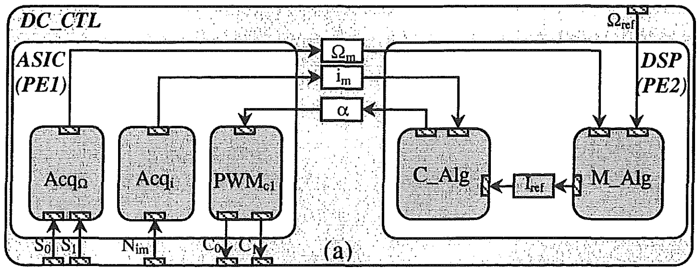
4.1 Allocation

The first task of the architectural exploration process is the allocation of a system target architecture consisting of a set of components and their connectivity. Allocation selects the number and types of processing elements (PEs), memories and busses in the architecture, and it defines the way PEs and memories are connected over the system busses. Components and protocols are taken out of a library and can range from full-custom designs to fixed IPs.
After an architecture has been allocated, the first step in implementing the specification on the given architecture is to map the SpecC behaviors onto the architecture's processing elements. In the refined model after behavior partitioning an additional level of hierarchy is inserted with top-level behaviors representing the components of the architecture.

For the control device application, usually the I/O modules are done by hardware modules (ADC, Timers, ...) while the control algorithm is implemented in a standard processor. However, sometimes, this solution is not adequate for sophisticated algorithm running in real time. Than, usually we remove a part of the control algorithm from the processor and we implemented it by hardware. This part is usually the current loop because it represents the most severe time constraints. In all these cases, the specification of the retained architecture, its validation and its design must be done using a methodology. In this work, we propose to develop these two architecture solutions using SpecC methodology.
Note that the I/O modules can be implemented on a common component or on different components.

According to the previous considerations, we distinguish two principles architectures that can be used for electric drives. These architectures will be studied here as an example. The first one (arch1) uses two components an ASIC for the I/O modules and a processor for the control algorithm. While the second one (arch2) uses a hardware component for each I/O module, an ASIC for the current control module and a processor for the speed control module and the interface with the user.

The obtained models are shown in figure 3. Note that in these architectures the clock generator behaviors used in the specification model are not considered as a part of the control device. These clock events are considered as inputs to the control unit…



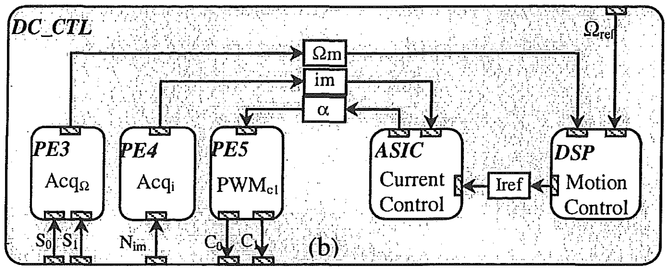


Figure 3: Architecture models after behavior partitioning (a-arch1 & b-arch2)

In Arch1, the processor core (DSP56600 core) running control algorithm is supported by a hardware component for the I/O functions. However, in arch2, only the motion control loop is implemented on the processor core while the current control loop is implemented on a custom coprocessor and each of the I/O function is implemented on a specific hardware.

Formerly local variables used for communication between behaviors mapped to different components now become global, system-level variables. Synchronization between behaviors mapped to different components is done by the clock behaviors integrated in the testbench specification as defined in the specification model. Other synchronization behaviors can be added if necessary in order to preserve the execution order as represented by the specification model…

In arch1, variables exchanged between ASIC and DSP are:

- Pulse width variable (α) computed by the current control loop module and sent to the PWM block at the beginning of each new current control period Tc;
- The current captured value (i_m: average value) determined by the current acquisition module and used by the current control module;
- The speed captured value (Ω_m) determined by the speed acquisition module and used by the motion control module.

The variable I_{ref} generates by the motion control module and used by the current control module is a local variable inside the DSP.

However, in arch2 we distinguish 5 PEs (PWM, ACQ$_i$, ACQ$_\Omega$, ASIC and DSP) and α, i_m, Ω_m variables are exchanged between respectively (ASIC-PWM), (ACQ$_i$-ASIC) and (ACQ$_\Omega$-DSP). In addition to these variables, I_{ref} become a global variable and it will be transmitted from DSP to ASIC.

In the following sections, these two architectures will be studied separately.

4.2 ARCH1

4.2.1 Variable Partitioning

After behavior partitioning, communication between behaviors mapped to different PEs is performed via global, shared variables. Global variables have to be assigned to local memory in the PEs or to a dedicated shared memory component. In the refined model after variable partitioning, global variables are replaced with abstract channels and code is inserted into the behaviors to communicate variable values over those channels.

In our application, the number of exchanged variables is very limited. So, our choice is to use local copies of these variables in each PEs. Then, the behaviors inside the PEs are connected to the corresponding local copy and operate on the data in local memory instead of accessing a global variable. Updated data values are communicated between ASIC and DSP through 3 abstract channels (C_α, C_{im} and $C_{\Omega m}$). Synchronizations are done by appointment at each current control period (284) for C_α and C_{im} and at each speed control period (20000) for $C_{\Omega m}$.

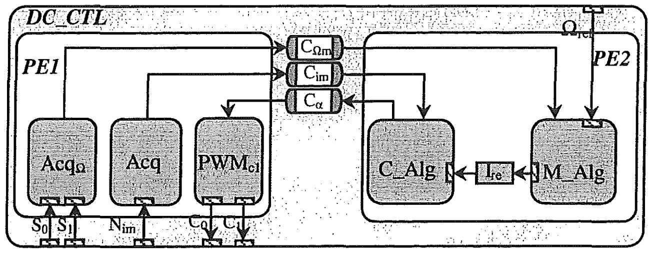


Figure 4: Architecture model after variable partitioning (ARCH1)

Partitioning is immediately follwed by the task of scheduling. Both are closely related since the quality of a partition is revealed only once scheduling has been performed.

4.2.2 Scheduling

Scheduling determines the execution order of behaviors that execute on inherently sequential PEs. Scheduling may be done statically or dynamically [4]. In static scheduling, each behavior is executed according to a fixed schedule. In the refined model after scheduling, behaviors inside each component are executed sequentially according to the computed schedule. Redundant synchronization between the behaviors is removed during optimization. In dynamic scheduling, the execution of behaviors on each component is determined at run-time. An application-specific run-time scheduler is created during refinement.
Figure 5 shows the scheduling of the parallel control algorithm running on the DSP core. Due to the dynamic timing relation between motion loop and current loop tasks, a dynamic scheduling scheme is implemented. The motion control represents the main program, which executes in periodic manner. Whenever a new current period arrives, the main task is interrupted in order to execute the current control.

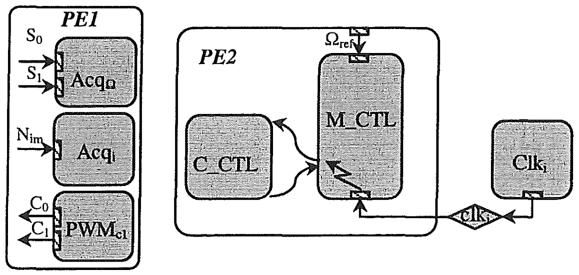


Figure 5: Architecture model after scheduling (ARCH1)

According to this scheduled model and in order to simplify synchronization for communication, we choice to do all exchanges at the beginning of each current control loop which means at each period Tc. The Ω_m value will be then a local variable of the DSP as well as Iref.
Exchanges synchronization can be done by an external clock (as represented on figure 5) or by an event generated by the ASIC and precisely by the PWM module (since it will integrate a temporization function at the period of Tc).

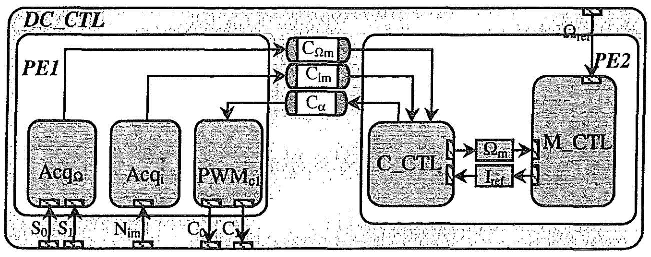


Figure 6: Modification of variable partitioning (ARCH1)

4.2.3 Channel Partitioning

Channel partitioning is the process of mapping and grouping the abstract, global communication channels between components onto the busses of the target architecture. In the refined model, additional top-level channels are used to represent system busses. Then channel partitioning is reflected by hierarchically grouping and encapsulating the abstract, global channels under the top-level bus channels.
Note that the bus is also a type of channel in SpecC, and it implies that the future implementation would be the wired buses. Channels connect the concurrent behaviors while buses connect the corresponding components into which these behaviors are mapped. Usually, only one bus is used between two components.

For this architecture, we used only one bus, which connects the processor to the custom hardware component. Therefore all communication are mapped to that bus. So, in the SpecC description of the refined control device, a single channel representing the system bus is inserted at the top level. The two components are connected to this bus channel and all abstract channels for communication between behaviors (C_α, C_{im} and $C_{\Omega m}$) are grouped under the top-level channel.

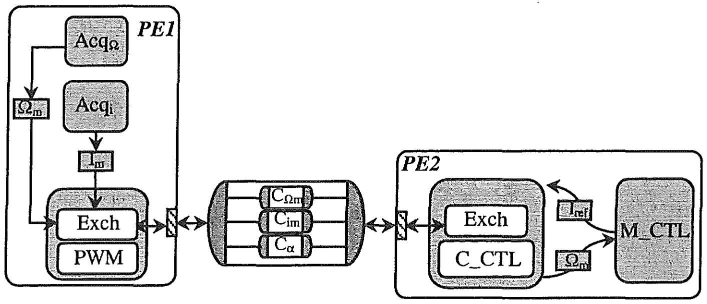


Figure 7: Architecture model after channel partitioning (ARCH1)

As indicated in the previous section, exchanges will be done at the beginning of each Tc period between the PWM module and the current control module.

4.3 ARCH2

4.3.1 Variable Partitioning

The number of exchanged variable is very limited. So local copies of global variables are added to the correspondent PEs. Updated data values are communicated between these PEs through 4 different abstract channels: $C_{\Omega m}$, C_{im}, C_α and C_{Iref}.

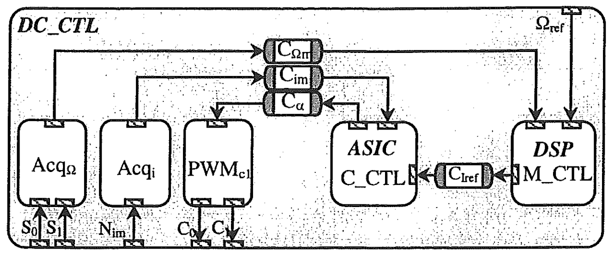


Figure 8: Architecture model after variable partitioning (ARCH2)

According to this architecture, the I/O modules are independent and their results are obtained in an asynchronous manner. Indeed, the speed acquisition module, for example, computes the period of a variable frequency signal, so the new result is obtained in an asynchronous manner (depends on the speed signal frequency which is variable).

In order to not charge theses components by the synchronization and the communication processes needed for transmission, we choice to add elementary memory blocks (one register) to these PEs (one block per PEs). These memories will be integrated in the I/O hardware module. So, obtained results are automatically loaded on these memories and transfer will be done between them and the control PEs (current PEs and motion PEs).
According to this architecture, no synchronization is needed between the I/O modules and the control modules since the exchange is done through memory blocks.
On the other hand, synchronization between the ASIC and the DSP is done by interruption. Indeed, at each Tc period, the ASIC interrupts the DSP and begins the transfer of Iref within the channel C_{Iref}.

4.3.2 Channel Partitioning

According to the previous specification, we distinguish at least two main possibilities of channel partitioning:

- As shown on figure 9, channels can be mapped onto two buses:
  - one bus between the ASIC and PE4/PE5: this bus will be managed by the ASIC;
  - one bus between the DSP and PE1/PE3: this bus will be managed by the DSP.

  Using this solution buses are managed easily since each bus will have only one master that initiate each transfer. However it presents a main inconvenient, in fact the number of pin in the coprocessor will be important since it used two different busses

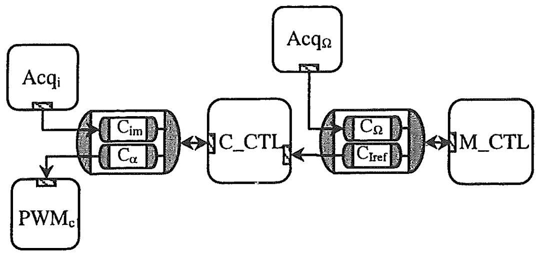


Figure 9: Solution 1 of Channel partitioning

- For that reason, we propose a second possibility on which only one bus is used as a common bus to all components as shown on figure 10. However, this solution includes a major difficulty of the bus management, in fact it will have two masters. So a management protocol should be added to resolve conflicts when these two masters tries to use the bus at the same time.

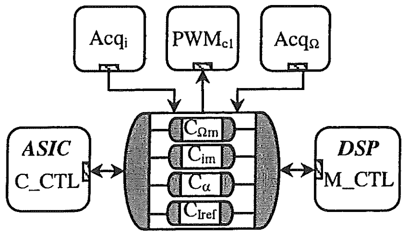


Figure 10: Solution 2 of Channel partitioning

In order to simplify the communication process, we choice to use another variables partition with only one bus and one master. So, we introduce some modification to our architecture model as shown on figure 11.

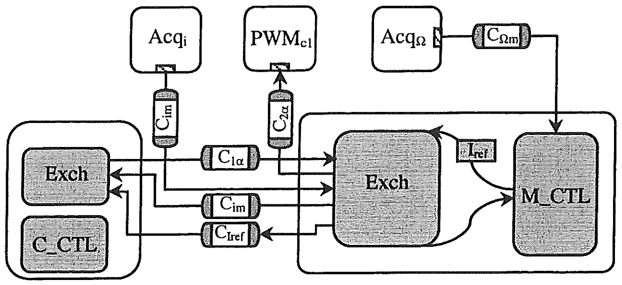


Figure 11: Modification of variable partitioning

Synchronization for the transfer is done by interruption. At the beginning of each new Tc period, the ASIC interrupts the DSP and both start the exchange process. Inside this process, The ASIC sends α and waits for i_m and i_{ref} while the DSP (the master) begins by reading the α value, then it writes this data to PE5, and reads i_m value from PE4 and finally it sends i_m and i_{ref} values to the ASIC. Acquisition of Ω_m value is done by the master at the beginning of each Tm period..

The final architecture model using one bus is represented by figure 12.

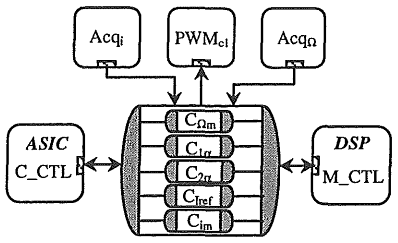


Figure 12: Architecture refined model (ARCH2)

5 Communication Synthesis

The purpose of communication synthesis is to refine the abstract communication in the architecture model into an actual implementation over the wires of the system busses. This requires insertion of communication protocols for the busses, synthesis of protocol transducers to translate between incompatible protocols, and inlining of protocols into hardware and software. These steps will be discussed in the following sections for the two retained architecture targets separately.

5.1 ARCH1

5.1.1 Protocol Insertion

During the protocol insertion, a description of the protocol is taken out of the protocol library in the form of a protocol channel and inserted into the corresponding virtual system bus channel (figure 13).
The protocol channel encapsulates the bus wires and implements the bus protocol by driving and sampling bus wires according to the protocol timing. At its interface, the protocol channel provides methods for all primitive transactions supported by the protocol, e.g. read, write, etc.
The abstract communication primitives provided of the bus channel are rewritten into an implementation using the primitives provided by the protocol layer. The outer application layer of the bus channel implements the required semantics over the actual bus protocol. This includes tasks like synchronization, arbitration, bus addressing, data slicing, and so on.
All the abstract bus channels in the model are replaced with their equivalent hierarchical combinations of protocol and application layers that implements the abstract communication of each bus over the actual protocol for that bus.

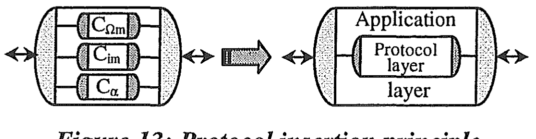


Figure 13: Protocol insertion principle

In this example, after protocol insertion, the processor is the central component and the master of the system bus. The software on the processor initiates all data transfers on the processor bus from and to the hardware component. However, these exchanges are initiated (provoked) either by an external clock (at Tc period) or by the hardware component that send an event at each Tc period to the processor by triggering its interrupt in order to execute the exchanges process (at the beginning of the current control loop).
At this stage, the processor as a master of the bus initiates and controls data transfers to and from the custom hardware. It initiates the transfer by reading from or writing to the memory location with the address of the HW component. The HW, 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.
So, at the beginning of each new Tc period, the custom hardware signals its ready state for communication by raising an interrupt. The corresponding interruption software on the processor begins transferring the data one word at a time by repeatedly executing instructions that initiate read or write cycles on the external bus: beginning by the send of α and then receive of i_m and Ω_m.

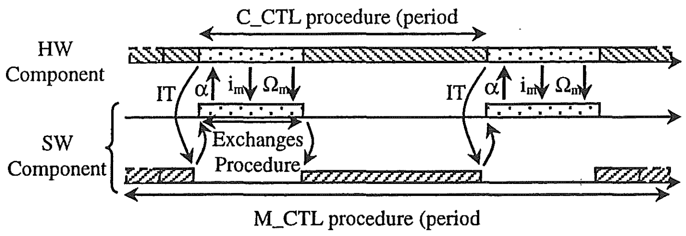
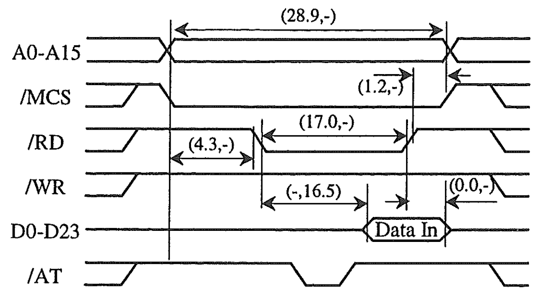


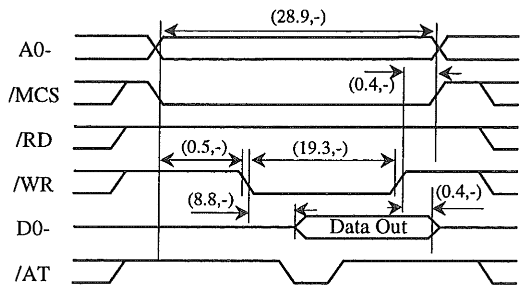
Figure 14: HW/SW Synchronization diagrams

Note that the clear separation between communication and computation enables replacement of a general component with an IP model plus wrapper and transducer at any stage of the design process. The wrapper specifies how to interface the IP model with the rest of the design. For simulation purposes, any model of the IP component that provides a suitable programming interface can be hooked into the SpecC simulator through the wrapper.

The protocol channel in the system bus and the wrapped processor model describe and implement the DSP56600 bus protocol according to its timing diagram [5], 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...



DSP56600 - SRAM Read Access



DSP56600 - SRAM Write Access

Figure 15: Protocols of the DSP56600 external bus

Figure 16 shows the system model after insertion of the DSP56600 bus protocol for the system bus. The bus protocol is modeled as a SpecC channel in the protocol library. The protocol channel is inserted into the top-level bus channel and all communication over the system bus is implemented using the primitives provided by the protocol.

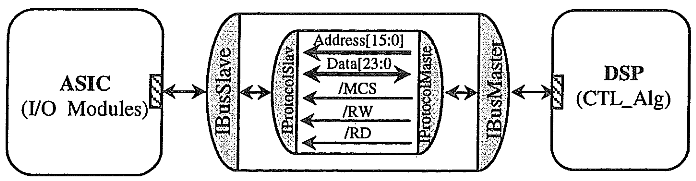


Figure 16: Communication model after protocol insertion (ARCH1)

5.1.2 Protocol Inlining

Protocol inlining is the process of inlining the channel functionality into the connected components and exposing the actual wires of the busses. The communication code is moved into the components where it is implemented in software or hardware. On the hardware side, FSMDs that implement the communication and bus protocol functionality are synthesized. On the software side, bus drivers and interrupt handlers that perform the communication using the processor's I/O instructions are generated or customized.

The communication model obtained after protocol inlining is shown in figure 17. In this case, all data transfers on the processor bus are initiated by the DSP. However the High-level handshaking and synchronization between hardware and software is realized using interrupt-based handshaking.

For the ASIC, communication primitives are inlined into the exchanges sub-behavior that have been created inside the PWM behavior, during partitioning, for synchronization and communication of the ASIC with the DSP. So, both application and potocol layers of the communication primitives that had been created during protocol insertion are inlined into the custom hardware behavior.

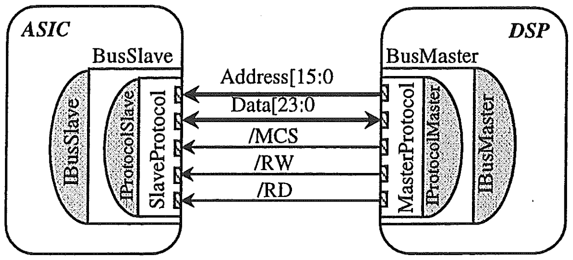


Figure 17: Communication model after protocol inlining (ARCH1)

During inlining, exchanges SFSMD model is created and inserted into the ASIC SFSMD model. They will later be synthesized into custom hardware together. Note that in general there are many different ways of implementing the transfer functionality, and a choice about the final hardware design has to be made at this point.

The exchanges hardware module synchronizes with the DSP by raising the processor's interrupt line IRQC in its first state S1 until a transfer with the address of the custom hardware is recognized. Then the WR control signal is sampled until a falling edge has been detected that signals the beginning of a bus write cycle. Communication continues at the same manner for two read cycles.

Note that the studied architecture is just a fictive discussed as an example to validate our approach. Usually, different hardware components are used for each I/O module, as we will see in next sections.

5.2 ARCH2

The DSP 56600 protocol is employed for ASIC and DSP while another simple memory protocol is used for memory blocks. We suppose that ASIC and DSP use the same protocol and that timing constraints are compatible between these two protocols. Otherwise we have to insert transducer.

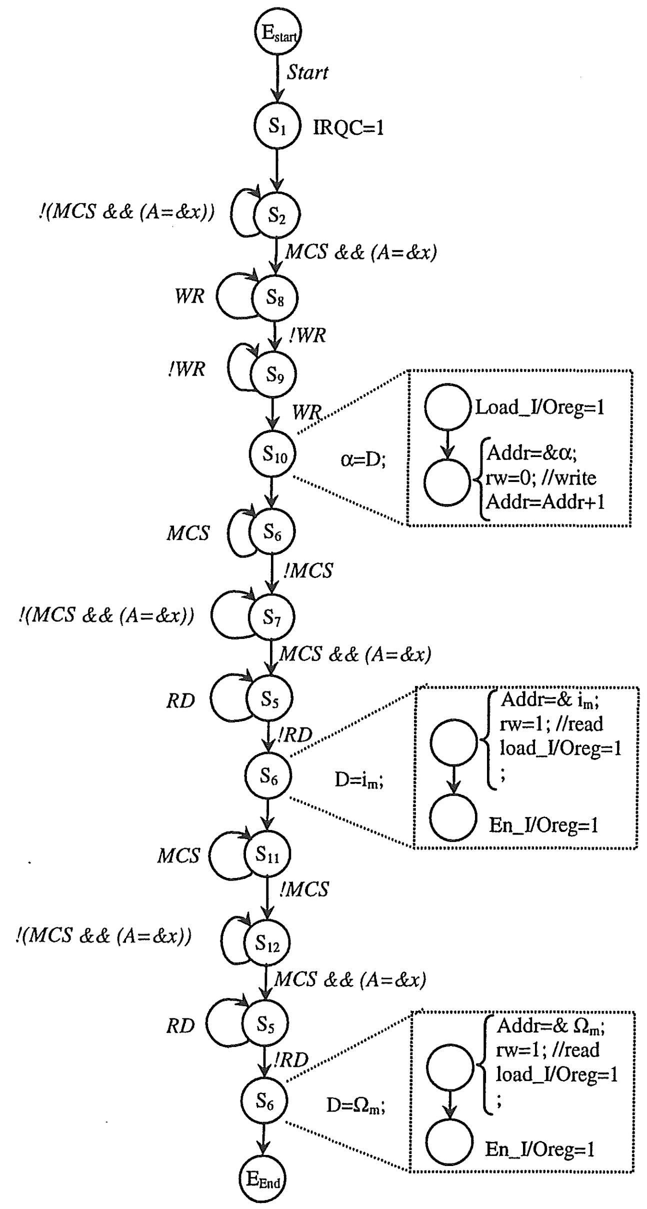


Figure 18: HW Communication SFSMDs

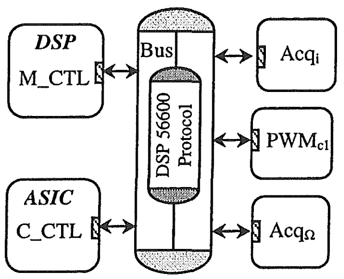


Figure 19: Communication model after protocol insertion (ARCH2)

The communication model is obtained as described in the ARCH1 exploration, using two steps: the protocol insertion (protocol of the DSP 56600) and the protocol inlining. The obtained communication model is shown on figure 20.

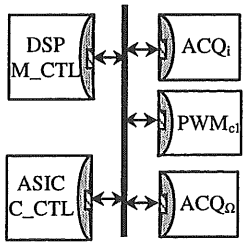


Figure 20: Communication model after protocol inlining (ARCH2)

We note that connections are done between memories (inside the I/O models) and PE1/PE2 according to the figure 21.

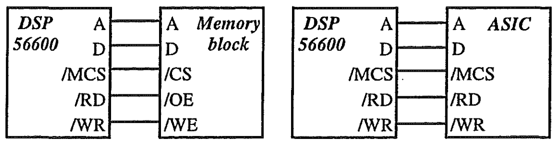


Figure 21: Components interconnections (ARCH2)

In our application, only one side of the register control is done by the master: for example for the Rα (register that stored α), it is controlled by PE2 only in writing operation, so the /WE signal is connected to the master /WR signal while the /OE is active controlled in local by the PWM hardware.
In the FSMD of the I/O module, the register will be controlled by the FSMD-controller only on one way (partially). For example in the case of the PWM module this register is only controlled for read by the FSMD-controller while it is controlled for write for the Acq_i, by its own FSMD-controller. The master (PE2) when transferring data with the DSP does the other control side.

We note that in this application and for necessity we used at the same time high and low levels design.

6 Backend

In the backend, the behavioral descriptions of the components in the communication model are synthesized into a structural view of all the components in the system architecture. The functionality of each component is implemented on top of the component's RTL or instruction-set microarchitecture. In the process, timing is refined down to the level of individual clock cycles based on each component's clock period.

The backend process encompasses three parallel tasks for different parts of the communication model [2]:

- High-level/behavioral synthesis for custom hardware components: The behavioral PE description in the form of straight-line code is synthesized into a netlist of register-transfer level (RTL) components.
- Software synthesis: The SpecC model of the behaviors mapped onto a programmable processor is converted into a C model, compiled into the processor's instruction set and possibly linked against an RTOS.
- Synthesis of bus interfaces and bus drivers: The application and protocol layer functionality is synthesized into a cycle-true 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 result of the backend process is the cycle-accurate implementation model.

In the communication model, components were modeled by PE behaviors containing a purely behavioral description of the component functionalities. In the implementation model, the PE behaviors are refined into cycle-true descriptions based on the component's RTL/instruction-set microarchitectures.

The implementation model is the result of the backend process and as such the final end-result of the whole system design flow. It is a structural description of both system and component architectures.
At the top-level, the system architecture is a set of non-terminating, concurrent components communicating via wires of system busses. At the component level, computation and communication functionalities are based on each component's microarchitecture: FSMD models for custom hardware and instruction-set models for software on programmable processors.
The implementation model is a cycle-accurate system description. The order and timing of computation and computation in the system is described in terms of component clocks. A global order is imposed among the system's components via the order of events on the common bus wires, generated and watched by the components connecting to the busses.
The implementation model is further processed and refined through traditional design flows down to manufacturing. For example, logic synthesis of custom hardware RTL descriptions is followed by *"place & route"* to generate the final chip layout.

7 Conclusions

In this report we introduce the SpecC system-level design methodology to the design of control systems for Power Electronics and Electric drives processes. We presented the study of a DC motor drive, which can be easily generalized to any other process control.

We have shown the various steps in the SpecC methodology that gradually refines the initial specification down to an actual communication model ready for implementation and manufacturing.
Starting with the executable SpecC specification, architecture exploration creates an architecture model of the control system through the steps of allocation, partitioning and scheduling. Communication synthesis then transforms the abstract communication of the architecture model into an implementation. After protocol selection, transducer synthesis and protocol inlining, the final communication model is obtained. This model is the result of the design process and will be handed off to the backend process for synthesis of the software and hardware parts.

The retained architecture target is usually obtained after estimations and analyzes of different modules in the specification model. However, in this report we presented two main architecture solutions that seems to be the most useful in Electric drive systems. The choice between them will be done according to the application constraints.

This project has shown that the SpecC methodology will result in significant productivity gains in the design of control systems. In fact:

➢ The SpecC methodology is based on the SpecC language which presents major advantages such as:

  - The SpecC language is a superset of C allows for drawing from the large body of existing algorithms. Therefore all control algorithms written in C language can be used and easily implemented through the SpecC methodology. On the other hand the control algorithm developers can be converted easily to the SpecC language and use it for the specification and validation of their new algorithms. Then, no rewritten of these programs will be needed between algorithm developers and control system designers since they use the same language. In general, communication among designers and customers will be largely minimized.
  - The SpecC language explicitly supports all the features necessary for system-level design

including hierarchy, timing, concurrency, communication and synchronization, exceptions, state transitions, and so on.
  - The clear separation of communication and computation in SpecC facilitates reuse of system components and enables easy integration of IP

- The 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.

- At every point, a model in SpecC language represents the design. This allows performing equivalence checking and simulation on each model to validate the transformations.

- The well-defined nature of the models and transformations provides the basis for design automation tools and in general enables application of formal methods for verification. These automation tools will cover a large part of the tedious and error-prone synthesis tasks and then reduce even further the time-to-silicon. To these tools some libraries specific to control systems design will be added in order to reduce the amount of resources and the man power required to complete a System-On-Chip design. A steep learning curve and the low designer expertise required, reduce the training overhead and limit the demand for highly qualified designers.

In future works, we will develop some libraries specific to the control system design and apply the SpecC methodology to the design of new sophisticated algorithms.

Acknowledgments

The authors would like to thank the Fulbright Scholar Program for supporting this project. We would also like to thank Andreas Gerstlauer and Rainer Doemer for 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] H. Yin, H. Du, T. Lee, D. Gajski, "Design of a JPEG Encoder using SpecC Methodology", University of California, Irvine, Technical Report ICS-TR-00-23, July 2000

[4] 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

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

A Specification Model for the DC System Control

B Architecture Model for the DC System Control (ARCH1)

C Communication Model for the DC System Control (ARCH1)

D Architecture Model for the DC System Control (ARCH2)

E Communication Model for the DC System Control (ARCH2)

```
////////////////////////////////////////////////////////////////
// SPEC_CMD_MCC1                                              //
//                                                            //
//                   SPECIFICATION MODEL                      //
//                                                            //
// .                   CONTROL DESIGN                         //
//                                                            //
// CASE OF A DC MOTOR WITH CURRENT AND SPEED SENSORS          //
//                                                            //
// Slim Ben Saoud                                             //
// CECS-UCI July 2001                                         //
//                                                            //
////////////////////////////////////////////////////////////////

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


#include "typedef.sh"
#include "clk.sc"
#include "mcc_command.sc"
#include "mcc_process.sc"


// STORAGE DEVICE SPECIFICATIONS
//////////////////////////////////

// Storage clock

behavior Clock_save(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESAVE);
            notify(clk);
            }while (SimActiv());
        }
    };

// Storage behavior

behavior Storage(in event clk, in float im, in float vit)
    {
    void main(void)
        {
        pTrace=fopen("R_spec_cmd_mcc1.dat","w");

        do
            {
            wait clk,EndofSim;
            fprintf(pTrace," %6f  %6f   \n",  im, vit);
            }while (SimActiv());

        fclose(pTrace);

        printf("im=%6f \n",im);
        printf("vit=%6f \n",vit);
        puts("Exiting...");

        exit(0);
        }
    };

// Storage global module

behavior Storage_glob(in float im, in float vit)
    {
    event           StorClk;

    Clock_save      Clock_save1(StorClk);
    Storage         Storage1(StorClk, im, vit);

    void main(void)
        {
        par{
            Clock_save1.main();
```

```
            Storage1.main();
            }
        }
    };


//TESTBENCHS
/////////////

behavior Main
    {
    bit[1:0]    c=0, S=0;
    float       vref=100.0,iref=0.0,Vh=0.0, Current=0.0, imc=0.0, Vitesse=0.0;
    int         Nim=512;


    Storage_glob    Storage_glob1(Current, Vitesse);
    Com_glob        Com_glob1(vref, S, Nim, c);
    Motor_glob      Motor_glob1(c, Current, Vitesse, Vh, Nim, S);

    int main(void)
        {
        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);
        printf("Simulation time=%d us \n",sim_time1);

        puts("starting...");

        par {
            Com_glob1.main();
            Motor_glob1.main();
            Storage_glob1.main();
            }


        return(0);
        }
    };

//EOF


////////////////////////////////////////////////////////////////
// TYPEDEF.sh                                                 //
//                 SPEC_CMD_MCC1 (July 2001)                  //
////////////////////////////////////////////////////////////////

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

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  1000000
#define CYCLESAVE   100
#define ALPHA       200

// Electromechanical system parameters

#define Ve          60.000000   //Power voltage
#define Alpha       0.999750    //0.999500
#define Betta       -0.000046   //-0.000092
#define Gamma       0.000250    //0.000500
#define Londa       0.000057    //0.000146
#define Mu          1.000000
#define Psi         -0.000127

#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

// Current Regulator

#define te          0.000284
#define kc          2.35
```

```
#define tc          0.004
#define CYCLETIME_comi  284
#define p1          1024        // Digital/Analog Converter resolution


// Speed Regulator

#define tev     0.02
#define kv      0.087
#define tv      8.8
#define CYCLETIME_comv  20000

// Results storage file

FILE    *pTrace;
FILE    *pTrace1;

// Simulation parameters

int sim_time1=5000;

bool SimActiv();

event EndofSim;


// UTILITY FUNCTION
////////////////////

bool SimActiv()
    {
    return ((int)now()<sim_time1);
    }


///////////////////////////////////////////////////////////////
// CLK.sc                                                    //
//              SPEC_CMD_MCC1 (July 2001)                    //
///////////////////////////////////////////////////////////////

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

behavior Clock(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while (SimActiv());

        notify EndofSim;
        }
    };


///////////////////////////////////////////////////////////////
// MCC_process.sc                                            //
//              SPEC_CMD_MCC1 (July 2001)                    //
///////////////////////////////////////////////////////////////


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

// Converter Module

behavior Converter(in event clk, in bit[1:0] cd, out float Vout)
    {
    float   a,b;
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            a=cd[0];
            b=cd[1];
            Vout=(a-b)*Ve;
```

```
            }while (SimActiv());
        }
    };

// Electric Module

behavior Electric(in event clk, in float vit, in float Vout, inout float im)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            im=Alpha*im+Betta*vit+Gamma*Vout;
            }while (SimActiv());
        }
    };

// Mechanical Module

behavior Mecanic(in event clk, in float im, inout float vit)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            vit=Londa*im+Mu*vit;
            }while (SimActiv());
        }
    };


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

// Current sensor Clock

behavior Clock_sen1(event clk1)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN1);
            notify(clk1);
            }while (SimActiv());
        }
    };

// Current sensor Generator

behavior Gen_cur(in event clk1, in float im, out int Nim)
    {
    void main(void)
        {
        do
            {
            wait clk1,EndofSim;
            Nim=(int)(p*(im+imax)/(2*imax));
            }while (SimActiv());
        }
    };

// Current sensor global module

behavior    Sen_cur(in float im, out int Nim)
    {
    event   clk_cur;

    Clock_sen1  Clock_sen11(clk_cur);
    Gen_cur     Gen_cur1(clk_cur, im, Nim);

    void main(void)
        {
        par{
            Clock_sen11.main();
            Gen_cur1.main();
            }
        }
    };

// Speed sensor Clock

behavior Clock_sen2(event clk2)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN2);
            notify(clk2);
            }while (SimActiv());
        }
    };

// Speed "refreshment"

behavior Raf_speed(in event clk2, in float vit, out int T4_vit, out bool s_vit)
    {
    float viti;
    void main(void)
        {
        do
            {
            wait clk2,EndofSim;
            viti=vit;
            if (viti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(viti)<0.5) viti=0.5;           //Limite basse de vitesse a reproduire

            T4_vit=(int)(pi*1e6/(n_cio*viti));      //quart de periode en us
            }while (SimActiv());
        }
    };

//Speed signals generation

behavior Gen_speed(in event clk, in int T4_vit,in bool s_vit, out bit[1:0] S)
    {
    int     i=0;

    void main(void)
        {

        do
            {
            wait clk,EndofSim;

            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;

            }while (SimActiv());

        }

    };

// Speed sensor global module

behavior    Sen_speed(in float vit, out bit[1:0] S)
    {
    event   Clk, clk_vit;
    int     T4_vit=24000;
    bool    s_vit=true;

    Clock       Clock3(Clk);
    Clock_sen2  Clock_sen21(clk_vit);
    Raf_speed   Raf_speed1(clk_vit, vit, T4_vit, s_vit);
    Gen_speed   Gen_speed1(Clk, T4_vit, s_vit, S);

    void main(void)
        {
        par(
            Clock3.main();
            Clock_sen21.main();
            Raf_speed1.main();
            Gen_speed1.main();
            }
        }
    };


// Electromechanical system global module

behavior Motor_glob(in bit[1:0] cd, inout float im, inout float vit,
                        inout float Vout, out int Nim, out bit[1:0] S)
    {
    event       Clk;

    Clock           Clock2(Clk);
    Converter       Converter1(Clk, cd, Vout);
    Electric        Electric1(Clk, vit, Vout, im);
    Mecanic         Mecanic1(Clk, im, vit);

    Sen_cur         Sen_cur1(im, Nim);
    Sen_speed       Sen_speed1(vit, S);

    void main(void)
        {
        par {
            Clock2.main();
            Converter1.main();
            Electric1.main();
            Mecanic1.main();

            Sen_cur1.main();
            Sen_speed1.main();
            }

        }
    };


/////////////////////////////////////////////////////////////
// mcc_command.sc                                          //
//              SPEC_CMD_MCC1 (July 2001)                  //
/////////////////////////////////////////////////////////////

#include "mcc_control_io.sc"
#include "mcc_control_iw.sc"

// Command global module

behavior Com_glob(in float vitref, in bit[1:0] S, in int Nim, inout bit[1:0] c)
    {
    event   Clk;
    int     vit=131070;
    int     alph=142, imc=512;
    Clock           Clock1(Clk);
    control_iw      Control_iw1(vitref, vit, imc, alph);
    ACQ_speed       ACQ_speed1(Clk,S, vit);
    ACQ_cur         ACQ_cur1(Clk, c, Nim, imc);
    Command_rap     Command_rap1(Clk, alph, c);

    void main(void)
        {
        par {
            Clock1.main();
            ACQ_speed1.main();
            ACQ_cur1.main();
            Command_rap1.main();

            Control_iw1.main();
            }

        }
    };

////////////////////////////////////////////////////////////
// mcc_control_io.sc                                      //
//                SPEC_CMD_MCC1 (July 2001)               //
////////////////////////////////////////////////////////////

// Speed acquisition

behavior ACQ_speed(event clk, in bit[1:0] S, out int vit)
    {
    int     viti, vit1=65535, vit2=65535;
    bool    sign_v=0;
    bool    flag1=0;
    int     i=0, j=0;

    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            if (flag1==0)
                {
                if (S[0]==1)
                    {
                    vit2=j;
                    j=0;
                    flag1=1;

                    if (S[1]==0)    sign_v=0;       // Rotation sens//
                    else    sign_v=1;

                    viti=vit1+vit2;
                    if (sign_v==1) viti=-viti;
                    vit=viti;

                    }
                else
                    {
                    j=j+1;
                    }
                }
            else
                {
                if (S[0]==0)
                    {
                    vit1=i;
                    i=0;
                    flag1=0;
                    }
                else
                    {
                    i=i+1;
                    }
                }

            }while (SimActiv());
        }
    };

// Current acquisition & Current average computing
```

```
behavior ACQ_cur(event clk, in bit[1:0] c, in int Nim, out int imc)
    {
    int im1=512, im2=512;
    bool ccomp;

    void main(void)
        {
        ccomp=0;
        do
            {
            wait clk,EndofSim;
            if (ccomp==1)
                {
                if (c[1]==1)
                    {
                    im1=Nim;
                    ccomp=0;
                    }
                }
            else
                {
                if (c[0]==1)
                    {
                    im2=Nim;
                    imc=(im1+im2)>>1;
                    ccomp=1;
                    }
                }
            }while (SimActiv());
        }

    };

// Control Signals Generator

behavior Command_rap(in event clk, in int alph1, out bit[1:0] cd)
    {
    int i=0;
    int alph=140;

    void main(void)
        {
        alph=alph1;
        do
            {
            wait clk,EndofSim;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i==284)
                    {
                    i=0;
                    alph=alph1;
                    }
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (SimActiv());
        }
    };

////////////////////////////////////////////////////////////
// mcc_control_iw.sc                                      //
//                SPEC_CMD_MCC1 (July 2001)               //
////////////////////////////////////////////////////////////

#include "mcc_control_i.sc"
#include "mcc_control_w.sc"

// Command global module

behavior control_iw(in float vitref, in int vit, in int imc, out int alph)
    {
    event       Clk_i, Clk_v;
    float       iref=0.0;
    Clock_comv      Clock_comv1(Clk_v);
    SpeCTL          SpeCTL1(Clk_v, vitref, vit, iref);

    Clock_comi      Clock_comi1(Clk_i);
    CurCTL          CurCTL1(Clk_i, iref, imc, alph);

    void main(void)
        {
        par {
            Clock_comv1.main();
            SpeCTL1.main();

            Clock_comi1.main();
            CurCTL1.main();
            }

        }
    };


///////////////////////////////////////////////////////////////
// mcc_control_i.sc                                          //
//               SPEC_CMD_MCC1 (July 2001)                   //
///////////////////////////////////////////////////////////////

// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

// Current controller Clock

behavior Clock_comi(event clk)
    {
    void main(void)
        {
        waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comi);
            notify(clk);
            }while (SimActiv());
        }
    };


// Current controller algorithm

behavior CurCTL(event clk_comi,in float iref, in int imc, out int alph)
    {
    float epsi, epsi1, integi, vmoy, alpha;
    float im, imcf;

    void main(void)
        {
        epsi=epsi1=integi=vmoy=alpha=0.0;
        do
            {
            wait clk_comi,EndofSim;

            imcf=imc;
            im=2*imcf*imax/p1 - imax;

            epsi=iref-im;
            integi=integi+((epsi+epsi1)/2)*te;
            vmoy=kc*epsi+(kc*integi)/tc;
            alpha=vmoy/(2*Ve)+0.5;
            if (alpha>0.95) alpha=0.95;
            if (alpha<0.06) alpha=0.06;
            alph=(int)(alpha*284);
            epsi1=epsi;
            }while (SimActiv());
        }
    };


///////////////////////////////////////////////////////////////
// mcc_control_w.sc                                          //
//               SPEC_CMD_MCC1 (July 2001)                   //
///////////////////////////////////////////////////////////////
```

```
// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

//Speed Controller Clock

behavior Clock_comv(event clk)
    {
    void main(void)
        {
        waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comv);
            notify(clk);
            }while (SimActiv());
        }
    };


// Speed Controller Algorithm

behavior SpeCTL(event clk_comv,in float vref, in int vit, out float iref)
    {
    float epsv, epsv1, integv, v;
    float vitf, irefi;
    void main(void)
        {
        epsv=epsv1=integv=0.0;
        do
            {
            wait clk_comv,EndofSim;

            vitf=vit;
            v=4*pi*1e6/(vitf*n_cio);
            epsv = vref - v;
            integv = integv +((epsv+epsv1)/2)*tev;
            irefi=kv*epsv+(kv*integv)/tv;
            epsv1=epsv;

            if (irefi>13.0) irefi=13.0;
            if (irefi<-13.0) irefi=-13.0;
            iref=irefi;
            }while (SimActiv());
        }

    };
```

```
/////////////////////////////////////////////////////////////////
// ARCH1_CMD_MCC                                               //
//                                                             //
//                  ARCHITECTURE MODEL                         //
//          (DSP for Control Alg. & ASIC for I/O)              //
//                    CONTROL DESIGN                           //
//                                                             //
// 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 "clk.sc"
#include "mcc_command.sc"
#include "mcc_process.sc"


// STORAGE DEVICE SPECIFICATIONS
/////////////////////////////////

// Storage clock

behavior Clock_save(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESAVE);
            notify(clk);
            }while (SimActiv());
        }
    };

// Storage behavior

behavior Storage(in event clk, in float im, in float vit)
    {
    void main(void)
        {
        pTrace=fopen("R_arch1_cmd_mcc.dat","w");

        do
            {
            wait clk,EndofSim;
            fprintf(pTrace," %6f  %6f   \n",  im, vit);
            }while (SimActiv());

        fclose(pTrace);

        printf("im=%6f \n",im);
        printf("vit=%6f \n",vit);
        puts("Exiting...");

        exit(0);
        }
    };

// Storage global module

behavior Storage_glob(in float im, in float vit)
    {
    event           StorClk;

    Clock_save      Clock_save1(StorClk);
    Storage         Storage1(StorClk, im, vit);

    void main(void)
        {
        par{
            Clock_save1.main();
```

```
            Storage1.main();
            }
        }
    };


//TESTBENCHS
/////////////

behavior Main
    {
    bit[1:0]    c=0, S=0;
    float       vref=100.0,iref=0.0,Vh=0.0, Current=0.0, imc=0.0, Vitesse=0.0;
    int         Nim=512;


    Storage_glob    Storage_glob1(Current, Vitesse);
    Com_glob        Com_glob1(vref, S, Nim, c);
    Motor_glob      Motor_glob1(c, Current, Vitesse, Vh, Nim, S);

    int main(void)
        {
        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);
        printf("Simulation time=%d us \n",sim_time1);

        puts("starting...");

        par {
            Com_glob1.main();
            Motor_glob1.main();
            Storage_glob1.main();
            }

        return(0);
        }
    };

//EOF

/////////////////////////////////////////////////////////////////
// TYPEDEF.sh                                                  //
//                  ARCH1_CMD_MCC (July 2001)                  //
/////////////////////////////////////////////////////////////////

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

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  1000000
#define CYCLESAVE   100
#define ALPHA       200

// Electromechanical system parameters

#define Ve          60.000000   //Power voltage
#define Alpha       0.999750    //0.999500
#define Betta       -0.000046   //-0.000092
#define Gamma       0.000250    //0.000500
#define Londa       0.000057    //0.000146
#define Mu          1.000000
#define Psi         -0.000127

#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

// Current Regulator

#define te          0.000284
#define kc          2.35
#define tc          0.004
#define CYCLETIME_comi  284
```

```
#define p1          1024        // Digital/Analog Converter resolution


// Speed Regulator

#define tev     0.02
#define kv      0.087
#define tv      8.8
#define CYCLETIME_comv  20000

// Results storage file

FILE    *pTrace;
FILE    *pTrace1;

// Simulation parameters

int sim_time1=5000;

bool SimActiv();

event EndofSim;


// UTILITY FUNCTION
////////////////////

bool SimActiv()
    {
    return ((int)now()<sim_time1);
    }


///////////////////////////////////////////////////////////////
// CLK.sc                                                    //
//              ARCH1_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////

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

behavior Clock(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while (SimActiv());

        notify EndofSim;
        }
    };


///////////////////////////////////////////////////////////////
// MCC_process.sc                                            //
//                                                           //
//              ARCH1_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////

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

// Converter Module

behavior Converter(in event clk, in bit[1:0] cd, out float Vout)
    {
    float   a,b;
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            a=cd[0];
            b=cd[1];
            Vout=(a-b)*Ve;
            }while (SimActiv());
        }
```

```
    };

// Electric Module

behavior Electric(in event clk, in float vit, in float Vout, inout float im)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            im=Alpha*im+Betta*vit+Gamma*Vout;
            }while (SimActiv());
        }
    };

// Mechanical Module

behavior Mecanic(in event clk, in float im, inout float vit)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            vit=Londa*im+Mu*vit;
            }while (SimActiv());
        }
    };


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

// Current sensor Clock

behavior Clock_sen1(event clk1)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN1);
            notify(clk1);
            }while (SimActiv());
        }
    };

// Current sensor Generator

behavior Gen_cur(in event clk1, in float im, out int Nim)
    {
    void main(void)
        {
        do
            {
            wait clk1,EndofSim;
            Nim=(int)(p*(im+imax)/(2*imax));
            }while (SimActiv());
        }
    };

// Current sensor global module

behavior    Sen_cur(in float im, out int Nim)
    {
    event   clk_cur;

    Clock_sen1  Clock_sen11(clk_cur);
    Gen_cur     Gen_cur1(clk_cur, im, Nim);

    void main(void)
        {
        par{
            Clock_sen11.main();
            Gen_cur1.main();
            }
        }
    };
```

```
// Speed sensor Clock

behavior Clock_sen2(event clk2)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN2);
            notify(clk2);
            }while (SimActiv());
        }
    };

// Speed "refreshment"

behavior Raf_speed(in event clk2, in float vit, out int T4_vit, out bool s_vit)
    {
    float viti;

    void main(void)
        {
        do
            {
            wait clk2,EndofSim;
            viti=vit;
            if (viti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(viti)<0.5) viti=0.5;           //Limite basse de vitesse a reproduire

            T4_vit=(int)(pi*1e6/(n_cio*viti));      //quart de periode en us
            }while (SimActiv());
        }
    };

//Speed signals generation

behavior Gen_speed(in event clk, in int T4_vit,in bool s_vit, out bit[1:0] S)
    {
    int     i=0;

    void main(void)
        {

        do
            {
            wait clk,EndofSim;

            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;
```

```
            }while (SimActiv());
        }
    };

// Speed sensor global module

behavior    Sen_speed(in float vit, out bit[1:0] S)
    {
    event   Clk, clk_vit;
    int     T4_vit=24000;
    bool    s_vit=true;

    Clock       Clock3(Clk);
    Clock_sen2  Clock_sen21(clk_vit);
    Raf_speed   Raf_speed1(clk_vit, vit, T4_vit, s_vit);
    Gen_speed   Gen_speed1(Clk, T4_vit, s_vit, S);

    void main(void)
        {
        par{
            Clock3.main();
            Clock_sen21.main();
            Raf_speed1.main();
            Gen_speed1.main();
            }
        }
    };


// Electromechanical system global module

behavior Motor_glob(in bit[1:0] cd, inout float im, inout float vit, inout float Vout, out int Nim,
out bit[1:0] S)
    {
    event       Clk;

    Clock           Clock2(Clk);
    Converter       Converter1(Clk, cd, Vout);
    Electric        Electric1(Clk, vit, Vout, im);
    Mecanic         Mecanic1(Clk, im, vit);

    Sen_cur         Sen_cur1(im, Nim);
    Sen_speed       Sen_speed1(vit, S);

    void main(void)
        {
        par {
            Clock2.main();
            Converter1.main();
            Electric1.main();
            Mecanic1.main();

            Sen_cur1.main();
            Sen_speed1.main();
            }

        }
    };


///////////////////////////////////////////////////////////////
// mcc_command.sc                                            //
//                                                           //
//              ARCH1_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////

#include "mcc_cd_bus.sc"
#include "mcc_cd_HW.sc"
#include "mcc_cd_SW.sc"

// Current controller Clock

behavior Clock_comi(event clk)
    {
    void main(void)
        {
        waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comi);
            notify(clk);
            }while (SimActiv());
        }
    };

// Command global module

behavior Com_glob(in float vitref, in bit[1:0] S, in int Nim, inout bit[1:0] c)
    {
    event   clk, intc;

    Bus     bus0;

    Clock           Clock1(clk);
    ctl_sw          ctl_sw1(bus0, intc, vitref);
    IO_hw           IO_hw1(bus0, clk, S, Nim, c);
    Clock_comi      Clock_comi1(intc);

    void main(void)
        {
        par {
            Clock1.main();
            IO_hw1.main();

            Clock_comi1.main();
            ctl_sw1.main();
            }

        }
    };

///////////////////////////////////////////////////////////////
// mcc_cd_bus.sc                                             //
//              ARCH1_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////

#include "mcc_cd_channels.sc"

#define ADDR_alph   0
#define ADDR_im     10
#define ADDR_wm     20

interface IBus
    {
    void sendWord(int data, int addr);
    void recvWord(int* data, int addr);
    };

channel Bus()
    implements  IBus
    {
    CWord   C_alph;
    CWord   C_im;
    CWord   C_wm;

    void sendWord(int data, int addr)
        {
        switch (addr)
            {
            case ADDR_alph:     return  C_alph.send(data);
            case ADDR_im:       return  C_im.send(data);
            case ADDR_wm:       return  C_wm.send(data);
            }
        }

    void recvWord(int* data, int addr)
        {
        switch (addr)
            {
            case ADDR_alph:     return  C_alph.recv(data);
            case ADDR_im:       return  C_im.recv(data);
            case ADDR_wm:       return  C_wm.recv(data);
            }
        }
    };
```

```
///////////////////////////////////////////////////////////////
// mcc_cd_channels.sc                                        //
//                                                           //
//              ARCH1_CMD_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;
        }
    };


///////////////////////////////////////////////////////////////
// mcc_cd_HW.sc                                              //
//                                                           //
//              ARCH1_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////

#include "mcc_control_io.sc"

// Current controller algorithm

behavior IO_hw(IBus bus0, in event clk, in bit[1:0] S, in int Nim, inout bit[1:0] cd)
    {
    int imc=512, vit=131070, alph;
    int k_i=0;

    ACQ_cur      ACQ_cur1(cd, Nim, imc);
    ACQ_speed    ACQ_speed1(S, vit);
    Command_rap  Command_rap1(alph, cd);


    void main(void)
        {

        bus0.recvWord(&alph, ADDR_alph);
        bus0.sendWord(vit, ADDR_wm);
        bus0.sendWord(imc, ADDR_im);

        do
            {
            wait clk,EndofSim;
            k_i=k_i+1;
            if (k_i==284)
                {
                k_i=1;
                bus0.recvWord(&alph, ADDR_alph);
                bus0.sendWord(vit, ADDR_wm);
                bus0.sendWord(imc, ADDR_im);
                }
            Command_rap1.main();
            ACQ_cur1.main();
            ACQ_speed1.main();
            }while (SimActiv());

        }
    };


////////////////////////////////////////////////////////////////
// mcc_control_io.sc                                          //
//                                                            //
//               ARCH1_CMD_MCC (July 2001)                    //
////////////////////////////////////////////////////////////////


// Speed acquisition

behavior ACQ_speed(in bit[1:0] S, out int vit)
    {
    int     viti, vit1=65535, vit2=65535;
    bool    sign_v=0;
    bool    flag1=0;
    int     i=0, j=0;

    void main(void)
        {
            if (flag1==0)
                {
                if (S[0]==1)
                    {
                    vit2=j;
                    j=0;
                    flag1=1;

                    if (S[1]==0)    sign_v=0;        // Rotation sens//
                    else    sign_v=1;

                    viti=vit1+vit2;
                    if (sign_v==1) viti=-viti;
                    vit=viti;

                    }
                else
                    {
                    j=j+1;
                    }
                }
            else
                {
                if (S[0]==0)
                    {
                    vit1=i;
                    i=0;
                    flag1=0;
                    }
                else
                    {
                    i=i+1;
                    }
                }

        }
    };


// Current acquisition & Current average computing

behavior ACQ_cur(in bit[1:0] c, in int Nim, out int imc)
    {
    int im1=512, im2=512;
    bool ccomp=0;

    void main(void)
        {
            if (ccomp==1)
                {
                if (c[1]==1)
                    {
                    im1=Nim;
```

```
                    ccomp=0;
                    }
                }
            else
                {
                if (c[0]==1)
                    {
                    im2=Nim;
                    imc=(im1+im2)>>1;
                    ccomp=1;
                    }
                }
        }

    };


// Control Signals Generator

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

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


////////////////////////////////////////////////////////////////
// mcc_cd_SW.sc                                               //
//                                                            //
//               ARCH1_CMD_MCC (July 2001)                    //
////////////////////////////////////////////////////////////////

#include "mcc_control_i.sc"
#include "mcc_control_w.sc"


// Command global module

behavior ctl_sw(IBus bus0, in event intc, in float vitref)
    {
    float       iref=0.0;
    int         k_i=60;
    int         vit=131070;

    SpeCTL          SpeCTL1(vitref, vit, iref);
    CurCTL          CurCTL1(bus0, iref, vit);

    void main(void)
        {

        try
            {
            SpeCTL1.main();
            }

        interrupt(intc)
            {
            CurCTL1.main();
            }

        }
    };

////////////////////////////////////////////////////////////////
// mcc_control_i.sc                                           //
//                                                            //
//              ARCH1_CMD_MCC (July 2001)                           //
//////////////////////////////////////////////////////////////////


// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////



// Current controller algorithm

behavior Current(in float iref, in int imc, out int alph)
    {
    float epsi=0., epsi1=0., integi=0., vmoy=0., alpha=0.;
    float im, imcf;

    void main(void)
        {
            imcf=imc;
            im=2*imcf*imax/p1 - imax;
            epsi=iref-im;
            integi=integi+((epsi+epsi1)/2)*te;
            vmoy=kc*epsi+(kc*integi)/tc;
            alpha=vmoy/(2*Ve)+0.5;
            if (alpha>0.95) alpha=0.95;
            if (alpha<0.06) alpha=0.06;
            alph=(int)(alpha*284);
            epsi1=epsi;
        }
    };


behavior exchange_sw(IBus bus0, in int alph, out int vit, out int imc)
    {
    void main(void)
        {
        bus0.sendWord(alph, ADDR_alph);
        bus0.recvWord(&vit, ADDR_wm);
        bus0.recvWord(&imc, ADDR_im);
        }
    };

behavior CurCTL(IBus bus0, in float iref, out int vit)
    {
    int imc, alph=142;

    Current     Current1(iref, imc, alph);
    exchange_sw exchange_sw1(bus0, alph, vit, imc);

    void main(void)
        {
        exchange_sw1.main();
        Current1.main();
        }
    };


//////////////////////////////////////////////////////////////////
// mcc_control_w.sc                                             //
//                                                              //
//              ARCH1_CMD_MCC (July 2001)                       //
//////////////////////////////////////////////////////////////////


// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

//Speed Controller Clock

behavior Clock_comv(event clk)
    {
    void main(void)
        {
//      waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comv);
            notify(clk);
            }while (SimActiv());
```

```
        }
    };


// Speed Controller Algorithm

behavior Speed(event clk_comv,in float vref, in int vit, out float iref)
    {
    float epsv, epsv1, integv, v;
    float irefi, vitf;

    void main(void)
        {
        epsv=epsv1=integv=0.0;
        do
            {
            wait clk_comv,EndofSim;

            vitf=vit;
            v=4*pi*1e6/(vitf*n_cio);
            epsv = vref - v;
            integv = integv +((epsv+epsv1)/2)*tev;
            irefi=kv*epsv+(kv*integv)/tv;
            epsv1=epsv;

            if (irefi>13.0) irefi=13.0;
            if (irefi<-13.0) irefi=-13.0;
            iref=irefi;

            }while (SimActiv());
        }

    };

behavior SpeCTL(in float vref, in int vit, out float iref)
    {
    event   clk_v;

    Clock_comv  Clock_comv1(clk_v);
    Speed       Speed1(clk_v, vref, vit, iref);

    void main(void)
        {
        par
            {
            Clock_comv1.main();
            Speed1.main();
            }
        }
    };
```

```
//////////////////////////////////////////////////////////////
// COM1_CMD_MCC                                             //
//                                                          //
//                   COMMUNICATION MODEL                    //
//          (DSP for Control Alg. & ASIC for I/O)           //
//                     CONTROL DESIGN                       //
//                                                          //
// 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 "clk.sc"
#include "mcc_command.sc"
#include "mcc_process.sc"


// STORAGE DEVICE SPECIFICATIONS
/////////////////////////////////

// Storage clock

behavior Clock_save(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESAVE);
            notify(clk);
            }while (SimActiv());
        }
    };

// Storage behavior

behavior Storage(in event clk, in float im, in float vit)
    {
    void main(void)
        {
        pTrace=fopen("R_com1_cmd_mcc.dat","w");

        do
            {
            wait clk,EndofSim;
            fprintf(pTrace," %6f  %6f   \n",  im, vit);
            }while (SimActiv());

        fclose(pTrace);

        printf("im=%6f \n",im);
        printf("vit=%6f \n",vit);
        puts("Exiting...");

        exit(0);
        }
    };

// Storage global module

behavior Storage_glob(in float im, in float vit)
    {
    event           StorClk;

    Clock_save      Clock_save1(StorClk);
    Storage         Storage1(StorClk, im, vit);

    void main(void)
        {
        par{
            Clock_save1.main();
```

```
            Storage1.main();
            }
        }
    };


//TESTBENCHS
/////////////

behavior Main
    {
    bit[1:0]    c=0, S=0;
    float       vref=100.0,iref=0.0,Vh=0.0, Current=0.0, imc=0.0, Vitesse=0.0;
    int         Nim=512;


    Storage_glob    Storage_glob1(Current, Vitesse);
    Com_glob        Com_glob1(vref, S, Nim, c);
    Motor_glob      Motor_glob1(c, Current, Vitesse, Vh, Nim, S);

    int main(void)
        {
        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);
        printf("Simulation time=%d us \n",sim_time1);

        puts("starting...");

        sim_time1=sim_time1*1000;

        par {
            Com_glob1.main();
            Motor_glob1.main();
            Storage_glob1.main();
            }

        return(0);
        }
    };

//EOF


//////////////////////////////////////////////////////////////
// TYPEDEF.sh                                               //
//                                                          //
//              COM1_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////

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

// Time references

#define CYCLETIME   1000                        //Multiplay by 1000***ns//
#define CYCLENB     50
#define MAXSIMTIME  5000000                     //Multiplay by 1000***ns//
#define CYCLESAVE   100000                      //Multiplay by 1000***ns//
#define ALPHA       200

// Electromechanical system parameters

#define Ve          60.000000   //Power voltage
#define Alpha       0.999750    //0.999500
#define Betta       -0.000046   //-0.000092
#define Gamma       0.000250    //0.000500
#define Londa       0.000057    //0.000146
#define Mu          1.000000
#define Psi         -0.000127

#define CYCLESEN1   5000                        //Multiplay by 1000***ns//
#define CYCLESEN2   1000000                     //Multiplay by 1000***ns//
#define p           1024        // Digital/Analog Converter resolution
#define n_cio       256         // CIO resolution
#define pi          3.1415927

#define imax        30

// Current Regulator
```

```
#define te          0.000284
#define kc          2.35
#define tc          0.004
#define CYCLETIME_comi  284000                    //Multiplay by 1000***ns//
#define p1          1024        // Digital/Analog Converter resolution


// Speed Regulator

#define tev     0.02
#define kv      0.087
#define tv      8.8
#define CYCLETIME_comv  20000000                  //Multiplay by 1000***ns//


#define ADDR_alph   0
#define ADDR_im     10
#define ADDR_wm     20

#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;

// Results storage file

FILE    *pTrace;
FILE    *pTrace1;

// Simulation parameters

int sim_time1=5000;

bool SimActiv();

event EndofSim;


// UTILITY FUNCTION
///////////////////

bool SimActiv()
    {
    return ((int)now()<sim_time1);
    }


///////////////////////////////////////////////////////////////
// CLK.sc                                                    //
//              COM1_CMD_MCC (July 2001)                     //
///////////////////////////////////////////////////////////////

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

behavior Clock(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while (SimActiv());

        notify EndofSim;
        }
    };


///////////////////////////////////////////////////////////////
// MCC_process.sc                                            //
//                                                           //
//              COM1_CMD_MCC (July 2001)                     //
///////////////////////////////////////////////////////////////

//ELECTROMECHANICAL SYSTEM
///////////////////////////
```

```
// Converter Module

behavior Converter(in event clk, in bit[1:0] cd, out float Vout)
    {
    float   a,b;
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            a=cd[0];
            b=cd[1];
            Vout=(a-b)*Ve;
            }while (SimActiv());
        }
    };

// Electric Module

behavior Electric(in event clk, in float vit, in float Vout, inout float im)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            im=Alpha*im+Betta*vit+Gamma*Vout;
            }while (SimActiv());
        }
    };

// Mechanical Module

behavior Mecanic(in event clk, in float im, inout float vit)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            vit=Londa*im+Mu*vit;
            }while (SimActiv());
        }
    };


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

// Current sensor Clock

behavior Clock_sen1(event clk1)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN1);
            notify(clk1);
            }while (SimActiv());
        }
    };

// Current sensor Generator

behavior Gen_cur(in event clk1, in float im, out int Nim)
    {
    void main(void)
        {
        do
            {
            wait clk1,EndofSim;
            Nim=(int)(p*(im+imax)/(2*imax));
            }while (SimActiv());
        }
    };

// Current sensor global module

behavior    Sen_cur(in float im, out int Nim)
```

```
    {
    event   clk_cur;

    Clock_sen1  Clock_sen11(clk_cur);
    Gen_cur     Gen_cur1(clk_cur, im, Nim);

    void main(void)
        {
        par{
            Clock_sen11.main();
            Gen_cur1.main();
            }
        }
    };


// Speed sensor Clock

behavior Clock_sen2(event clk2)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN2);
            notify(clk2);
            }while (SimActiv());
        }
    };

// Speed "refreshment"

behavior Raf_speed(in event clk2, in float vit, out int T4_vit, out bool s_vit)
    {
    float viti;

    void main(void)
        {
        do
            {
            wait clk2,EndofSim;
            viti=vit;
            if (viti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(viti)<0.5) viti=0.5;           //Limite basse de vitesse a reproduire

            T4_vit=(int)(pi*1e6/(n_cio*viti));      //quart de periode en us
            }while (SimActiv());
        }
    };

//Speed signals generation

behavior Gen_speed(in event clk, in int T4_vit,in bool s_vit, out bit[1:0] S)
    {
    int     i=0;

    void main(void)
        {

        do
            {
            wait clk,EndofSim;

            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;

            }while (SimActiv());

        }

    };

// Speed sensor global module

behavior    Sen_speed(in float vit, out bit[1:0] S)
    {
    event   Clk, clk_vit;
    int     T4_vit=24000;
    bool    s_vit=true;

    Clock       Clock3(Clk);
    Clock_sen2  Clock_sen21(clk_vit);
    Raf_speed   Raf_speed1(clk_vit, vit, T4_vit, s_vit);
    Gen_speed   Gen_speed1(Clk, T4_vit, s_vit, S);

    void main(void)
        {
        par{
            Clock3.main();
            Clock_sen21.main();
            Raf_speed1.main();
            Gen_speed1.main();
            }
        }
    };


// Electromechanical system global module

behavior Motor_glob(in bit[1:0] cd, inout float im, inout float vit, inout float Vout, out int Nim,
out bit[1:0] S)
    {
    event       Clk;

    Clock           Clock2(Clk);
    Converter       Converter1(Clk, cd, Vout);
    Electric        Electric1(Clk, vit, Vout, im);
    Mecanic         Mecanic1(Clk, im, vit);

    Sen_cur         Sen_cur1(im, Nim);
    Sen_speed       Sen_speed1(vit, S);

    void main(void)
        {
        par {
            Clock2.main();
            Converter1.main();
            Electric1.main();
            Mecanic1.main();

            Sen_cur1.main();
            Sen_speed1.main();
            }

        }
    };


/////////////////////////////////////////////////////////////
// mcc_command.sc                                          //
//                                                            //
//              COM1_CMD_MCC (July 2001)                      //
////////////////////////////////////////////////////////////////

#include "mcc_cd_sync.sc"
#include "mcc_cd_HW.sc"
#include "mcc_cd_SW.sc"

// Current controller Clock

behavior Clock_comi(event clk)
    {
    void main(void)
        {
//      waitfor(CYCLETIME);
//      notify(clk);
        do
            {
            waitfor(CYCLETIME_comi);
            notify(clk);
            }while (SimActiv());
        }
    };

// Command global module

behavior Com_glob(in float vitref, in bit[1:0] S, in int Nim, inout bit[1:0] c)
    {

    bit[15:0]   A;          // address
    bit[23:0]   D;          // data
    event       MCS;        // chip select
    bool        nRD;        // control lines
    bool        nWR;
    event       intC1;

    event   clk, intc;

    Clock           Clock1(clk);
    Clock_comi      Clock_comi1(intc);
    ctl_sw          ctl_sw1(A, D, MCS, nRD, nWR, intC1, vitref, intc);
    IO_hw           IO_hw1(A, D, MCS, nRD, nWR, intC1, clk, S, Nim, c);

    void main(void)
        {
        par {
            Clock1.main();
            IO_hw1.main();

            Clock_comi1.main();
            ctl_sw1.main();
            }
        }
    };


////////////////////////////////////////////////////////////////
//  mcc_cd_sync.sc                                            //
//                                                            //
//              COM1_CMD_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;
        }

    };


////////////////////////////////////////////////////////////////
// mcc_cd_HW.sc                                               //
//                                                            //
//              COM1_CMD_MCC (July 2001)                      //
////////////////////////////////////////////////////////////////

#include "mcc_cd_hw_protocol.sc"
#include "mcc_cd_hw_bus.sc"
#include "mcc_control_io.sc"

// Current controller algorithm

behavior IO_hw(bit[15:0]    A,          // address
               bit[23:0]    D,          // data
               event        MCS,        // chip select
               bool         nRD,        // control lines
               bool         nWR,
               out event    intC1,      // communication synchronization
               in event clk, in bit[1:0] S, in int Nim, inout bit[1:0] cd)
    {
    int imc=512, vit=131070, alph=142;
    int k_i=0;

    SlaveBus    bus0(A, D, MCS, nRD, nWR,intC1);

    ACQ_cur     ACQ_cur1(cd, Nim, imc);
    ACQ_speed   ACQ_speed1(S, vit);
    Command_rap Command_rap1(alph, cd);


    void main(void)
        {

        do
            {
            wait clk,EndofSim;
            k_i=k_i+1;
            if (k_i==284)
                {
                k_i=1;
                bus0.Slave_recvW(&alph, ADDR_alph);
                bus0.Slave_sendW(vit, ADDR_wm);
                bus0.Slave_sendW(imc, ADDR_im);
                }

            Command_rap1.main();
            ACQ_cur1.main();
            ACQ_speed1.main();
            }while (SimActiv());

        }
    };


////////////////////////////////////////////////////////////////
//  mcc_cd_hw_protocol.sc                                     //
//                                                            //
//              COM1_CMD_MCC (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;
        }
    };


///////////////////////////////////////////////////////////////
//  mcc_cd_hw_bus.sc                                         //
//                                                           //
//               COM1_CMD_MCC (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);
            }
        }

    };


///////////////////////////////////////////////////////////////
// mcc_control_io.sc                                         //
//                                                           //
//                COM1_CMD_MCC (July 2001)                   //
///////////////////////////////////////////////////////////////

// Speed acquisition

behavior ACQ_speed(in bit[1:0] S, out int vit)
    {
    int     viti, vit1=65535, vit2=65535;
    bool    sign_v=0;
    bool    flag1=0;
    int     i=0, j=0;

    void main(void)
        {
            if (flag1==0)
                {
                if (S[0]==1)
                    {
                    vit2=j;
                    j=0;
                    flag1=1;

                    if (S[1]==0)    sign_v=0;        // Rotation sens//
                    else    sign_v=1;

                    viti=vit1+vit2;
                    if (sign_v==1) viti=-viti;
                    vit=viti;

                    }
                else
                    {
                    j=j+1;
                    }
                }
            else
                {
                if (S[0]==0)
                    {
                    vit1=i;
                    i=0;
                    flag1=0;
                    }
                else
                    {
                    i=i+1;
                    }
                }
        }
    };


// Current acquisition & Current average computing

behavior ACQ_cur(in bit[1:0] c, in int Nim, out int imc)
    {
    int im1=512, im2=512;
    bool ccomp=0;

    void main(void)
        {
            if (ccomp==1)
                {
                if (c[1]==1)
                    {
                    im1=Nim;
                    ccomp=0;
                    }
                }
            else
                {
                if (c[0]==1)
                    {
                    im2=Nim;
                    imc=(im1+im2)>>1;
                    ccomp=1;
                    }
                }
        }

    };


// Control Signals Generator

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

    void main(void)
        {
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i==284)
                    {
                    i=0;
                    }
                cd[0]=0;
                }
            cd[1]=!cd[0];
        }
    };


//////////////////////////////////////////////////////////////
// mcc_cd_SW.sc                                             //
//                                                          //
//              COM1_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////


#include "mcc_cd_sw_protocol.sc"
#include "mcc_cd_sw_bus.sc"

#include "mcc_control_i.sc"
#include "mcc_control_w.sc"


// Command global module

behavior SwMain(IBusMaster bus0, in event intc, in float vitref)
    {
    float       iref=0.0;
    int         k_i=60;
    int         vit=131070;

    SpeCTL          SpeCTL1(vitref, vit, iref);
    CurCTL          CurCTL1(bus0, iref, vit);

    void main(void)
        {

        try
            {
            SpeCTL1.main();
            }

        interrupt(intc)
            {
```

```
            CurCTL1.main();
            }

        }
    };


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

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

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

behavior ctl_sw(bit[15:0]   A,          // address
                bit[23:0]   D,          // data
                event       MCS,        // chip select
                bool        nRD,        // control lines
                bool        nWR,
                in event    intC1,      //Communication synchronization
                in float    vitref,
                in event    intC)       //synchronization of the current control loop
    {
    CSync       intCflag;
    IntHandler  IntChandler(intCflag);

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


    void main(void)
        {
        try
            {
            SwMain1.main();
            }
        interrupt(intC1)
            {
            IntChandler.main();
            }
        }
    };


//////////////////////////////////////////////////////////////
//  mcc_cd_sw_bus.sc                                        //
//                                                          //
//              COM1_CMD_MCC (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);
            }
        }

    };


/////////////////////////////////////////////////////////////////
// mcc_control_i.sc                                           //
//                                                            //
//              COM1_CMD_MCC (July 2001)                      //
/////////////////////////////////////////////////////////////////

// CONTROL DEVICE SPECIFICATIONS
//////////////////////////////////

// Current controller algorithm

behavior Current(in float iref, in int imc, out int alph)
    {
    float epsi=0., epsi1=0., integi=0., vmoy=0., alpha=0.;
    float im, imcf;

    void main(void)
        {
            imcf=imc;
            im=2*imcf*imax/p1 - imax;
            epsi=iref-im;
            integi=integi+((epsi+epsi1)/2)*te;
            vmoy=kc*epsi+(kc*integi)/tc;
            alpha=vmoy/(2*Ve)+0.5;
            if (alpha>0.95) alpha=0.95;
            if (alpha<0.06) alpha=0.06;
            alph=(int)(alpha*284);
            epsi1=epsi;
        }
    };

behavior exchange_sw(IBusMaster bus0, in int alph, out int vit, out int imc)
    {
    void main(void)
        {
        bus0.Master_sendW(alph, ADDR_alph);
        bus0.Master_recvW(&vit, ADDR_wm);
        bus0.Master_recvW(&imc, ADDR_im);
        }
    };

behavior CurCTL(IBusMaster bus0, in float iref, out int vit)
    {
    int imc, alph=142;
```

```
    Current     Current1(iref, imc, alph);
    exchange_sw exchange_sw1(bus0, alph, vit, imc);

    void main(void)
        {
        exchange_sw1.main();
        Current1.main();
        }
    };


/////////////////////////////////////////////////////////////////
// mcc_control_w.sc                                           //
//                                                            //
//              COM1_CMD_MCC (July 2001)                      //
/////////////////////////////////////////////////////////////////

// CONTROL DEVICE SPECIFICATIONS
//////////////////////////////////

//Speed Controller Clock

behavior Clock_comv(event clk)
    {
    void main(void)
        {
//      waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comv);
            notify(clk);
            }while (SimActiv());
        }
    };


// Speed Controller Algorithm

behavior Speed(event clk_comv,in float vref, in int vit, out float iref)
    {
    float irefi, epsv, epsv1, integv, v, vitf;

    void main(void)
        {
        epsv=epsv1=integv=0.0;
        do
            {
            wait clk_comv,EndofSim;
            vitf=vit;
            v=4*pi*1e6/(vitf*n_cio);

            epsv = vref - v;
            integv = integv +((epsv+epsv1)/2)*tev;
            irefi=kv*epsv+(kv*integv)/tv;
            epsv1=epsv;

            if (irefi>13.0) irefi=13.0;
            if (irefi<-13.0) irefi=-13.0;
            iref=irefi;

            }while (SimActiv());
        }

    };

behavior SpeCTL(in float vref, in int vit, out float iref)
    {
    event   clk_v;

    Clock_comv  Clock_comv1(clk_v);
    Speed       Speed1(clk_v, vref, vit, iref);

    void main(void)
        {
        par
            {
            Clock_comv1.main();
            Speed1.main();
            }
    }
  };
```

```
//////////////////////////////////////////////////////////////
// ARCH2_CMD_MCC                                            //
//                                                          //
//                  ARCHITECTURE MODEL                      //
//                                                          //
//                    CONTROL DESIGN                        //
//                                                          //
// 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 "clk.sc"
#include "mcc_command.sc"
#include "mcc_process.sc"


// STORAGE DEVICE SPECIFICATIONS
/////////////////////////////////

// Storage clock

behavior Clock_save(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESAVE);
            notify(clk);
            }while (SimActiv());
        }
    };

// Storage behavior

behavior Storage(in event clk, in float im, in float vit)
    {
    void main(void)
        {
        pTrace=fopen("R_arch2b_cmd_mcc.dat","w");

        do
            {
            wait clk,EndofSim;
            fprintf(pTrace," %6f  %6f   \n",  im, vit);
            }while (SimActiv());

        fclose(pTrace);

        printf("im=%6f \n",im);
        printf("vit=%6f \n",vit);
        puts("Exiting...");

        exit(0);
        }
    };

// Storage global module

behavior Storage_glob(in float im, in float vit)
    {
    event           StorClk;

    Clock_save      Clock_save1(StorClk);
    Storage         Storage1(StorClk, im, vit);

    void main(void)
        {
        par{
            Clock_save1.main();
```

```
            Storage1.main();
            }
        }
    };


//TESTBENCHS
/////////////

behavior Main
    {
    bit[1:0]    c=0, S=0;
    float       vref=100.0,iref=0.0,Vh=0.0, Current=0.0, imc=0.0, Vitesse=0.0;
    int         Nim=512;


    Storage_glob    Storage_glob1(Current, Vitesse);
    Com_glob        Com_glob1(vref, S, Nim, c);
    Motor_glob      Motor_glob1(c, Current, Vitesse, Vh, Nim, S);

    int main(void)
        {
        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);
        printf("Simulation time=%d us \n",sim_time1);

        puts("starting...");

        par {
            Com_glob1.main();
            Motor_glob1.main();
            Storage_glob1.main();
            }

        return(0);
        }
    };

//EOF


//////////////////////////////////////////////////////////////
// TYPEDEF.sh                                               //
//                                                          //
//              ARCH2_CMD_MCC (July 2001)                   //
//////////////////////////////////////////////////////////////

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

// Time references

#define CYCLETIME   1
#define CYCLENB     50
#define MAXSIMTIME  1000000
#define CYCLESAVE   100
#define ALPHA       200

// Electromechanical system parameters

#define Ve          60.000000   //Power voltage
#define Alpha       0.999750    //0.999500
#define Betta       -0.000046   //-0.000092
#define Gamma       0.000250    //0.000500
#define Londa       0.000057    //0.000146
#define Mu          1.000000
#define Psi         -0.000127

#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

// Current Regulator

#define te          0.000284
#define kc          2.35
```

```
#define tc          0.004
#define CYCLETIME_comi  284
#define p1          1024       // Digital/Analog Converter resolution

#define p2          65536      // Conversion of iref in integer coded by 16 bits


// Speed Regulator

#define tev     0.02
#define kv      0.087
#define tv      8.8
#define CYCLETIME_comv  20000


// Communication parameters

#define ADDR_alph   0
#define ADDR_alphR  5
#define ADDR_im     10
#define ADDR_imR    15
#define ADDR_irefn  20
#define ADDR_wm     30
#define ADDR_wmR    35



// Results storage file

FILE    *pTrace;
FILE    *pTrace1;

// Simulation parameters

int sim_time1=5000;

bool SimActiv();

event EndofSim;


// UTILITY FUNCTION
////////////////////

bool SimActiv()
    {
    return ((int)now()<sim_time1);
    }


///////////////////////////////////////////////////////////////
// CLK.sc                                                    //
//              ARCH2_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////


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

behavior Clock(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME);
            notify(clk);
            }while (SimActiv());

        notify EndofSim;
        }
    };


///////////////////////////////////////////////////////////////
// MCC_process.sc                                            //
//                                                           //
//              ARCH2_CMD_MCC (July 2001)                    //
///////////////////////////////////////////////////////////////
```

```
//ELECTROMECHANICAL SYSTEM
///////////////////////////

// Converter Module

behavior Converter(in event clk, in bit[1:0] cd, out float Vout)
    {
    float   a,b;
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            a=cd[0];
            b=cd[1];
            Vout=(a-b)*Ve;
            }while (SimActiv());
        }
    };

// Electric Module

behavior Electric(in event clk, in float vit, in float Vout, inout float im)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            im=Alpha*im+Betta*vit+Gamma*Vout;
            }while (SimActiv());
        }
    };

// Mechanical Module

behavior Mecanic(in event clk, in float im, inout float vit)
    {
    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            vit=Londa*im+Mu*vit;
            }while (SimActiv());
        }
    };


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

// Current sensor Clock

behavior Clock_sen1(event clk1)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN1);
            notify(clk1);
            }while (SimActiv());
        }
    };

// Current sensor Generator

behavior Gen_cur(in event clk1, in float im, out int Nim)
    {
    void main(void)
        {
        do
            {
            wait clk1,EndofSim;
            Nim=(int)(p*(im+imax)/(2*imax));
            }while (SimActiv());
        }
    };
```

```
// Current sensor global module

behavior    Sen_cur(in float im, out int Nim)
    {
    event   clk_cur;

    Clock_sen1  Clock_sen11(clk_cur);
    Gen_cur     Gen_cur1(clk_cur, im, Nim);

    void main(void)
        {
        par{
            Clock_sen11.main();
            Gen_cur1.main();
            }
        }
    };


// Speed sensor Clock

behavior Clock_sen2(event clk2)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN2);
            notify(clk2);
            }while (SimActiv());
        }
    };

// Speed "refreshment"

behavior Raf_speed(in event clk2, in float vit, out int T4_vit, out bool s_vit)
    {
    float viti;

    void main(void)
        {
        do
            {
            wait clk2,EndofSim;
            viti=vit;
            if (viti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(viti)<0.5) viti=0.5;           //Limite basse de vitesse a reproduire

            T4_vit=(int)(pi*1e6/(n_cio*viti));      //quart de periode en us
            }while (SimActiv());
        }
    };

//Speed signals generation

behavior Gen_speed(in event clk, in int T4_vit,in bool s_vit, out bit[1:0] S)
    {
    int     i=0;

    void main(void)
        {

        do
            {
            wait clk,EndofSim;

            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;

            }while (SimActiv());
        }

    };

// Speed sensor global module

behavior    Sen_speed(in float vit, out bit[1:0] S)
    {
    event   Clk, clk_vit;
    int     T4_vit=24000;
    bool    s_vit=true;

    Clock       Clock3(Clk);
    Clock_sen2  Clock_sen21(clk_vit);
    Raf_speed   Raf_speed1(clk_vit, vit, T4_vit, s_vit);
    Gen_speed   Gen_speed1(Clk, T4_vit, s_vit, S);

    void main(void)
        {
        par{
            Clock3.main();
            Clock_sen21.main();
            Raf_speed1.main();
            Gen_speed1.main();
            }
        }
    };


// Electromechanical system global module

behavior Motor_glob(in bit[1:0] cd, inout float im, inout float vit, inout float Vout, out int Nim,
out bit[1:0] S)
    {
    event       Clk;

    Clock         Clock2(Clk);
    Converter     Converter1(Clk, cd, Vout);
    Electric      Electric1(Clk, vit, Vout, im);
    Mecanic       Mecanic1(Clk, im, vit);

    Sen_cur       Sen_cur1(im, Nim);
    Sen_speed     Sen_speed1(vit, S);

    void main(void)
        {
        par {
            Clock2.main();
            Converter1.main();
            Electric1.main();
            Mecanic1.main();

            Sen_cur1.main();
            Sen_speed1.main();
            }

        }
    };
```

```
///////////////////////////////////////////////////////////////
// mcc_command.sc                                            //
//                                                           //
//                  ARCH2_CMD_MCC (July 2001)                //
///////////////////////////////////////////////////////////////

#include "mcc_cd_sync.sc"
#include "mcc_cd_bus.sc"
#include "mcc_acq_cu.sc"
#include "mcc_acq_sp.sc"
#include "mcc_gen_pwm.sc"
#include "mcc_control_i.sc"
#include "mcc_control_w.sc"

// Command global module

behavior Com_glob(in float vitref, in bit[1:0] S, in int Nim, inout bit[1:0] c)
    {
    event   Clk, Clk_i, Clk_v;

    Bus     bus0;
    event   intci;

    Clock           Clock1(Clk);
    Clock_comv      Clock_comv1(Clk_v);
//  Clock_comi      Clock_comi1(Clk_i);

    CurCTL          CurCTL1(bus0, Clk_i);
    Sw_w            Sw_w1(bus0, Clk_i, Clk_v, vitref);
    ACQ_speed       ACQ_speed1(bus0, Clk,S);
    ACQ_cur         ACQ_cur1(bus0, Clk, c, Nim);
    Command_rap     Command_rap1(bus0, Clk, c, Clk_i);

    void main(void)
        {
        par {
            Clock1.main();

            Command_rap1.main();
            ACQ_speed1.main();
            ACQ_cur1.main();

            Clock_comv1.main();
            Sw_w1.main();
//          Clock_comi1.main();
            CurCTL1.main();
            }

        }
    };


///////////////////////////////////////////////////////////////
//  mcc_cd_sync.sc                                           //
//                                                           //
//                  ARCH2_CMD_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;
        }

    };

///////////////////////////////////////////////////////////////
// mcc_cd_bus.sc                                             //
//                                                           //
//                  ARCH2_CMD_MCC (July 2001)                //
///////////////////////////////////////////////////////////////

#include "mcc_cd_channels.sc"

interface IBus
    {
    void sendWord(int data, int addr);
    void recvWord(int* data, int addr);
    void writeWord(int data, int addr);
    void readWord(int* data, int addr);
    };

channel Bus()
    implements  IBus
    {
    CWord       C_alph;
    CWord       C_im;
    CWord       C_wm;
    CWord       C_irefn;
    CWord_alph  CW_alph;
    CWord_i     CW_i;
    CWord_v     CW_v;

    void sendWord(int data, int addr)
        {
        switch (addr)
            {
            case ADDR_alph:     return  C_alph.send(data);
            case ADDR_im:       return  C_im.send(data);
            case ADDR_irefn:    return  C_irefn.send(data);
            case ADDR_wm:       return  C_wm.send(data);
            case ADDR_alphR:    return  CW_alph.send(data);
            case ADDR_imR:      return  CW_i.send(data);
            case ADDR_wmR:      return  CW_v.send(data);
            }
        }

    void recvWord(int* data, int addr)
        {
        switch (addr)
            {
            case ADDR_alph:     return  C_alph.recv(data);
            case ADDR_im:       return  C_im.recv(data);
            case ADDR_irefn:    return  C_irefn.recv(data);
            case ADDR_wm:       return  C_wm.recv(data);
            case ADDR_alphR:    return  CW_alph.recv(data);
            case ADDR_imR:      return  CW_i.recv(data);
            case ADDR_wmR:      return  CW_v.recv(data);
            }
        }

    void writeWord(int data, int addr)
        {
        switch (addr)
            {
            case ADDR_alph:     return  CW_alph.send(data);
            case ADDR_im:       return  CW_i.send(data);
            case ADDR_wm:       return  CW_v.send(data);
            }
        }

    void readWord(int* data, int addr)
        {
        switch (addr)
```

```
            {
            wait clk,EndofSim;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i==284)
                    {
                    notify(clk_i);
                    i=0;
                    alph=alph_in;
                    }
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (SimActiv());

        }
    };

behavior Command_rap(IBus bus0, in event clk, out bit[1:0] cd, out event clk_i)
    {
    int alph=142;

    Reg_alph    Reg_alph1(bus0, alph);
    Command_gen Command_gen1(clk, alph, cd, clk_i);

    void main(void)
        {
        par{
            Reg_alph1.main();
            Command_gen1.main();
            }
        }
    };


/////////////////////////////////////////////////////////////////
// mcc_control_i.sc                                            //
//                                                             //
//               ARCH2_CMD_MCC (July 2001)                     //
/////////////////////////////////////////////////////////////////


// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

// Current controller Clock

behavior Clock_comi(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME_comi);
            notify(clk);
            }while (SimActiv());
        }
    };


behavior Exchange_i(IBus bus0, in int alph, out int imc, out int irefn)
    {
    void main(void)
        {
        bus0.sendWord(alph, ADDR_alph);
        bus0.recvWord(&imc, ADDR_im);
        bus0.recvWord(&irefn, ADDR_irefn);
        }
    };


// Current controller algorithm

behavior CurCTL(IBus bus0, event clk_comi)
    {
    float epsi, epsi1, integi, vmoy, alpha;
    float im, iref;
    int alph=142, imc=512;
```

```
    int irefn=32768;
    float irefnf=0.0;
    float imcf=0.0;

    Exchange_i  Exchange_i1(bus0, alph, imc, irefn);

    void main(void)
        {
        epsi=epsi1=integi=vmoy=alpha=0.0;

        do
            {
            wait clk_comi,EndofSim;

            if(SimActiv())              //To avoid Deadlock due to the execution of
                {                       //this behavior one time more than the speed one
                Exchange_i1.main();

                imcf=imc;
                im=2*imcf*imax/p1 - imax;
                irefnf=irefn;
                iref=2*irefnf*imax/p2 - imax;

                epsi=iref-im;
                integi=integi+((epsi+epsi1)/2)*te;
                vmoy=kc*epsi+(kc*integi)/tc;
                alpha=vmoy/(2*Ve)+0.5;
                if (alpha>0.95) alpha=0.95;
                if (alpha<0.06) alpha=0.06;
                alph=(int)(alpha*284);
                epsi1=epsi;
                }

            }while (SimActiv());

        }
    };


/////////////////////////////////////////////////////////////////
// mcc_control_w.sc                                            //
//                                                             //
//               ARCH2_CMD_MCC (July 2001)                     //
/////////////////////////////////////////////////////////////////


// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

//Speed Controller Clock

behavior Clock_comv(event clk)
    {
    void main(void)
        {
//      waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comv);
            notify(clk);
            }while (SimActiv());
        }
    };


behavior Exchange_w(IBus bus0, in int irefn)
    {
    int alph, imc;

    void main(void)
        {
        bus0.recvWord(&alph, ADDR_alph);

        bus0.sendWord(alph, ADDR_alphR);
        bus0.recvWord(&imc, ADDR_imR);

        bus0.sendWord(imc, ADDR_im);
        bus0.sendWord(irefn, ADDR_irefn);
        }
    };


// Speed Controller Algorithm

behavior SpeCTL(IBus bus0, event clk_comv,in float vref, out int irefn)
    {
    float epsv, epsv1, integv, v, iref=0.0;
    int vit;
    float vitf;

    void main(void)
        {
        epsv=epsv1=integv=0.0;
        do
            {
            wait clk_comv,EndofSim;

            bus0.recvWord(&vit, ADDR_wmR);

            vitf=vit;
            v=4*pi*1e6/(vitf*n_cio);
            epsv = vref - v;
            integv = integv +((epsv+epsv1)/2)*tev;
            iref=kv*epsv+(kv*integv)/tv;
            epsv1=epsv;

            if (iref>13.0) iref=13.0;
            if (iref<-13.0) iref=-13.0;

            irefn=(int)((iref+imax)*p2/(2*imax));

            }while (SimActiv());

        }
    };

behavior Sw_w(IBus bus0, in event intci, event clk_comv,in float vref)
    {
    int irefn=32768;

    Exchange_w  Exchange_w1(bus0, irefn);
    SpeCTL      SpeCTL1(bus0, clk_comv,vref, irefn);

    void main(void)
        {
        try{
            SpeCTL1.main();
            }
        interrupt(intci)
            {
            Exchange_w1.main();
            }
        }
    };
```

```
//////////////////////////////////////////////////////////////
// COM2_CMD_MCC                                             //
//                                                          //
//                    COMMUNICATION MODEL                   //
//  (DSP for Speed Control  & ASIC for Current Control)     //
//                      CONTROL DESIGN                      //
//                                                          //
// 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 "clk.sc"
#include "mcc_command.sc"
#include "mcc_process.sc"



// STORAGE DEVICE SPECIFICATIONS
/////////////////////////////////

// Storage clock

behavior Clock_save(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESAVE);
            notify(clk);
            }while (SimActiv());
        }
    };

// Storage behavior

behavior Storage(in event clk, in float im, in float vit)
    {
    void main(void)
        {
        pTrace=fopen("R_com2_cmd_mcc.dat","w");

        do
            {
            wait clk,EndofSim;
            fprintf(pTrace," %6f  %6f   \n",  im, vit);
            }while (SimActiv());

        fclose(pTrace);

        printf("im=%6f \n",im);
        printf("vit=%6f \n",vit);
        puts("Exiting...");

        exit(0);
        }
    };

// Storage global module

behavior Storage_glob(in float im, in float vit)
    {
    event           StorClk;

    Clock_save      Clock_save1(StorClk);
    Storage         Storage1(StorClk, im, vit);

    void main(void)
        {
        par{
```

```
            Clock_save1.main();
            Storage1.main();
            }
        }
    };


//TESTBENCHS
/////////////

behavior Main
    {
    bit[1:0]     c=0, S=0;
    float        vref=100.0,iref=0.0,Vh=0.0, Current=0.0, imc=0.0, Vitesse=0.0;
    int          Nim=512;


    Storage_glob    Storage_glob1(Current, Vitesse);
    Com_glob        Com_glob1(vref, S, Nim, c);
    Motor_glob      Motor_glob1(c, Current, Vitesse, Vh, Nim, S);

    int main(void)
        {
        printf("Enter the simulation time (integer) us:");
        scanf("%d",&sim_time1);
        printf("Simulation time=%d us \n",sim_time1);

        puts("starting...");

        sim_time1=sim_time1*1000;

        par {
            Com_glob1.main();
            Motor_glob1.main();
            Storage_glob1.main();
            }

        return(0);
        }
    };

//EOF


//////////////////////////////////////////////////////////////
// mcc_declar.h                                             //
//                                                          //
//               COM2_CMD_MCC (July 2001)                   //
//////////////////////////////////////////////////////////////

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

// Time references

#define CYCLETIME   1000
#define CYCLENB     50
#define MAXSIMTIME  1000000000
#define CYCLESAVE   100000
#define ALPHA       200

// Electromechanical system parameters

#define Ve          60.000000   //Power voltage
#define Alpha       0.999750    //0.999500
#define Betta       -0.000046   //-0.000092
#define Gamma       0.000250    //0.000500
#define Londa       0.000057    //0.000146
#define Mu          1.000000
#define Psi         -0.000127

#define CYCLESEN1   5000            // cycle de rafraichissement du courant
#define CYCLESEN2   1000000     // 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

// Current Regulator
```

```
            {
            wait clk1,EndofSim;
            Nim=(int)(p*(im+imax)/(2*imax));
            }while (SimActiv());
        }
    };

// Current sensor global module

behavior    Sen_cur(in float im, out int Nim)
    {
    event   clk_cur;

    Clock_sen1  Clock_sen11(clk_cur);
    Gen_cur     Gen_cur1(clk_cur, im, Nim);

    void main(void)
        {
        par{
            Clock_sen11.main();
            Gen_cur1.main();
            }
        }
    };


// Speed sensor Clock

behavior Clock_sen2(event clk2)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLESEN2);
            notify(clk2);
            }while (SimActiv());
        }
    };

// Speed "refreshment"

behavior Raf_speed(in event clk2, in float vit, out int T4_vit, out bool s_vit)
    {
    float viti;

    void main(void)
        {
        do
            {
            wait clk2,EndofSim;
            viti=vit;
            if (viti>0)  s_vit=true;
            else s_vit=false;

            if (fabs(viti)<0.5) viti=0.5;           //Limite basse de vitesse a reproduire

            T4_vit=(int)(pi*1e6/(n_cio*viti));      //quart de periode en us
            }while (SimActiv());
        }
    };

//Speed signals generation

behavior Gen_speed(in event clk, in int T4_vit,in bool s_vit, out bit[1:0] S)
    {
    int     i=0;

    void main(void)
        {

        do
            {
            wait clk,EndofSim;

            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;

            }while (SimActiv());

        }

    };

// Speed sensor global module

behavior    Sen_speed(in float vit, out bit[1:0] S)
    {
    event   Clk, clk_vit;
    int     T4_vit=24000;
    bool    s_vit=true;

    Clock       Clock3(Clk);
    Clock_sen2  Clock_sen21(clk_vit);
    Raf_speed   Raf_speed1(clk_vit, vit, T4_vit, s_vit);
    Gen_speed   Gen_speed1(Clk, T4_vit, s_vit, S);

    void main(void)
        {
        par{
            Clock3.main();
            Clock_sen21.main();
            Raf_speed1.main();
            Gen_speed1.main();
            }
        }
    };


// Electromechanical system global module

behavior Motor_glob(in bit[1:0] cd, inout float im, inout float vit, inout float Vout, out int Nim,
out bit[1:0] S)
    {
    event       Clk;

    Clock           Clock2(Clk);
    Converter       Converter1(Clk, cd, Vout);
    Electric        Electric1(Clk, vit, Vout, im);
    Mecanic         Mecanic1(Clk, im, vit);

    Sen_cur         Sen_cur1(im, Nim);
    Sen_speed       Sen_speed1(vit, S);

    void main(void)
        {
        par {
            Clock2.main();
            Converter1.main();
            Electric1.main();
            Mecanic1.main();
            Sen_cur1.main();
            Sen_speed1.main();
            }

        }
    };


//////////////////////////////////////////////////////////////
// mcc_command.sc                                           //
//                                                          //
//              COM2_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////

#include "mcc_cd_sync.sc"

#include "mcc_acq_cu.sc"
#include "mcc_acq_sp.sc"
#include "mcc_gen_pwm.sc"
#include "mcc_control_i.sc"
#include "mcc_control_w.sc"

// Command global module

behavior Com_glob(in float vitref, in bit[1:0] S, in int Nim, inout bit[1:0] c)
    {

    int         A;          // address
    int         D;          // data
    event       MCS;        // chip select
    bool        nRD;        // control lines
    bool        nWR;

    event   Clk, Clk_i, Clk_v;

    Clock           Clock1(Clk);
    Clock_comv      Clock_comv1(Clk_v);
//  Clock_comi      Clock_comi1(Clk_i);

    CurCTL          CurCTL1(A, D, MCS, nRD, nWR, Clk_i);
    Sw_w            Sw_w1(A, D, MCS, nRD, nWR, Clk_i, Clk_v, vitref);
    ACQ_speed       ACQ_speed1(A, D, MCS, nRD, nWR, Clk,S);
    ACQ_cur         ACQ_cur1(A, D, MCS, nRD, nWR, Clk, c, Nim);
    Command_rap     Command_rap1(A, D, MCS, nRD, nWR, Clk, c, Clk_i);

    void main(void)
        {
        par {
            Clock1.main();

            Command_rap1.main();
            ACQ_speed1.main();
            ACQ_cur1.main();

            Clock_comv1.main();
            Sw_w1.main();
//          Clock_comi1.main();
            CurCTL1.main();
            }

        }
    };


//////////////////////////////////////////////////////////////
//  mcc_cd_sync.sc                                          //
//                                                          //
//              COM2_CMD_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;
        }

    };

//////////////////////////////////////////////////////////////
// mcc_acq_cu.sc                                            //
//                                                          //
//              COM2_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////
interface IProtCu
    {
    void    Reg_RW(bit[15:0] addr, word24* data);
    };


channel CuProt(int A, int D, event MCS, bool nRD, bool nWR)
        implements IProtCu
    {
    void Reg_RW(bit[15:0] addr, word24* data)
        {
        t1: wait(MCS);
        if (A != addr) goto t1;

        while (nWR==1 && nRD==1)
            {
            waitfor(1);
            }
        if (nRD==0)
            {
            waitfor(6);
            D=*data;
            while (nRD==0)
                {
                waitfor(1);
                }
            }
        if (nWR==0)
            {
            waitfor(9);
            while (nWR==0)
                {
                waitfor(1);
                }
            *data=D;
            }
        }
    };

interface IBusCu
    {
    void    RegSlave_RW(word24* data, int   addr);
    };


channel CuBus(int A, int D, event MCS, bool nRD, bool nWR)
    implements IBusCu
    {
    CuProt  protocol(A, D, MCS, nRD, nWR);

    void RegSlave_RW(word24* data, int addr)
        {
        protocol.Reg_RW(addr, data);
        }
    };


// Current acquisition & Current average computing

behavior Reg_cur(IBusCu bus0, in int imc)
    {
    void main(void)
        {
        do
            {
            bus0.RegSlave_RW(&imc, ADDR_imR);
            }while (SimActiv());
        }
    };

behavior Meas_cur(in event clk, in bit[1:0] c, in int Nim, out int imc)
    {
    int im1=512, im2=512;
    bool ccomp=0;

    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            if (ccomp==1)
                {
                if (c[1]==1)
                    {
                    im1=Nim;
                    ccomp=0;
                    }
                }
            else
                {
                if (c[0]==1)
                    {
                    im2=Nim;
                    imc=(im1+im2)>>1;
                    ccomp=1;
                    }
                }
            }while (SimActiv());
        }
    };


behavior ACQ_cur(int A, int D, event MCS, bool nRD, bool nWR, in event clk, in bit[1:0] c, in int Ni
m)
    {
    int imc=512;

    CuBus   bus0(A, D, MCS, nRD, nWR);

    Reg_cur     Reg_cur1(bus0, imc);
    Meas_cur    Meas_cur1(clk, c, Nim, imc);

    void main(void)
        {
        par{
            Reg_cur1.main();
            Meas_cur1.main();
            }
        }

    };


//////////////////////////////////////////////////////////////
// mcc_acq_sp.sc                                            //
//                                                          //
//                COM2_CMD_MCC (July 2001)                  //
//////////////////////////////////////////////////////////////

interface IProtSp
    {
    void    Reg_RW(bit[15:0] addr, word24* data);
    };
```

```
channel SpProt(int A, int D, event MCS, bool nRD, bool nWR)
            implements IProtSp
    {
    void Reg_RW(bit[15:0] addr, word24* data)
        {
        t1: wait(MCS);
        if (A != addr) goto t1;

        while (nWR==1 && nRD==1)
            {
            waitfor(1);
            }
        if (nRD==0)
            {
            waitfor(6);
            D=*data;
            while (nRD==0)
                {
                waitfor(1);
                }
            }
        if (nWR==0)
            {
            waitfor(9);
            while (nWR==0)
                {
                waitfor(1);
                }
            *data=D;
            }
        }
    };


interface IBusSp
    {
    void    RegSlave_RW(word24* data, int   addr);
    };


channel SpBus(int A, int D, event MCS, bool nRD, bool nWR)
    implements IBusSp
    {
    SpProt  protocol(A, D, MCS, nRD, nWR);

    void RegSlave_RW(word24* data, int addr)
        {
        protocol.Reg_RW(addr, data);
        }
    };


// Speed acquisition

behavior Reg_sp(IBusSp bus0, in int vit)
    {
    void main(void)
        {
        do
            {
            bus0.RegSlave_RW(&vit, ADDR_wmR);
            }while (SimActiv());
        }
    };

behavior Meas_sp(in event clk, in bit[1:0] S, out int vit)
    {
    int     viti, vit1=65535, vit2=65535;
    bool    sign_v=0;
    bool    flag1=0;
    int     i=0, j=0;

    void main(void)
        {
        do
            {
            wait clk,EndofSim;
            if (flag1==0)
```

```
                {
                if (S[0]==1)
                    {
                    vit2=j;
                    j=0;
                    flag1=1;

                    if (S[1]==0)    sign_v=0;        // Rotation sens//
                    else    sign_v=1;

                    viti=vit1+vit2;
                    if (sign_v==1) viti=-viti;
                    vit=viti;

                    }
                else
                    {
                    j=j+1;
                    }
                }
            else
                {
                if (S[0]==0)
                    {
                    vit1=i;
                    i=0;
                    flag1=0;
                    }
                else
                    {
                    i=i+1;
                    }
                }

            }while (SimActiv());
        }
    };


behavior ACQ_speed(int A, int D, event MCS, bool nRD, bool nWR, in event clk, in bit[1:0] S)
    {
    int vit=131070;

    SpBus   bus0(A, D, MCS, nRD, nWR);

    Reg_sp      Reg_sp1(bus0, vit);
    Meas_sp     Meas_sp1(clk, S, vit);

    void main(void)
        {
        par{
            Reg_sp1.main();
            Meas_sp1.main();
            }
        }
    };


//////////////////////////////////////////////////////////////
// mcc_acq_pwm.sc                                           //
//                                                          //
//                  COM2_CMD_MCC (July 2001)                //
//////////////////////////////////////////////////////////////

interface IProtPWM
    {
    void    Reg_RW(bit[15:0] addr, word24* data);
    };


channel PWMProt(int A, int D, event MCS, bool nRD, bool nWR)
            implements IProtPWM
    {
    void Reg_RW(bit[15:0] addr, word24* data)
        {
        t1: wait(MCS);
        if (A != addr) goto t1;

        while (nWR==1 && nRD==1)
            {
```

```
            waitfor(1);
            }
        if (nRD==0)
            {
            waitfor(6);
            D=*data;
            while (nRD==0)
                {
                waitfor(1);
                }
            }
        if (nWR==0)
            {
            waitfor(9);
            while (nWR==0)
                {
                waitfor(1);
                }
            *data=D;
            }
        }
    };

interface IBusPWM
    {
    void    RegSlave_RW(word24* data, int   addr);
    };


channel PWMBus(int A, int D, event MCS, bool nRD, bool nWR)
    implements IBusPWM
    {
    PWMProt protocol(A, D, MCS, nRD, nWR);

    void RegSlave_RW(word24* data, int addr)
        {
        protocol.Reg_RW(addr, data);
        }
    };

// Control Signals Generator

behavior Reg_alph(IBusPWM bus0, out int alph)
    {
    void main(void)
        {
        do
            {
            bus0.RegSlave_RW(&alph, ADDR_alphR);
            }while (SimActiv());
        }
    };

behavior Command_gen(in event clk, in int alph_in, out bit[1:0] cd, out event clk_i)
    {
    int alph;
    int i=0;
    void main(void)
        {
        alph=alph_in;
        waitfor(CYCLETIME);
        notify(clk_i);
        do
            {
            wait clk,EndofSim;
            i=i+1;
            if (i<alph) cd[0]=1;
            else
                {
                if (i==284)
                    {
                    notify(clk_i);
                    i=0;
                    alph=alph_in;
                    }
                cd[0]=0;
                }
            cd[1]=!cd[0];
            }while (SimActiv());
        }
    };

behavior Command_rap(int A, int D, event MCS, bool nRD, bool nWR, in event clk, out bit[1:0] cd, out
 event clk_i)
    {
    int alph=142;

    PWMBus  bus0(A, D, MCS, nRD, nWR);

    Reg_alph    Reg_alph1(bus0, alph);
    Command_gen Command_gen1(clk, alph, cd, clk_i);

    void main(void)
        {
        par{
            Reg_alph1.main();
            Command_gen1.main();
            }
        }
    };


//////////////////////////////////////////////////////////////
// mcc_control_i.sc                                         //
//                                                          //
//              COM2_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////

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

channel SlaveProt(int A, int D, event MCS, bool nRD, bool nWR)
        implements IProtSlave
    {
    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;
        }
    };

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(int A, int D, event MCS, bool nRD, bool nWR)
    implements IBusSlave
    {
    SlaveProt   protocol(A, D, MCS, nRD, nWR);

    void Slave_sendW(word24 data, int addr)
        {
//      intC.send();
        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)
        {
//      intC.send();
        *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);
            }
        }
    };

// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

// Current controller Clock

behavior Clock_comi(event clk)
    {
    void main(void)
        {
        do
            {
            waitfor(CYCLETIME_comi);
            notify(clk);
            }while (SimActiv());
        }
    };


behavior Exchange_i(IBusSlave bus0, in int alph, out int imc, out int irefn)
    {
    void main(void)
        {
        bus0.Slave_sendW(alph, ADDR_alph);
        bus0.Slave_recvW(&imc, ADDR_im);
        bus0.Slave_recvW(&irefn, ADDR_irefn);
        }
    };


// Current controller algorithm

behavior CurCTL(int A, int D, event MCS, bool nRD, bool nWR, event clk_comi)
    {
    float epsi, epsi1, integi, vmoy, alpha;
    float im, iref;
    int alph=142, imc=512;
    int irefn=32768;
    float irefnf=0.0;
    float imcf=0.0;

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

    Exchange_i  Exchange_i1(bus0, alph, imc, irefn);

    void main(void)
        {
        epsi=epsi1=integi=vmoy=alpha=0.0;

        do
            {
            wait clk_comi,EndofSim;

            Exchange_i1.main();
            imcf=imc;
            im=2*imcf*imax/p1 - imax;
            irefnf=irefn;
            iref=2*irefnf*imax/p2 - imax;

            epsi=iref-im;
            integi=integi+((epsi+epsi1)/2)*te;
            vmoy=kc*epsi+(kc*integi)/tc;
            alpha=vmoy/(2*Ve)+0.5;
            if (alpha>0.95) alpha=0.95;
            if (alpha<0.06) alpha=0.06;
            alph=(int)(alpha*284);
            epsi1=epsi;

            }while (SimActiv());
        }
    };


//////////////////////////////////////////////////////////////
// mcc_control_w.sc                                         //
//                                                          //
//              COM2_CMD_MCC (July 2001)                    //
//////////////////////////////////////////////////////////////

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

channel ProtMaster(int A, int D, event MCS, bool nRD, bool nWR)
    implements IProtMaster
    {
    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);
        }
     };


 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(int A, int D, event MCS, bool nRD, bool nWR)
     implements IBusMaster
     {
```

```
    ProtMaster  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);
            }
        }
    };

// CONTROL DEVICE SPECIFICATIONS
/////////////////////////////////

//Speed Controller Clock

behavior Clock_comv(event clk)
    {
    void main(void)
        {
//      waitfor(CYCLETIME);
        notify(clk);
        do
            {
            waitfor(CYCLETIME_comv);
            notify(clk);
            }while (SimActiv());
        }
    };


behavior Exchange_w(IBusMaster bus0, in int irefn)
    {
    int alph, imc;

    void main(void)
        {
        bus0.Master_recvW(&alph, ADDR_alph);

        bus0.Master_sendW(alph, ADDR_alphR);
        bus0.Master_recvW(&imc, ADDR_imR);

        bus0.Master_sendW(imc, ADDR_im);
        bus0.Master_sendW(irefn, ADDR_irefn);
        }
    };


 // Speed Controller Algorithm

 behavior SpeCTL(IBusMaster bus0, event clk_comv,in float vref, out int irefn)
     {
     float epsv, epsv1, integv, v, iref=0.0;
     int vit;
     float vitf;

     void main(void)
```

```
        {
        epsv=epsv1=integv=0.0;
        do
            {
            wait clk_comv,EndofSim;

            bus0.Master_recvW(&vit, ADDR_wmR);

            vitf=vit;
            v=4*pi*1e6/(vitf*n_cio);
            epsv = vref - v;
            integv = integv +((epsv+epsv1)/2)*tev;
            iref=kv*epsv+(kv*integv)/tv;
            epsv1=epsv;

            if (iref>13.0) iref=13.0;
            if (iref<-13.0) iref=-13.0;

            irefn=(int)((iref+imax)*p2/(2*imax));
            }while (SimActiv());

        }
    };

behavior Sw_w(int A, int D, event MCS, bool nRD, bool nWR, in event clk_i, event clk_comv,in float v
ref)
    {
    int irefn=32768;

    MasterBus   bus0(A, D, MCS, nRD, nWR);

    Exchange_w  Exchange_w1(bus0, irefn);
    SpeCTL      SpeCTL1(bus0, clk_comv,vref, irefn);

    void main(void)
        {
        try{
            SpeCTL1.main();
            }
        interrupt(clk_i)
            {
            Exchange_w1.main();
            }
        }
    };
```