UNIVERSITAT POLITÈCNICA DE CATALUNYA

ESCOLA TÈCNICA SUPERIOR D'ENGINYERS
INDUSTRIALS DE BARCELONA

Doctoral Thesis

MUSS: A CONTRIBUTION TO THE STRUCTURAL ANALYSIS OF CONTINUOUS SYSTEM SIMULATION LANGUAGES

December 1987

Antoni Guasch i Petit
Institut de Cibernètica

Chapter 5

Conclusions and future research

"Expert systems are programs that emulate human expertise in a specific domain by applying techniques of logical inferences to a knowledge base. The knowledge base stores information about how to carry a task including facts, uncertain and heuristic knowledge, as well as non-algorithmic inference procedures. When applying the high potential of these features, expert systems can be used more efficient than conventional software to enhance the acceptance and effectiveness of modelling and of simulation."

[Lehmann87a]

5.1 Conclusions

The outcomes of this research work can be separated in three groups: in the first, the *main abstract contributions* are outlined, in the second, the *present implementation state* of the *MUSS* system is described, whereas in the third some *results* are outlined.

5.1.1 Abstract contributions

The main end results towards the enhancement of the state of the art in general purpose continuous time simulation systems are:

- **Architectural aspects:**
  - The proposed architecture of the *MUSS* language is coherent with the natural division of the physical system into subsystems trough the minimal but sufficient number of blocks which have been defined.
  - The design of the *MUSS* strengths a modularity converging to the object oriented language concept.
- **Algorithmic proposals:**
  - The proposed *submodel digraph* concept is the base supporting the increase in the robustness of the submodel code in a way so far not achieved in CSSL-like languages.
  - The *segmentation concept* contributes to the reliability of the simulator software.

    Splitting up the submodel code into the initial, ODE and discontinuous segments is consistent with the functional tasks involved in a simulation run.
  - *Isolated preprocessing* of the submodel, experiment and study blocks is allowed. It has been achieved, avoiding restrictions, by the division of the ODE segment into the state, algebraic and derivative subsegments.

    An additional advantage of the proposed subsegmentation method is that the algebraic chains are detached from the rest of the code into the algebraic segments. Therefore, the subsegmentation also opens the door to implement algorithms for the solution of implicit equations in an elegant manner though the algebraic loops will be spread on a set of submodels hierarchically related [Holiander81a] [Troch86a].
- **Structural aspects:**
  - The *definition digraph* links the submodel data structures of the submodel and experiment blocks. It is the bed structure used by the *MCL executive* for handling the simulation environment.
  - The proposed *MUSS Command Language* allows the user to communicate with the simulation environment.

5.1.2 Present implementation state of the MUSS system

To achieve a production code has never been considered an objective into the thesis goals. Nevertheless some insight on the behaviour of the solutions proposed was necessary. As a result, some pieces of code exist:

- **Preprocessor**

  The architectural design has been completed and the following modules coded:

  - Lexical analyzer.
  - Syntax analyzer.
  - Table manager.
  - Sorting and segmentation procedures of submodel digraphs associated to continuous time submodels without time or state events [Guasch85d].

- **Simulation environment**

  A prototype of the simulation environment has been completed and successfully tested. Its core embraces the following main modules:

  - The *MCL interpreter* whose grammar is fully described in appendix B, has the responsibility for understanding users' commands. It has been coded using automatic compiler production techniques.
  - The *MCL executive* which embodies the set of algorithms achieving the users' commands. It includes those processes involved in study, experiment and model instantiation, activation, execution and deletion. It also allocates the dynamic memory necessary for the integration package. Most of its procedures extract the information they need to perform from the definition digraph.
  - The *LSODAR integration and root finder* package [Hindmarsh82a]. It is written in Fortran. Some minor changes have been introduced to properly interface it with the *MUSS* environment.
  - The *LSODAR front-end* routine. Its main tasks are:
    - ◇ To set the base addresses of the working memory areas. These addresses depend on the experiment instance being executed.
    - ◇ To set set the discontinuous states associated to the discontinuous functions at initial time and at each event occurrence.
    - ◇ To invoke I/O tasks at each communication point.
    - ◇ To call the *LSODAR* package with the proper input arguments.

5.1.3 Results

The proposed analysis, segmentation and sorting algorithms described in chapter 3 have been successfully implemented in the preprocessor and applied to:

- A complex electrical network model [Riera85a] which has seventeen submodels and whose hierarchical structure has four levels.
- The switched-mode power regulator (see appendix C).

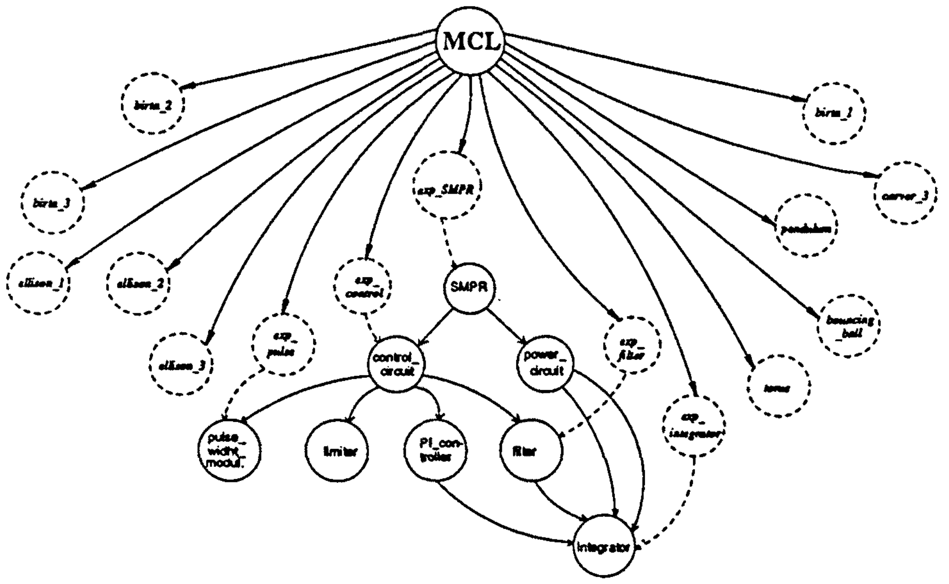

Figure 5.1: *Subset of experiments which have been used for testing the simulation environment.*

To get an insight about the *overall system performance* some study cases have been selected and lead to execution according with the following procedure:

- The *C* target codes have been generated from the *MUSS* source code using the existing modules of the preprocessor and emulating the outputs of those not yet implemented.
- The experiment interface routines (see code 4.13 in page 130) have been emulated although its implementation is straightforward.
- The *C* object codes along with the associated experiment interface routines have been linked with the *MCL* interpreter and executive to generate a simulation environment.

The cases chosen have been put together in the environment represented in figure 5.1 and they can be assembled in two groups:

- First:
  - *bouncing_ball, torus, simple_switch* and *pendulum* [Thompson85a].
  - *ellison_1, ellison_2* and *ellison_3* [Ellison81a].
  - *carver_3* [Carver75a].
  - *birta_1, birta_2* and *birta_3* [Birta85a].

  They have been successfully used for testing the discontinuity handling mechanisms.
- Second:
  - *exp_integrator, exp_filter, exp_pulse, exp_control* and *exp_SMPR*.

  They have been successfully used for testing the management and execution of hierarchical models (*MCL* interpreter and *MCL* executive).

5.2 Future research

Part of the planned immediate research is concerned with a further development of the submodel digraph concept and that of the declarative block definition to allow the inclusion and analysis of new sentences (`case`, `do while`) in the continuous submodel. This involves the definition of new vertices such *case* or *do while* executable vertices.

In addition, a first prototype of the *MUSS* system will be coded and tested although the aspects related to distributed and real-time simulation will be left aside.

The research on simulation environments is an interdisciplinary task involving a large number of computer science branches. The future research concentrates on the topics reported right after and some of them will be hopefully developed in conjunction with other research teams:

- **Artificial intelligence (AI):**

  The role of Artificial Intelligence (in particular Expert Systems) in the simulation domain is becoming important. This can be observed looking at the ratio of

AI related papers presented Simulation Congresses, Symposiums or Multiconferences. Some figures in recent events are:

- "2nd European Simulation Congress", September 1986. Antwerp, Belgium. *(10 AI related papers).*
- "AI, Expert Systems and Language in Modelling and Simulation", IMACS International Symposium, June 1987. Barcelona. *(38 AI related papers).*
- "1988 SCS Multiconference", February 1988. San Diego, California. *(63 AI related papers).*

Three different categories of applications combining expert systems and simulation can be distinguished [Lehmann87a]:

- Simulation models embedded in expert systems.
- Expert systems as integral part of a simulation model.
- Expert system as supporting framework for a goal-directed application of modelling tools.

The specific objectives will be selected towards the goal of assistance to the users in the modelling, simulation, verification, validation and documentation phases involved in a simulation study.

In addition, an expert system shell should be chosen and some aspects which have to be analyzed beforehand are: the interface between the expert system and the simulation model, the ability of the expert system for working with the different results produced by a set of simulation runs and the degree of flexibility and adaptability of the expert system.

- **Real-time features:**

  The stress in this thesis is on the software robustness and reliability rather than in the time requirements because of the availability of fast and low-cost computers.

  The *MUSS* language, as well as the majority of the general purpose simulation languages, can not support hardware in the loop real-time simulation were time is a critical constraint for the scenario (aerospace, flight simulators) [Crosbie82d]. However, the type of real-time problems we are concentrated in is less time critical because the physical subsystem only exchanges data with the modelled subsystem at time events. This may be the case of a controller wich governs a simulated subsystem.

  In the *MUSS* architecture, the *sampled submodel that interfaces with a hardware subsystem* has already been defined. Nevertheless, a further research study has to be made to know if the proposed software concept is able to synchronize in real-time. This depends on two main aspects,

- Fitness of the simulation system software which embraces:
  - ◇ The selection of the appropriate integration and root finder algorithms.
  - ◇ The time overhead inherent to the ODE segmentation. New C code optimizers may remove this question performing in-line expansions. When the optimizer expands a function in-line, it eliminates the function call and therefore the overhead [Wolverton86a].
  - ◇ The memory management mechanisms which should be analyzed in more detail. In fact, to allot memory dynamically to all the submodel variables is time expensive. A better strategy could be to select the variables with which really need to be allocated dynamically.
- Complexity of the model:
  - ◇ Given a fixed real-time simulation environment, it corresponds to the user to modify or simplify his simulated subsystem and the communication with the hardware subsystem to satisfy the real-time requirements without loosing the validity of the model.

• **Simulation of combined models:**

The term combined system (model) is defined by Prof. Cellier as follows:

> "*Combined systems are described, either during the whole period under investigation or during a part of it, by a fixed or variable set of differential equations where at least one state variable or one state derivative is not continuous over a simulation run*" [Cellier79a].

Using this definition, the *MUSS* system can be classified as a combined simulation system. However, even though the discrete features are restricted to discontinuities in the state variables (implicit solutions of the ODE), the architecture and the structure of the *MUSS* system allow the expansion to a truly combined simulation system.

Future research on combined simulation might include the architecture and grammar definition of the *MUSS* discrete subprocesses, the inclusion of discrete features in the simulation environment and the interface between discrete and continuous subprocesses (present hierarchical models).

• **Object oriented languages:**

The object-oriented programming methodology allows easy and conceptually clean development of special purpose application packages [Kreutzer86a] as well as software reusability.

Objects are programming structures which contain data structures and access routines which perform the allowed operations on those data structures [Zenor87a]. In the context of simulation programming objects are often referred as entities.

In *C*, the object-oriented programming style can be emulated even though it is not straightforward. Therefore, the object-oriented programming methodologies will be investigated and if needed, parts of the *MUSS* system could be coded in a language which supports object-oriented programming styles —MODULA (module) [Wirth83a], ADA (package) [Ledgard81a], SIMULA (class) [Dahl66a]—.

- **Distributed parameters submodels:**

  In the *MUSS* architecture definition, the *distributed parameters submodel* has already been defined. However, a flexible syntax is difficult to achieve. This may constitute an additional research topic.

  Distributed parameter systems are characterized by partial differential equations. These equations have their own solution methods [Karplus82b]. In, the *MUSS* system, the implementation of the method of lines is straightforward, specific algorithms as Galerkin's procedure for parabolic equations can be attached and packages like *FORSIM IV* [Carver79a] can be appended to LSODAR.

Appendix A

Metalanguage used to define MUSS and MCL grammars

The metalanguage used to specify *MUSS* and *MCL* grammars follows the guidelines presented by Johnson [Johnson75a] and implemented in the *YACC* compiler-compiler. *YACC* allows the users to specify an unambiguous grammara or an ambiguous one along with precedence and associative information about operators.

Although the most widely used notation to specify formal languages seems to be the *Extended Backus-Naur Formalism* (EBNF) [Wirth83a], we rather use the Johnson syntax because grammar rules specified with it can be directly input to the YACC compiler-compiler.

The rules of the metalanguage are:

- Terminal names —*tokens*— are enclosed within *quotation marks* or are written in *UPPER CASE LETTERS.*
- Non terminal names —*grammar rules*— are written in *lower case letters*
- Names are of arbitrary length, and alphanumeric characters and underscore "_" can be used in the string. The first character has to be alphabetic.
- A *space* or a set of spaces is the delimiter between two syntactic units.
- A *semicolon* indicates the end of a rule.
- A *colon* separates the left and right side of any rule.
- The exclusive or is represented by the symbol "|".

- One or more occurrences of a syntactic unit are defined by left recursion, which is encouraged by *YACC*, as shown in the following example,

```
list
        : item
        | list item
        ;
```

- Zero, one or more occurrences of a syntactic unit are represented as follows,

```
optional_list
        : /* empty */
        | list
        ;

list
        : item
        | list item
        ;
```

  where `/*empty*/` is just a comment, not a syntactic unit.

MUSS and *MCL* grammar once defined according the above rules can be directly used as input to *YACC* in order to generate the corresponding syntactic analyzers. The syntactic analyzers call a lower level input routine *—lexical analyzer—* to get the basic tokens.

The necessary lexical analyzers for both grammars have been generated by *LEX* [Lesk75a]. *LEX* accepts a set of regular expressions and produces code in C language which recognizes them.

Appendix B

MUSS command language (MCL)

The Muss Command Language (MCL) is the language designed to communicate the simulation users with the *MUSS* user defined interactive simulation environment.

B.1 Grammar

The design of the *MCL* commands has been influenced by the friendly syntax of the VAX/VMS *DCL* commands [Dec84a]. As a result, the specification of the *MCL* grammar has the flavour of the general syntax of the *DCL*,

```
command
        : command_name option qualifiers parameters
        ;

option
        : IDENTIFIER
        ;

qualifiers
        : /* empty */
        | qualifier_list
        ;

qualifier_list
        : qualifier
        | qualifier_list qualifier
        ;
```

```
qualifier
        : '/' IDENTIFIER qualifier_extension
        ;
```

The *MCL* grammar presented in this chapter is a subset of the actual grammar used in the implementation. The simplifications introduced are concerned with:

- Error recovery procedures.
- Error report generation.

Both aspects are very important and must be taken into account when designing a grammar [Aho77a] but, they are not introduced here because the aim is to focus the *MCL* commands.

B.1.1 MCL

MCL can be viewed as a set of commands. There are not syntactic precedence restrictions in the order of execution of the commands but, inconsistency errors will be reported if the whole semantic is incorrect. For example, to command the execution of a non activated experiment is a cause of error because only active experiments or studies can be executed.

```
mcl
        : command_set
        ;
```

B.1.2 Command_set

At present, a minimal set of commands has been implemented for communicating users with the *MUSS* environment. To Include new commands is a rather easy task because of the great modularity and flexibility of the *YACC*. Three types of commands can be emphasized among the set of commands still not implemented:

- Commands to control an alphanumeric and graphic postprocessor.
- Commands to call a *menu driven module* which would help inexperienced users.

- Commands to control the execution flow into the *MCL* command files (i.e. GOTO, IF...ENDIF).

```
command_set
        : command_newline
        | command_set command_newline
        ;

command_newline
        : command      '\n'
        | /* empty */ '\n'
        ;

command
        : calculator
        | comment_line
        | control
        | create
        | do
        | edit
        | exit
        | external_file
        | help
        | parameter
        | prepare
        | print
        | remove
        | scope
        | set
        | show
        | type
        ;
```

Calculator

OPER command is used to ask for performing basic mathematical operations.

Grammar rules written below are the specification of a small desk-top calculator. *YACC* [Johnson75a] precedence rules are used to solve ambiguities.

```
calculator
        : OPER expression
        ;

expression
        :   '(' expression ')'
        |   expression '+' expression
        |   expression '-' expression
        |   expression '*' expression
        |   expression '/' expression
        |   '-' expression
        |   number
        |   SIN '(' expression ')'
        |   COS '(' expression ')'
        |   TAN '(' expression ')'
        |   SQRT '(' expression ')'
        |   LOG '(' expression ')'
        |   LN '(' expression ')'
        |   ACOS '(' expression ')'
        |   ASIN '(' expression ')'
        |   ATAN '(' expression ')'
        |   ABS '(' expression ')'
        |   EXP '(' expression ')'
        |   expression POT expression
        ;

number
        : DECIMAL_NUMBER
        | INTEGER_NUMBER
        ;
```

Comment_line

Comment lines are suited for documenting *MCL* files.

```
comment_line
        :  '!'  STRING
        ;
```

Control

Command used to assign values to system parameters.

```
control
        :  CONTROL control_list
        ;

control_list
        :  control_element
        |  control_list ',' control_element
        ;

control_element
        :  wild_card_identifier '=' expression
        ;
```

Create

Creates experiment or study instances, Thus, allocates data storage for studies, experiments and models and performs the static initialization.

An experiment or a study may have more than one instance at a time, this facility may look superfluous when working in continuous simulation but, this option has been introduced thinking about a future expansion of the language to the simulation of *combined models.*

```
create
        : CREATE process_specification
        ;
```

Do

Indicates to the system which study or experiment (created) instance has to be executed.

```
do
        : DO qualifiers process_specification
        ;
```

• Do qualifiers, by now, are:

/STATISTICS : Used to get statistics about the simulation runs. CPU time, number of calls to F and number of calls to G are some of the parameters whose values will be displayed after each simulation run.

Edit

MCL editor is an easy to use, line-oriented and *MCL* command-oriented editing module.

The most important features of this editor are:

- It performs on-line syntactic analysis of the *MCL* commands written by the user. A simple way to specify this has been achieved with the grammar rules enhanced by the dashed box,

  When in editing mode and after given an *insert* edit-command, semantic actions associated to grammar rules are disabled —except *insert* semantic actions— to inhibit interactive execution of edited commands. The insert mode of the editor is cancelled by means of an empty command.

- It is called within the *MUSS* simulation environment. Thus, activated studies and experiments remain unchanged (frozen) in the system while editing.

Up to now, five basic editing commands have been implemented: *exit, quit, insert, delete, type*. To make the editor more powerful, the set of editing commands should be expanded to: *search, copy, move, substitute* and *write*.

The internal data structure used to implement the editing buffer is a *linked list*. This internal representation has been chosen because editing operations are straightforward and easy to implement with it.

```
edit      : edit_begin edit_command_set edit_end
          ;

edit_begin
          : EDIT mcl_file_specification      '\n'
          | EDIT '\n' mcl_file_specification '\n'
          ;

mcl_file_name
          : file_specification
          ;

edit_end
          : EXIT
          | QUIT
          ;

edit_command_set
          : /* empty */
          | edit_command_newline
          | edit_command_set edit_command_newline
          ;

edit_command_newline
          : edit_command '\n'
          | /* empty */  '\n'
```

```
        ;

edit_command
        : insert
        | delete
        | type
        | renumerate
        ;

-------------------------------------------
| insert                                  |
|       : I  '\n' line_list               |
|       | I number  '\n' line_list        |
|       ;                                 |
|                                         |
| line_list                               |
|       : /*empty*/                       |
|       | line_newline                    |
|       | line_list line_newline          |
|       ;                                 |
|                                         |
| line_newline                            |
|       : command  '\n'                   |
|       ;                                 |
-------------------------------------------

delete
        : D
        | D number
        | D number ':' number
        ;

type
        : T W
        | T number
        | T number ':' number
        ;

renumerate
        : R number  '\n'
        ;
```

Exit

When this command is issued at the level of the "muss in:" prompt it causes the orderly termination of the session. The screen is restored to its original state and control is passed to the operating system.

When EXIT is declared into a *MCL* command file then it just forces the exit of the file and control is passed to *MCL* or to a calling *MCL* command file.

```
exit
        : EXIT
        ;
```

External_file

Intended to drive the *MCL* command interpreter to execute a specified command file.

MCL command files can be created using the *MCL* editor by users in the *MUSS* environment or utilizing the general purpose editors of the computing system by users in the operating system environment.

Inside *MCL* command files other *MCL* command files can be invoked using the "@" command. Default file type is *mcl*.

```
external_file
        :  '@' qualifiers file_specification
        ;
```

• *External_file* qualifiers are:

/LOG : Writes the succesive commands being executed from the specified file.

Help

An *MCL* help library has been created using the VAX/VMS help utilities [Dec84a]. The library is organized as a tree.

Library access is controlled by VAX procedures. The main problem related with VAX/VMS library utilities is that they are not portable to non DEC computers but we think that this is an acceptable restriction for the *MUSS* prototype.

```
help
        : HELP
        ;
```

Parameter

Command used to assign values to the parameters of the study, experiment or submodel blocks.

One semantic action associated to the command consists on checking that a `wild-_card_identifier` is the specification of a single parameter. That is, this command is illegal if `wild_card_identifier` matches more than one parameter (see page 169).

```
parameter
        : PARAMETER parameter_list
        ;

parameter_list
        : parameter_element
        | parameter_list ',' parameter_element
        ;

parameter_element
        : block_specification wild_card_identifier '=' expression
        ;
```

Prepare

Specifies those variables to be recorded at each communication interval.

```
prepare
        : PREPARE qualifiers optional_prepare_list
        ;

optional_prepare_list
        : /* empty */
        | prepare_list
        ;

prepare_list
        : prepare_element
        | prepare_list ',' prepare_element
        ;

prepare_element
        : block_specification wild_card_identifier
        ;
```

• *Prepare* qualifiers are:

/DELTA=(expression) : Sets `prepare` step size.

/RESET : Resets `prepare_list` buffer.

/OUTPUT=[(file_name)] : Directs the information in `prepare_list` towards the specified file instead of the standard SIM.PRE file. File type is always *.PRE*; users should not specify the file type.

Print

This command should be used for specifying the variables whose values are to be printed at specified time intervals.

```
print
        : PRINT qualifiers optional_print_list
        ;

optional_print_list
        : /* empty */
        | print_list
        ;

print_list
        : print_element
        | print_list ',' print_element
        ;

print_element
        : block_specification wild_card_identifier
        ;
```

• *Print* qualifiers are:

/DELTA=(expression) : Sets `print` step size.

/RESET : Resets `print_list` buffer.

/OUTPUT[=(file_name)] : Forces the information in `print_list` to be output to the specified file instead of the current standard output device. If the file name field is empty, `print` assumes the default (SIM.DAT).

Remove

Removes experiments or study instances. The main involved procedure is to deallocate memory to the removed items in the defined environment.

```
remove
        : REMOVE process_specification
        ;
```

Scope

This command is used for specifying the variables whose values are to be plotted versus and independent variable at defined time intervals.

```
scope
        : SCOPE qualifiers optional_scope_list
        ;

optional_scope_list
        : /* empty */
        | scope_list
        ;

scope_list
        : scope_element
        | scope_list ',' scope_element
        ;

scope_element
        : block_specification wild_card_identifier
        ;
```

- *Scope* qualifiers are:

  /DELTA=(expression) : Sets scope step size.

  /RESET : Resets scope_list buffer.

  /OUTPUT[=(file_name)] : Defines the specified file as output of the variables in scope_list instead of the current standard output device. If the file name field is empty, scope assumes by default (SIM.SCO).

/ORIGIN=(expression,expression) : Sets the origin of the X and Y axes in the chart.

/SCALE=(expression,expression) : Sets the scale factors associated to the X and Y axes.

/INDEPENDENT=(variable) : Sets the plotting independent variable. The independent variable of the model is assumed by default.

/ERASE : Erases the drawings in the screen before starting a new one.

Set

Defines or redefines, for the current session, characteristics associated with the blocks, the files or the devices owned by the *MUSS* environment.

```
set
        : SET set_options
        ;

set_options
        : set_screen
        | set_block
        | set_trace
        ;

set_screen
        : SCREEN qualifiers
        ;

set_block
        : BLOCK block_specification
        ;

set_trace
        : TRACE qualifiers
        ;
```

• `Set_screen` is used to erase the screen and to switch the screen mode between graphics and alphanumeric mode. `Set_screen` qualifiers are:

/ERASE : Erases the screen.

/GRAPHICS : Sets screen into graphic mode.

/LISTING : Sets screen into alphanumeric mode.

`set_block` sets the default block which can be directly accessed from *MCL* to get information of it. A block can be an experiment, a study or a submodel.

- `Set_trace` actives established tracepoints. `Set_trace` qualifiers are :

  /F_G_CALLS : Trace F and G calls.

  /STATE_EVENTS : Trace state events.

  /TIME_EVENTS : Trace time events.

  /DISCONTINUOUS : Trace discontinuous function values. Discontinuous vector is written after each call to G.

  /STATE_VECTOR : Trace state vector.

  /DERIVATIVE : Trace derivative vector.

  /ALL : Includes all above qualifiers.

  /RESET : Resets trace qualifiers included in the same `set_trace` command.

Show

Displays information about the present status of the blocks in the *MUSS* environment.

```
show
        : SHOW show_options
        ;

show_options
        : show_model
        | show_experiment
        | show_study
        | show_block
        ;

show_model
        : MODEL qualifiers submodel_level
        ;

show_experiment
        : EXPERIMENT qualifiers experiment
        ;

show_study
        : STUDY qualifiers study
        ;

show_block
        : BLOCK qualifiers block_specification
        ;
```

• `Show_model` displays information about models present in the user defined simulation environment. A model is a set of hierarchical related submodels. `Show_model` qualifiers are:

/SUBMODEL : The names of the submodels are listed.

/EXPERIMENT : Lists the names of the experiments defined over models or submodels (qualifier /SUBMODEL).

• `Show_experiment` displays information about experiments present in the system. `Show_experiment` qualifiers are:

/ACTIVE : To display only active experiments names.

/FULL : The main characteristics of experiments selected in the command and present in the user defined simulation environment are described.

• `Show_study` displays information about studies present in the system. `Show_study` qualifiers are:

/ACTIVE : Display only active studies.

• `Show_block` displays information about selected blocks. `Show_block` qualifiers are:

/SYMBOL : Display all block symbols.

/STATE : Display all state variables.

/AUXILIARY : Display all auxiliary variables.

/INTEGER : Display all integer variables.

/LOGICAL : Display all logical variables.

/CONSTANT : Display all constants.

/PARAMETER : Display all parameters.

/EXPERIMENT : If the selected block is a submodel, then display associated experiments.

/FULL : Display the main characteristics of the submodel.

/SUBMODEL : If the selected block is an experiment or a submodel, display the selected information for all lower level submodels.

Type

Displays the contents of files or the value of a request set of variables

```
type
        : TYPE type_options
        ;

type_options
        : type_variable
        | type_file
        | type_system
        ;

type_variable
        : VARIABLE qualifiers type_variable_list
        ;

type_variable_list
        : type_variable_element
        | type_variable_list ',' type_variable_element
        ;

type_variable_element
        : block_specification wild_card_identifier
        ;


type_file
        : FILE file_specification_list
        ;

type_system
        : SYSTEM type_system_list
        ;

type_system_list
        : type_system_element
        | type_system_list ',' type_system_element
        ;
```

```
type_system_element
        : wild_card_identifier
        ;
```

• Type_file displays the contents of a group of files.

• type_system displays selected system variables.

• type_variable displays the value of a set of variables. Type_variable qualifiers are:

/ALL : Type the value of all the variables maching a variable specification.

/STATE : Type the value of all the state variables.

/DERIVATIVE : Type the value of all the derivative variables.

/DISCONTINUOUS : Type the value of all the discontinuous functions.

/AUXILIARY : Type the value of the auxiliary variables.

/INTEGER : Type the value of all the integer variables.

/LOGICAL : Type the value of all the logical variables.

/CONSTANT : Type the value of all the constants.

/PARAMETER : Type the value of all the parameters.

/SUBMODEL : If the selected block is an experiment or a submodel, display the selected information for all lower level submodels.

B.1.3 Auxiliary grammar rules

File_specification_list

A file_specification has two fields, file_name and file_type, separated by a period.

```
file_specification_list
        : file_specification
        | file_specification_list ',' file_specification
        ;
```

```
file_specification
        : file_name file_extension
        ;

file_name
        : IDENTIFIER
        ;

file_extension
        : /* empty */
        | '.' IDENTIFIER
        ;
```

Process_specification

The `process_specification` identifies an experiment or study instance.

```
process_specification
        : study_level
        | experiment_level
        ;

study_level
        : study process_copy
        ;

study
        : ':' ':' wild_card_identifier
        ;

experiment_level
        : experiment process_copy
        ;

experiment
        : ':' wild_card_identifier
        ;

process_copy
        : /* empty */
        | '(' wild_card_identifier ')'
        ;
```

Block specification

The following grammar rules specify a block. Three types of blocks are defined, studies, experiments and submodels. Submodel specification may be absolute or relative to other submodels in the same model.

An example would illustrate both alternatives. Figure B.1 represents the model and the experiments structure of the case study presented in chapter C. Some absolute specification are:

- :exp_smpr(test1)

  Specifies experiment block *exp_smpr* of instance "test1" of experiment exp_smpr.
- :exp_smpr(test2)[SMPR.control] Specifies submodel block *control* called from submodel SMPR in the instance "test2" of experiment exp_smpr.

and, if the default block is:

:exp_control[control.limiter]

some relative specifications are:

[.-]

Specifies submodel block *control.*

[.-.PI_controller]

Specifies submodel block *PI_controller.*

```
block_specification
        : relative_block_specification
        | absolute_block_specification
        ;

relative_block_specification
        : '[' '.' level_list ']'
        | /* empty */
        ;
```

```
level_list
        : level_element
        | level_list '.' level_element
        ;
```

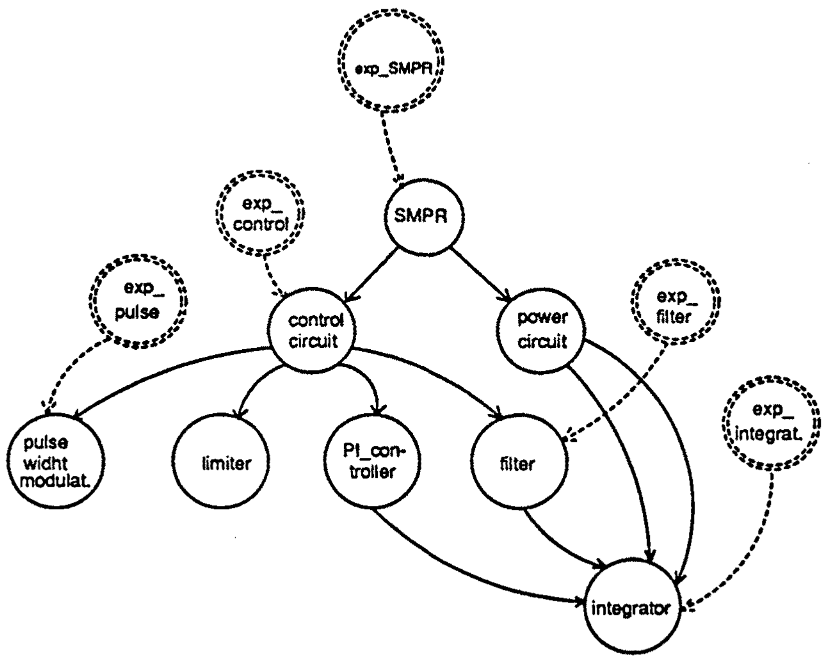

Figure B.1: *SMPR experiments and model structure. To specify a block, let say, filter submodel block, is not as easy as to state "Look at filter submodel block" because several instances may be present at a given time. It must be precisely specified which filter instance is wanted to look at, otherwise, only static information would be available (see subsection 4.2.2).*

```
level_element
        : '-'
        | submodel_name
        ;

submodel_name
        : wild_card_identifier
        ;
```

```
absolute_block_specification
        : study_level optional_experiment_and_submodel_level
        | experiment_and_submodel_level
        ;

optional_experiment_and_submodel_level
        : /* empty */
        | experiment_and_submodel_level
        ;

experiment_and_submodel_level
        : experiment_level optional_submodel_level
        ;

optional_submodel_level
        : /* empty */
        | submodel_level
        ;

submodel_level
        | '[' level_list ']'
        ;
```

Qualifiers

Only identifiers that match allowed qualifiers are accepted. Selected qualifiers depend on the command and option given.

```
qualifiers
        : /* empty */
        | qualifier_list
        ;

qualifier_list
        : qualifier
        | qualifier_list qualifier
        ;

qualifier
        : '/' IDENTIFIER qualifier_extension
        ;
```

Qualifier_extension

```
qualifier_extension
        : /* empty */
        | '=' '(' extension ')'
        ;

extension
        : constant_list_extension
        | variable_list_extension
        ;

variable_list_extension
        : variable_element_extension
        | variable_list_extension ',' variable_element_extension
        ;

variable_element_extension
        : block_specification wild_card_identifier
        ;

constant_list_extension
        : constant_element_extension
        | constant_list_extension ',' constant_element_extension
        ;

constant_element_extension
        : expression
        ;
```

Wild_card_identifier

The wild card character "*" is a symbol that users can utilize with many *MCL* commands. It is suited to address variables or blocks in a shorten form or to match sets.

For example, let suppose that a block has been set, the command to type the value of all its variables is,

muss in > *type variables* *

and the command to type all variables starting with the latter *b* is,

muss in > *type variables b**

In the following example the wild card character is used to specify a model. Instead of the command,

muss ln > *show model [spain_france_interconnected_network]*

we can write,

muss ln > *show model [spain_fran*]*

note that,

muss ln > *show model [*]*

specifies all models present in the user defined interactive simulation environment.

```
wild_card_identifier
        : IDENTIFIER
        | IDENTIFIER '*'
        | '*'
        ;
```

Appendix C

Case study: switched-mode power regulator

This case study is taken from the *ESL* Software User Manual [Hay85a]. The main objective on analyzing it is to emphasize the top-down modelling and the bottom-up coding and testing approach which is strongly supported by the *MUSS* system.

A switched-mode power regulator (SMPR) takes as input an un-regulated power supply voltage (V_s) and produces a stabilized output voltage (V_0) with minimal power loss. The level of the output is determined by a reference voltage (V_{ref}). A high yield is achieved using a high frequency switch with a controlled mark-space ratio, to 'chop' the input voltage which is then presented to a filter circuit. Accurate control of the output voltage is achieved by a control system which generates a switching waveform with a variable mark-space ratio.

The various components which form the model of the switched-mode power regulator circuit are discussed and then the *MUSS* submodels code which represent the components, the posterior analysis performed by the preprocessor and the experiments to test their behavior are presented.

C.1 Circuit Elements, Top-Down Modelling

C.1.1 SMPR circuit

The SMPR circuit is composed by the two modules showed in figure C.1 , the power circuit and the control circuit.

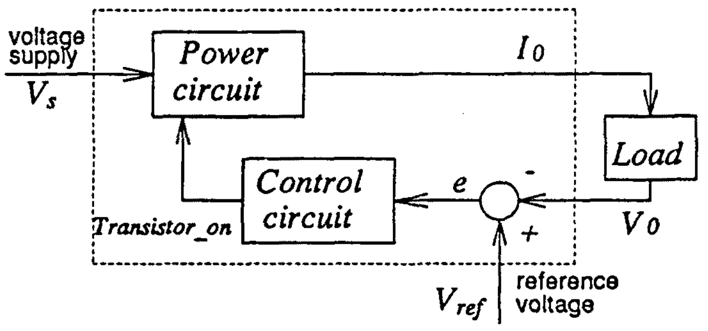

Figure C.1: *Switched-mode power regulator circuit (SMPR).*

C.1.2 Power circuit

The power circuit (figure C.2) consists of a switch (transistor plus diode), an inductor/capacitor (*LC*) filter stage and the load resistance R_0. The logical output of the control circuit, drives the switch which connects the supply voltage (V_s) to the *LC* filter.

The SMPR has two modes of operation depending on the current flow (I_l) into the *LC* filter, and are known as the discontinuous and continuous mode. In the discontinuous mode I_l returns to zero during each period of the pulse-width modulator whereas, in the continuous mode it does not.

There may be three states during each period of the pulse-width modulator:

1. When the transistor is *ON* it effectively connects the supply voltage to the power circuit, i.e:

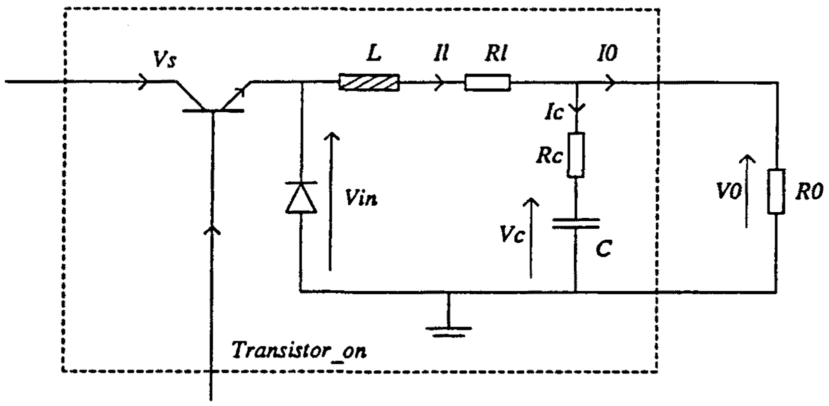

Figure C.2: *Power circuit.*

$$V_{in} = V_s$$

2. While the transistor is *ON*, the inductance stores energy. When the transistor switches *OFF*, this stored energy will be transferred by I_l continuing to flow through the 'free-wheeling' diode which is forward biased. Hence, V_{in} being the voltage drop in the diode when conducting and assuming a perfect diode:

$$V_{in} = 0$$

3. The energy stored, and also I_l, will decay to zero and the diode will stop conducting. Then V_{in} is in parallel with, and equal to the sum of the voltage drop across the inductor (i.e. $R_l \times I_l$) and the output voltage (V_0). In this case, however, I_l will be zero and so:

$$V_{in} = V_0$$

The above three states describe the discontinuous mode of operation.

The equations associated to the *power circuit* are:

$$I_l' = \frac{(V_{in} - R_l \times I_l - V_0)}{L}$$

$$I_0 = \frac{(V_c + I_l \times R_c)}{(R_0 + R_c)}$$

$$V_0 = I_0 \times R_0,\ I_c = I_l - I_0,\ V_c' = I_c/C$$

C.1.3 Control circuit

The control circuit (figure C.3) provides the timing pulses to control the state of the transistor.

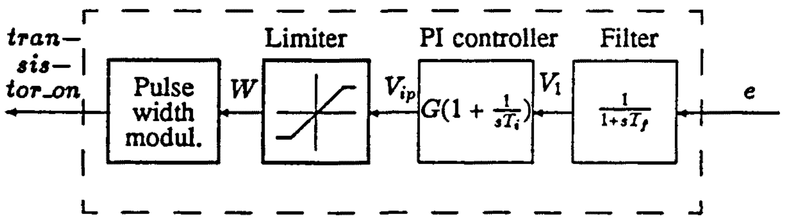

Figure C.3: *Control circuit.*

This circuit can be decomposed into four modules:

Filter: Its purpose is to reduce the ripple introduced by the sampling frequency (f_0) of the pulse-width modulator.

Proportional-integral controller The gain G of the PI controller has been chosen to be 1.0. The real pole time constant (T_i) is such that the oscillatory response of the LC filter stage in the power circuit is dumped.

Limiter: The mark-space ratio control signal input to the pulse-width modulator must not be outside the range of the ramp waveform. The limiter is used to keep the signal within this range.

Pulse-width modulator: The function of the pulse-width modulator is to provide the timing pulses to the base of the transistor which in turn controls the state of the transistor. The modulator is based on a ramp timing waveform and has two input signals:

- The sampling frequency (f_0).
- The mark-space ratio control signal (W).

The action of the pulse-width modulator is described as follows:

Initially the logical output (***transistor_on***) is set to 1 (or 0). The raising ramp successively crosses two pints: First, when the value of the ramp becomes equal to the input W (the output becomes 0); second, when the ramp becomes equal 1.0 at the end of each period (the output returns to 1 state and the ramp is reset to 0.0). The process is then repeated.

C.2 Bottom-up coding and testing

Figure C.4 represents the hierarchical relationships between SMPR submodels. Methodologically, the way to code the model is to start coding and testing submodels at level 0, then continue at level 1 and so on.

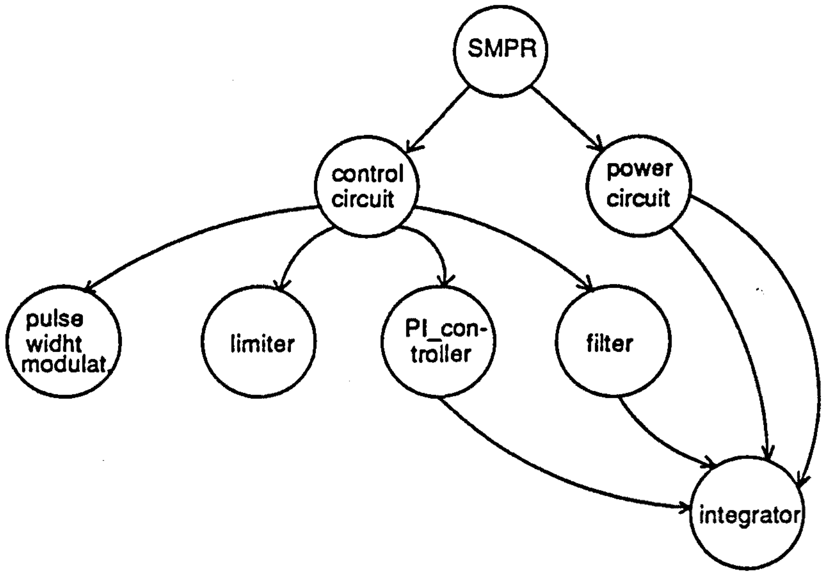

Figure C.4: *SMPR hierarchical structure.*

MUSS allows simulation users to hold in the simulation environment a set of user defined experiments and studies. This facility helps the users to verify the submodels, when the model does not behaves properly or when a malfunction is supposed in one or several submodels. Figure C.5 shows the defined experiments associated to SMPR submodels; in general, a submodel may have zero, one or more experiments associated to it and an experiment may call zero, one or more submodels.

In the following subsections the submodels of the SMPR in figure C.4 will be presented stressing the following aspects:

- *Code:* the submodel source code is written in the *MUSS* language. The design of the submodel structure has been influenced by GEST [Ören84a] (static region), ESL [Hay84a] (initial and dynamic region).
- *Analysis:* the submodel analysis includes the submodel digraph, the classification of its vertices, the segmentation process and the segment-link digraphs. Parameter and constant symbol vertices are not represented in the digraphs.
- *Test:* for the submodels with an experiment associated to them, a description of the experiment goal is given as well as some graphic results.

C.2.1 Integrator

Its calling syntax and use is equal to CSSL-like standard integrator. We keep the full name (*integrator*) instead of shortened names (INTEG [Mitchell81a], INTGRL [Syn85a]). In our opinion shortened names are meaningless for novice users.

Code

State variable declaration (state{...}) should include their type. Usually, simulation languages do not include the type of the state variables because they are implicitly supposed to be real, but we support that the state variables declaration grammar has to be similar to that of other variable declarations.

The submodel *MUSS* code will be:

```
Continuous submodel integrator is
      Static region
            inputs  {real ic,dx;}
            outputs {real x;}
            state {real x;}
```

```
        End static region;
        Initial region
              x = ic;
        End initial region;
        Dynamic region
              x' = dx;
        End dynamic region;
  End submodel integrator;
```

Code C.1 *Integrator submodel.*

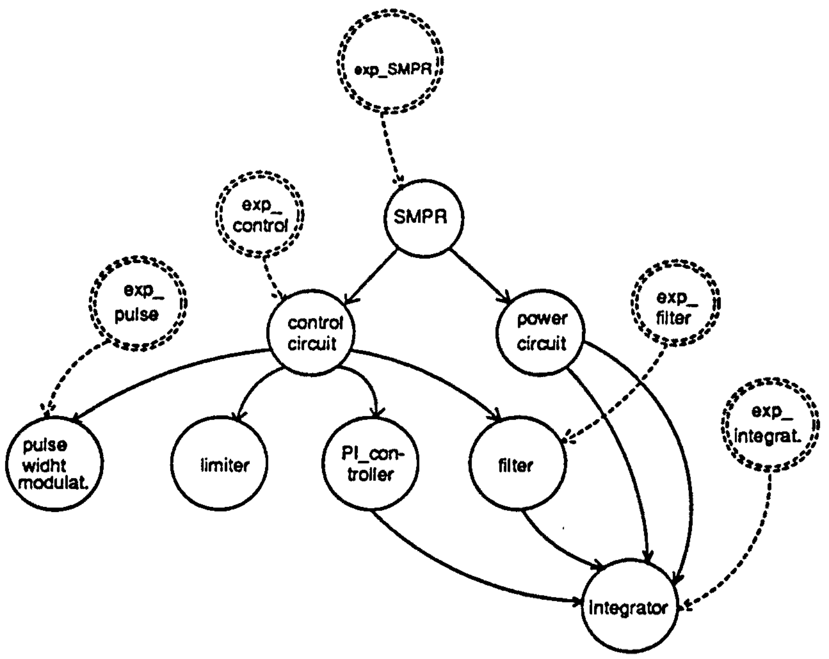

Figure C.5: *Experiments associated to the SMPR submodels. The experiments are represented with dashed circles.*

Analysis

The integrator intermediate code is written below,

```
-1-  Initial region
            x = ic;
     End initial region;
     Dynamic region
-2-         x' = dx;
     End dynamic region;
```

and the integrator submodel digraph is represented if figure C.6.

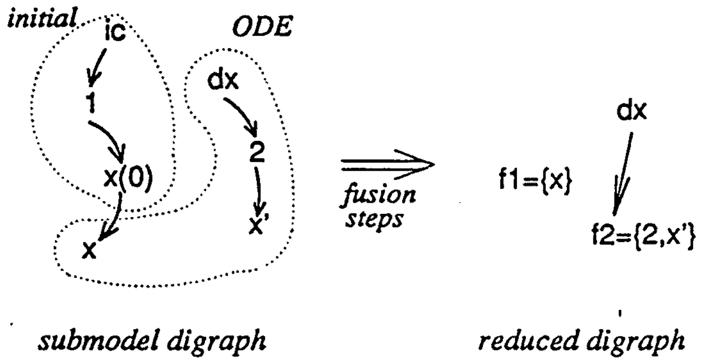

Figure C.6: *Integrator submodel digraph and segmentation process. The submodel digraph is first decomposed into the initial and the ODE segments and afterwards, the reduced digraph is get applying the fusion steps to the ODE segment digraph.*

Edge $(x(0), x)$ indicates that the state variable x will be initialized before each simulation run to the value $x(0)$. Otherwise, the analysis algorithm would detect and report an initialization error (subsection 3.3.2 in page 75).

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{ic, dx, x, x'\}$
   - *input vertices:* $Vs_i = \{ic, dx\}$
   - *output vertices:* $Vs_o = \{x\}$
   - *derivative vertices:* $Vs_d = \{x'\}$
   - *state vertices:* $Vs_s = \{x\}$

- *local vertices:* $Vs_{l_o} = \{x(0)\}$
- *initial vertices:* $Vs_n = \{x(0)\}$

2. Subsets of Ve

- *initial vertices:* $Ve_n = \{1\}$

<u>Initial segment digraph</u>

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1\}) \cap Ve = \{1\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{ic\}$
- *Output symbol vertices:* $Ivs_o = \emptyset$

<u>ODE segment digraph</u>

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n)$$
$$\cup (Ve - (Eve \cup Ve_n)) = (Q(\{x', x\}) \cap Ve) \cup \emptyset \cup \{2\} = \{2\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{dx\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{x\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.6.

The ODE subdigraphs are:

- State segment: The aim of this state segment is to make visible the state variable x to the calling higher level submodels.
  - $Sve = \emptyset$
  - $Svs_i = \emptyset$
  - $Svs_o = \{x\}$

- Derivative segment:
  - $Dve = \{2\}$
  - $Dvs_i = \{dx\}$
  - $Dvs_o = \emptyset$

The integrator segment-link digraph is represented in figure C.7 and the higher level calls to integrator segments will look like,

```
initial_integrator(ic);
x = state_integrator();
derivative_integrator(dx);
```

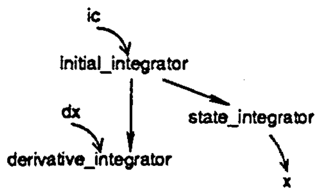

Figure C.7: *Integrator segment-link digraph.*

Test

Figure C.8 shows the behaviour of the integrator (output x). Its input (dx) switches from 0.0 to 0.2 at time equal to 1.0 *sec* and from 0.2 to −0.1 at time equal to 4.0 *sec*.

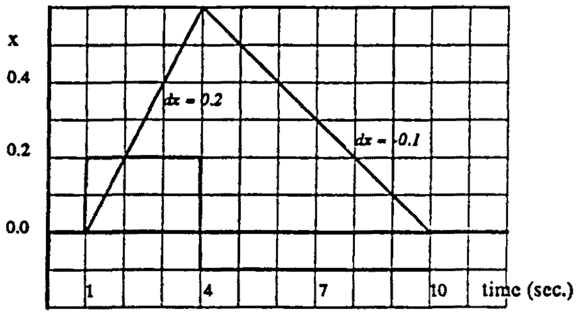

Figure C.8: *Integrator test results.*

C.2.2 Filter

The filter submodel is equivalent to the CSSL REALP simulation operator. Like CSSL languages, *MUSS* has a standard library of submodels (system library). Simulation users may use system or user defined library of submodels or a combination of both.

Code

```
Continuous submodel filter is
      Static region
            inputs  (real ic,tau,x;)
            outputs (real y;)
            submodels called (
                  integrator;
            )
      End static region;
      Dynamic region
             y = integrator(ic, (x-y)/tau);
      End dynamic region;
End submodel filter;
```

Code C.2 *Filter submodel.*

Analysis

Filter submodel intermediate code:

```
   Dynamic region
          /*... y = integrator(ic, (x-y)/tau);*/
-1-       initial_integrator(ic);
-2-       y = state_integrator();
-3-       derivative_integrator((x-y)/tau);
   End dynamic region;
```

The *MUSS* preprocessor splits the call to the integrator submodel into a call for each submodel segment (initial, state and derivative). Afterwards, it builds the submodel digraph being each segment call a vertex of the submodel digraph (transformation rule T.1 in page 53).

The submodel digraph is represented in figure C.9.

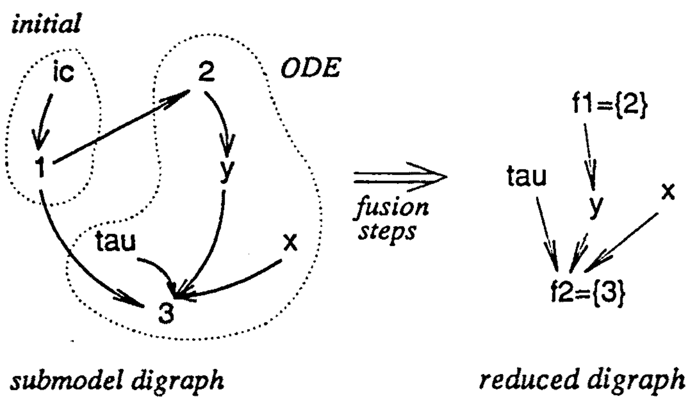

Figure C.9: *Filter submodel digraph and segmentation process.*

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{ic, tau, x, y\}$
   - *input vertices:* $Vs_i = \{ic, tau, x\}$
   - *output vertices:* $Vs_o = \{y\}$
2. Subsets of Ve
   - *derivative vertices:* $Ve_d = \{3\}$
   - *state vertices:* $Ve_s = \{2\}$
   - *initial vertices:* $Ve_n = \{1\}$

Initial segment digraph

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1\}) \cap Ve = \{1\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{ic\}$
- *Output symbol vertices:* $Ivs_o = \emptyset$

ODE segment digraph

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n)$$
$$\cup (Ve - (Eve \cup Ve_n)) = (Q(\{3, y\}) \cap Ve) \cup \emptyset \cup \{2,3\} = \{2,3\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{tau, x\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{y\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.9.

The ODE subdigraphs are:

- State segment:
  - $Sve = \{2\}$
  - $Svs_i = \emptyset$
  - $Svs_o = \{y\}$
- Derivative segment:
  - $Dve = \{3\}$
  - $Dvs_i = \{tau, x\}$
  - $Dvs_o = \emptyset$

The filter segment-link digraph is represented in figure C.10 and the higher level calls to filter segments will look like,

```
initial_filter(ic);
y = state_filter();
derivative_filter(tau,x);
```

Although the statement associated to vertex number 1 of the submodel digraph is in the dynamic region, it will be included into the initial segment because it is an initial vertex.

The *derivative_filter* segment must be called (in a higher level submodel) after the *state_filter* segment because it needs the state variable y for its own computations (edge *(state_filter, derivative_filter)*)

$$\begin{tikzcd}
 & \text{ic} \arrow[d] & & & \\
 & \text{initial\_filter} \arrow[ddd] \arrow[drr] & & & \\
 & & & \text{state\_filter} \arrow[ddll] \arrow[dr] & \\
\text{x} \arrow[dr] & & \text{tau} \arrow[dl] & & \text{y} \\
 & \text{derivative\_filter} & & &
\end{tikzcd}$$

Figure C.10: *Filter segment-link digraph.*

Test

The results of figure C.11 have been obtained in a multirun study. The responses of the filter (y) to an input step are given for $tau = 0.1, 1.0, 2.0, 10.0$.

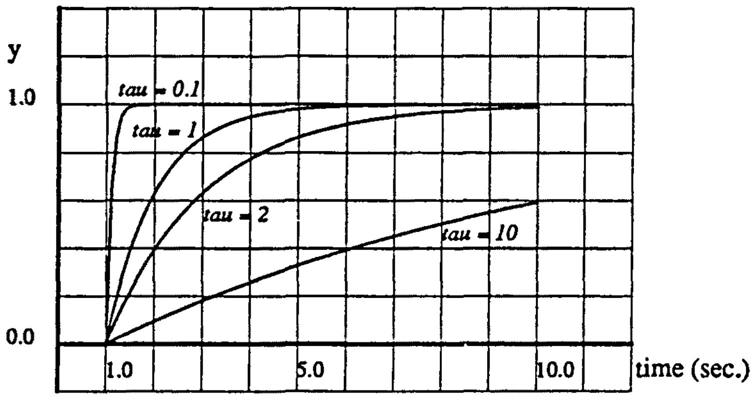

Figure C.11: *Filter test results obtained in a multirun study.*

C.2.3 Proportional plus integral controller

The PI_controller submodel represents a proportional plus integral controller.

Code

```
Continuous submodel PI_controller is
      Static region
            inputs  {real IC,Tc,K,x;}
            outputs {real y;}
            auxiliary variable {real z;}
            submodels called {
                  integrator;
            }
      End static region;
      Dynamic region
            z = integrator(IC,x/Tc);
            y = K*(x+z);
      End dynamic region;
End submodel PI_controller;
```

Code C.3 *PI_controller submodel.*

Analysis

The PI_controller intermediate code is:

```
   Dynamic region
         /*... z = integrator(IC,x/Tc);*/
-1-      initial_integrator(IC);
-2-      z = state_integrator();
-3-      derivative_integrator(x/Tc);
-4-      y = K*(x+z);
   End dynamic region;
```

and its associated submodel digraph is represented in figure C.12.

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{IC, Tc, K, x, y, z\}$
   - *input vertices:* $Vs_i = \{IC, Tc, K, x\}$
   - *output vertices:* $Vs_o = \{y\}$

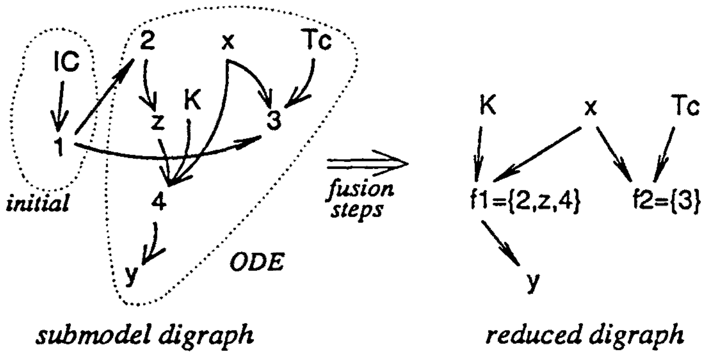

Figure C.12: *PI_controller submodel digraph and segmentation process.*

2. Subsets of Ve

   - *derivative vertices:* $Ve_d = \{3\}$
   - *state vertices:* $Ve_s = \{2\}$
   - *initial vertices:* $Ve_n = \{1\}$

<u>Initial segment digraph</u>

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1\}) \cap Ve = \{1\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{IC\}$
- *Output symbol vertices:* $Ivs_o = \emptyset$

<u>ODE segment digraph</u>

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{uh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n)$$
$$\cup (Ve - (Eve \cup Ve_n)) = (Q(\{3, y\}) \cap Ve) \cup \emptyset \cup \{2,3,4\} = \{2,3,4\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{Tc, K, x\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{y\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.12.

The ODE subdigraphs are:

- Algebraic segment:
  - $Ave = \{2,4\}$
  - $Avs_i = \{K, x\}$
  - $Avs_o = \{y\}$
- Derivative segment:
  - $Dve = \{3\}$
  - $Dvs_i = \{x, Tc\}$
  - $Dvs_o = \emptyset$

The PI_controller segment-link digraph is represented in figure C.13 and the higher level calls to its segments will look like,

```
initial_PI_controller(IC);
y = algebraic_PI_controller(K,x);
derivative_PI,controller(Tc,x);
```

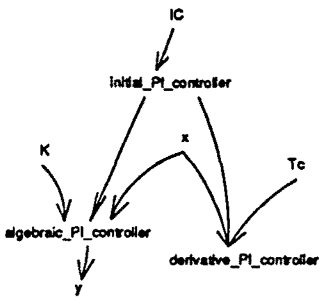

Figure C.13: *PI_controller segment-link digraph.*

C.2.4 Limiter

A limiter sets lower and upper limits to the amplitude of an input variable.

Code

```
Continuous submodel limiter is
      Static region
            inputs  {real ll,ul,x;}
            outputs {real y;}
      End static region;
      Dynamic region
            if(x>=ul) {
               y = ul;
            } else if(x<ll) {
               y = ll;
            } else {
               y = x;
            };
      End dynamic region;
End submodel limiter;
```

Code C.4 *Limiter submodel.*

Analysis

Limiter intermediate code is:

```
   Dynamic region
 -1-      groot1 = x-ul;
 -2-      groot2 = ll-x;
 -3-      if(lroot1) {
 -4-         y = ul;
          } else {
 -5-         if(lroot2) {
 -6-            y = ll;
             } else {
 -7-            y = x;
             };
          };
   End dynamic region;
```

The limiter submodel digraph is represented in figure C.14.

In this submodel, a declarative `if` statement appears. The causes of the discontinuities associated with this statement are the zero crossing functions $x - ul$ and $ll - x$. *Lroot*1 and *lroot*2 are the logical states associated to the discontinuous function and are recalculated each time a discontinuity is found.

See, looking to the `if` statement and to its representation in the submodel digraph that,

$$y = y_+ + y_{-+} + y_{--}$$

being y defined over continuous time and y_+, y_{-+} and y_{--} defined over continuous spans of time.

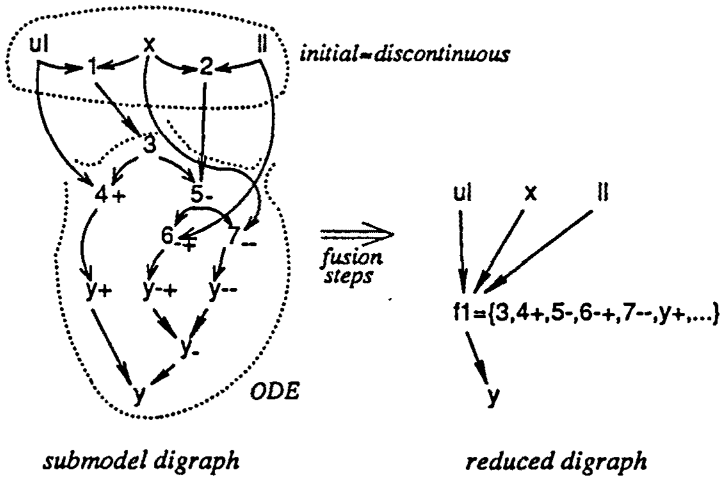

Figure C.14: *Limiter submodel digraph and segmentation process.*

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{ll, ul, x, y\}$
   - *input vertices:* $Vs_i = \{ll, ul, x\}$
   - *output vertices:* $Vs_o = \{y\}$
   - *local vertices:* $Vs_{lo} = \{y_+, y_-, y_{-+}, y_{--}\}$
   - *if vertices:* $Vs_{if} = \{y_+, y_-, y_{-+}, y_{--}\}$
2. Subsets of Ve
   - *discontinuous vertices:* $Ve_g = Ve_{gf} = \{1, 2\}$

- *if vertices:* $Ve_{if} = \{3, 5_-\}$

Initial segment digraph

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1,2\}) \cap Ve = \{1,2\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{ll, ul, x\}$
- *Output symbol vertices:* $Ivs_o = \emptyset$

Discontinuous segment digraph

- *Executable vertices:* $Eve = Q(Ve_g) \cap Ve = Q(\{1,2\}) \cap Ve = \{1,2\}$
- *Input symbol vertices:* $Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \{ll, ul, x\}$
- *Output symbol vertices:* $Evs_o = \emptyset$

ODE segment digraph

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{uh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n) \cup (Ve - (Eve \cup Ve_n)) =$$
$$(Q(\{y\}) \cap Ve) \cup \emptyset \cup \{3, 4_+, 5_-, 6_{-+}, 7_{--}\} =$$
$$\{3, 4_+, 5_-, 6_{-+}, 7_{--}\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{ll, ul, x\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{x\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.14.

The ODE subdigraphs are:

- Algebraic segment:
  - $Ave = \{3, 4_+, 5_-, 6_{-+}, 7_{--}\}$

- $Avs_i = \{ll, ul, x\}$
- $Avs_o = \{y\}$

The limiter segment-link digraph is represented in figure C.15 and the higher level calls to its segments will look like,

```
initial_limiter(ul,ll,x);
y = algebraic_limiter(ul,ll,x);
discontinuous_limiter(ul,ll,x);
```

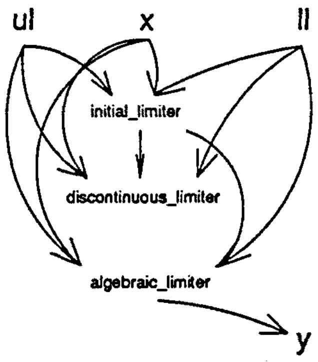

Figure C.15: *Limiter segment-link digraph.*

The initial segment holds the code needed to initialize the logical states associated to the discontinuous function of the submodel at initial time but does not include code for computing lower level discontinuous segments (see control submodel in subsection C.2.6 in page 197).

C.2.5 Pulse-width modulator

Two-level pulse-width modulator (PWM) which generates a square wave train with specified period and mark-space ratio.

Code

The calling sequence is:

```
y = pulse_width_modulator(time_delay,ratio,period);
```

where,

- *time_delay* is the time at which the pulse train starts.
- *ratio* is the modulation control signal in the range [0,1].
- *period* is the period of the pulse train in units of time.

```
Continuous submodel pulse_width_modulator is
      Static region
            inputs  {real time_delay,ratio,period;}
            outputs {logical y;}
            auxiliary variable {real start,ramp;}
      End static region;
      Initial region
            if(time_delay > 0.0) {
               y = FALSE;
               start = time_delay-period;
            } else if (time_delay < 0.0) {
               y = TRUE;
               start = -time_delay-period*ratio;
            } else {
               y = TRUE;
               start = 0.0;
            };
      End initial region
      Dynamic region
            ramp = (Time-start)/period;
            when(ramp>=ratio) {
               y = FALSE;
            } when(ramp>=1.0) {
               start = start+period;
               y = TRUE;
            };
      End dynamic region;
End submodel pulse_width_modulator;
```

Code C.5 *Pulse_width_modulator submodel.*

Analysis

Pulse-width modulator intermediate code is:

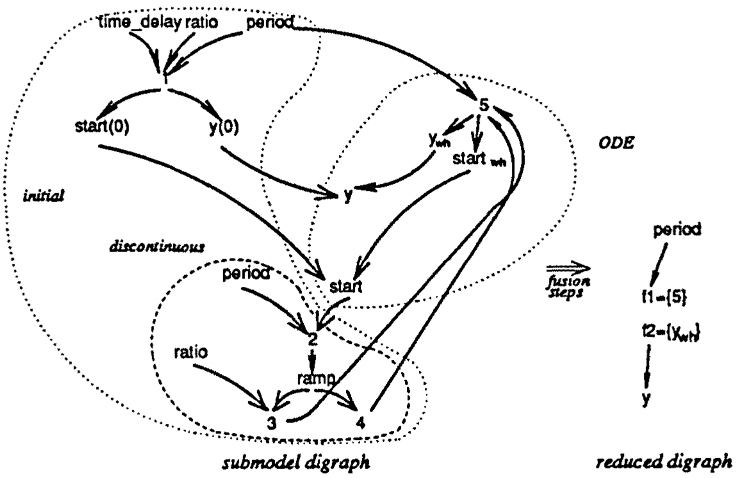

Figure C.16: *PWM submodel digraph and segmentation process. The edges $(5, start_{wh})$ and $(5, y_{wh})$ are suppressed when building the ODE segment digraph (subsection 3.3.4, page 85). Therefore, there is not a directed path between the input variable period and the output variable y in the reduced digraph.*

```
-1-Initial region
        if(time_delay > 0.0) {
           y = FALSE;
           start = time_delay-period;
        } else {
           if (time_delay < 0.0) {
               y = TRUE;
               start = -time_delay-period*ratio;
           } else {
               y = TRUE;
               start = 0.0;
           };
        };
   End initial region
   Dynamic region
-2-     ramp = (Time-start)/period;
-3-     grootl = ramp-ratio;
```

```
-4-       groot2 = ramp-1.0;
-5-       when(lroot1) {
             y = FALSE;
          } when(lroot2) {
             start = start+period;
             y = TRUE;
          };
   End dynamic region;
```

The associated submodel digraph is represented in figure C.16.

Code in the initial region as well as code in the when discontinuous statements have been defined being procedural and its segmentation is not allowed, i.e. sorting and segmentation algorithms manage the initial region and when statements as single executable vertices.

Since variables y (logical output variable) and $start$ (auxiliary variable needed to compute the discontinuous functions) must be initialized at initial time, it concerns to the analysis algorithm to check if both variables are initialized (subsection 3.3.2, analysis step (2) in page 75).

Actions associated to a when statement are placed into the derivative segment and should be executed just once each time its associated discontinuity function has a root. After a discontinuity has been localized, two extra calls to ODE subsegments have to be made to compute the actions associated to the when clause. Then, the discontinuous segment is called at event time to update the discontinuous function values.

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{time\_delay, ratio, period, y, start, ramp\}$
   - *input vertices:* $Vs_i = \{time\_delay, ratio, period\}$
   - *output vertices:* $Vs_o = \{y\}$
   - *local vertices:* $Vs_{lo} = \{start(0), y(0), start_{wh}, y_{wh}\}$
   - *initial vertices:* $Vs_n = \{start(0), y(0)\}$
   - *when vertices:* $Vs_{wh} = \{start_{wh}, y_{wh}\}$
2. Subsets of Ve
   - *discontinuous vertices:* $Ve_g = Ve_{gf} = \{3, 4\}$

- *initial vertices:* $Ve_n = \{1\}$
- *when vertices:* $Ve_{wh} = \{5\}$

Initial segment digraph

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1,3,4\}) \cap Ve = \{1,2,3,4\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{time\_delay, ratio, period\}$
- *Output symbol vertices:* $Ivs_o = \emptyset$

Discontinuous segment digraph

- *Executable vertices:* $Eve = Q(Ve_g) \cap Ve = Q(\{3,4\}) \cap Ve = \{2,3,4\}$
- *Input symbol vertices:* $Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \{ratio, period\}$
- *Output symbol vertices:* $Evs_o = \emptyset$

ODE segment digraph

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n)$$
$$\cup (Ve - (Eve \cup Ve_n)) = (Q(\{y,5\}) \cap Ve) \cup \emptyset \cup \{5\} = \{5\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{period\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{y\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.16.

The ODE subdigraphs are:

- State segment:
  - $Sve = \emptyset$
  - $Svs_i = \emptyset$

  - $Svs_o = \{y\}$

- Derivative segment:
  - $Dve = \{5\}$
  - $Dvs_i = \{period\}$
  - $Dvs_o = \emptyset$

The pulse_width_modulator segment-link digraph is represented in figure C.17 and the higher level calls to its segments will look like,

```
initial_pulse_width_modulator(time_delay,ratio,period);
y = state_pulse_width_modulator();
derivative_pulse_width_modulator(period);
discontinuous_pulse_width_modulator(ratio,period);
```

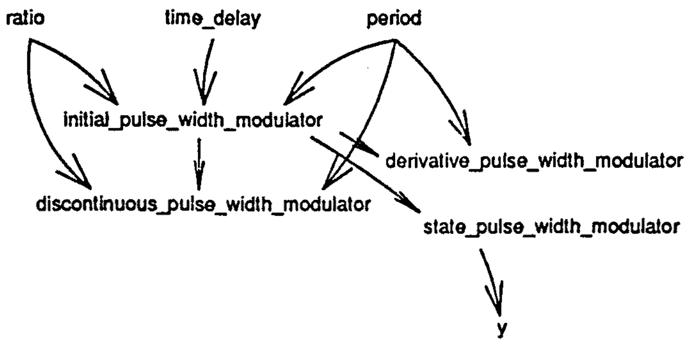
ratio
time_delay
period
initial_pulse_width_modulator
derivative_pulse_width_modulator
discontinuous_pulse_width_modulator
state_pulse_width_modulator
y

Figure C.17: *PWM segment-link digraph.*

Test

Figure C.18 shows the behavior of the pulse-width modulator submodel. See the effect of the modulator control signal $ratio$ in the width of each pulse, width increases when $ratio$ increases. Let see in more detail how it works. When the output y is set to 1, the ramp is reset to 0. When the ramp raises, two separate triggering points are passed. The first crossing point arises when $ramp$ becomes equal to $ratio$, then the output is reset to 0. The second point is passed when $ramp$ becomes equal to 1.0, now the action consists on setting again the output y to 1. The process is then repeated for ever.

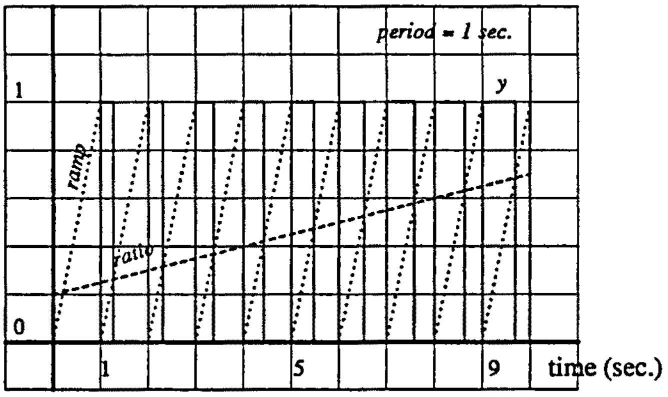

Figure C.18: *Pulse-width modulator test results.*

C.2.6 Control circuit

Code

```
Continuous submodel control_circuit is
      Static region
            inputs  {real error;}
            outputs {logical transistor_on;}
            auxiliary variables { real W,Vip,V1;}
            parameters {
                  real upper_limit = 0.05,
                       lower_limit = 0.95,
                       V1ic = 0.0125, V2ic = 0.50,
                       Tf = 2.0E-4, Ti = 4.5E-4,
                       Td = 0.0, G = 1.0, period = 1.25E-5;
            }
            submodels called {
                  filter;
                  PI_controller;
                  limiter;
                  pulse_width_modulator;
            }
      End static region;
      Dynamic region
            V1  = filter(V1ic,Tf,error);
            Vip = PI_controller(V2ic,Ti,G,V1);
            W   = limiter(lower_limit,upper_limit,Vip);
```

```
              transistor_on = pulse_width_modulator(Td,W,period);
          End dynamic region;
      End submodel control_circuit;
```

Code C.6 *Control_circuit submodel.*

Analysis

The control_circuit intermediate code is:

```
      Dynamic region
            /*... V1  = filter(V1ic,Tf,error);*/
 -1-        initial_filter(V1ic);
 -2-        V1 = state_filter();
 -3-        derivative_filter(Tf,error);

             /*... Vip = PI_controller(V2ic,Ti,G,V1);*/
 -4-        initial_PI_controller(V2ic);
 -5-        Vip = algebraic_PI_controller(G,V1);
 -6-        derivative_PI_controller(Ti,V1);

             /*... W   = limiter(lower_limit,upper_limit,Vip);*/
 -7-        initial_limiter(lower_limit,upper_limit,Vip);
 -8-        W = algebraic_limiter(lower_limit,upper_limit,Vip);
 -9-        discontinuous_limiter(lower_limit,upper_limit,Vip);

             /*... transistor_on =
                 pulse_width_modulator(Td,W,period);*/
 -10-       initial_pulse_width_modulator(Td,W,period);
 -11-       transistor_on = state_pulse_width_modulator();
 -12-       derivative_pulse_width_modulator(period);
 -13-       discontinuous_pulse_width_modulator(W,period);
      End dynamic region;
```

The control_circuit submodel digraph is represented in figure C.19.

An important overlap exists between submodel segments, but looking in more detail to the submodel digraph it can be seen that the overhead is small because the initial region is only executed once before each simulation run. Thus, its overall cost can be neglected.

The symbol vertices and the executable vertices can be grouped into the following subsets:

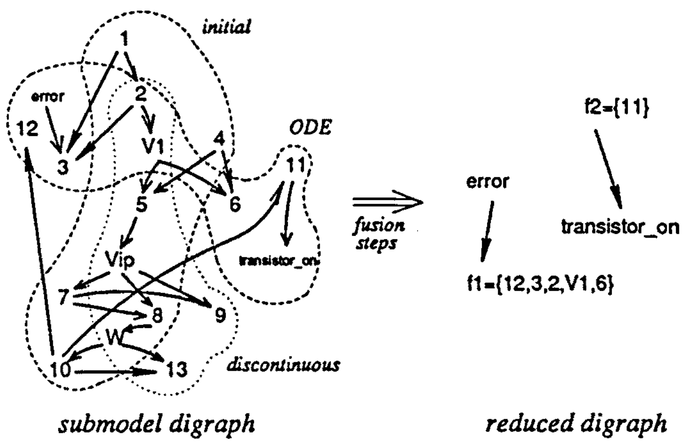

Figure C.19: *Control_circuit submodel digraph and segmentation process. Notice that executable vertices associated to calls to the state an algebraic segments are included in the initial segment digraph.*

1. Subsets of Vs
   - *global vertices:*

$$Vs_{gl} = \{error, transistor\_on, W, Vip, V1, upper\_limit,$$
$$lower\_limit, V1ic, V2ic, Tf, Ti, Td, G, period\}$$

   - *parameter vertices:*

$$Vs_p = \{upper\_limit, lower\_limit, V1ic, V2ic,$$
$$Tf, Ti, Td, G, period\}$$

   - *input vertices:* $Vs_i = \{error\}$
   - *output vertices:* $Vs_o = \{transistor\_on\}$

2. Subsets of Ve
   - *derivative vertices:* $Ve_d = \{3, 6, 12\}$
   - *discontinuous vertices:* $Ve_g = Ve_{gs} = \{9, 13\}$
   - *state vertices:* $Ve_s = \{2, 11\}$
   - *initial vertices:* $Ve_n = \{1, 4, 7, 10\}$

<u>Initial segment digraph</u>

- *Executable vertices:*

  $$Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1,4,7,10\}) \cap Ve = \{1,2,4,5,7,8,10\}$$

- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \emptyset$
- *Output symbol vertices:* $Ivs_o = \emptyset$

<u>Discontinuous segment digraph</u>

- *Executable vertices:* $Eve = Q(Ve_g) \cap Ve = Q(\{9,13\}) \cap Ve = \{2,5,9,8,13\}$
- *Input symbol vertices:* $Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \emptyset$
- *Output symbol vertices:* $Evs_o = \emptyset$

<u>ODE segment digraph</u>

- *Executable vertices:*

  $$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n) \cup (Ve - (Eve \cup Ve_n)) =$$

  $$(Q(\{3,6,12,transistor\_on\}) \cap Ve) \cup \emptyset \cup \{11,3,12,6\} = \{3,6,12,2,11\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{error\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{transistor\_on\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.19.

The ODE subdigraphs are:

- State segment:
  - $Sve = \{11\}$
  - $Svs_i = \emptyset$
  - $Svs_o = \{transistor\_on\}$

- Derivative segment:
  - $Dve = \{12, 3, 2, 6\}$
  - $Dvs_i = \{error\}$
  - $Dvs_o = \emptyset$

The control_circuit segment-link digraph is represented in figure C.20 and the higher level calls to its segments will look like,

```
initial_control_circuit();
transistor_on = state_control_circuit();
derivative_state_control_circuit(error);
discontinuous_control_circuit();
```

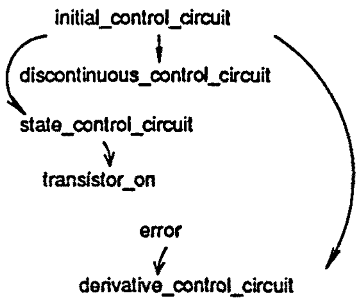

Figure C.20: *Control_circuit segment-link digraph.*

Test

The control circuit (figure C.3) provides the timing pulses (*transistor_on*) to control the state of the transistor. Input *error* (figure C.21) is filtered ($V1$), then is input to a PI controller (Vip) and limited (W). W is the pulse-width modulator mark-space control signal.

In this case, input *error* is a three level signal chosen for testing the control_circuit submodel.

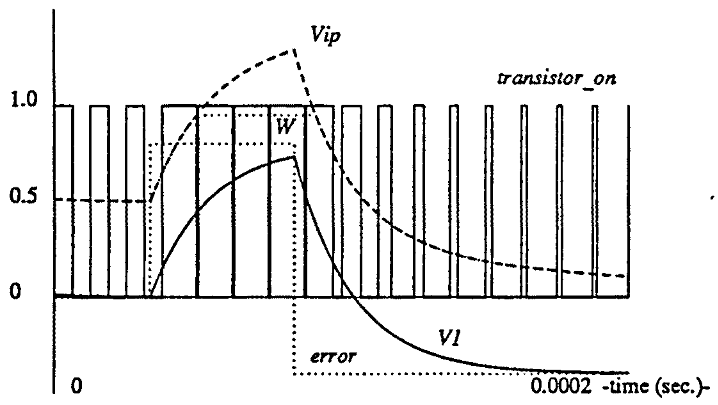

Figure C.21: *Control_circuit test results.*

C.2.7 Power circuit

Code

```
Continuous submodel power_circuit is
      Static region
            inputs  {real Vs,R0,transistor_on;}
            outputs {real V0;}
            auxiliary variables {real I0,Ic,Il,Vc,dIl,dVc;
                                logical idiode;}
            parameters {
               real L = 2.1E-5, Rl = 0.0,
                       Rc = 0.1, C = 3.5E-4;
            }
            submodels called {
                  integrator;
            }
      End static region;
      Dynamic region
            dIl = (Vin-Rl*Il-V0)/L;
            Il  = integrator(0.0,dIl);
            I0  = (Vc+Il*Rc)/(R0+Rc);
            V0  = I0*R0;
            Ic  = Il-I0;
            dVc = Ic/C;
            Vc  = integrator(50.0,dVc);
            idiode = Il >= 0.0;
            if(transistor_on) {
               Vin = Vs;
            } else if (idiode) {
               Vin = 0.0;
            } else {
               Vin = V0;
            };
        End dynamic region;
    End submodel power_circuit;
```

Code C.7 *Power_circuit submodel.*

Analysis

The power_circuit intermediate code is:

```
    Dynamic region
 -1-       dIl = (Vin-Rl*Il-V0)/L;

           /*... Il  = integrator(0.0,dIl);*/
 -2-       initial_integrator(0.0);
 -3-       Il = state_integrator();
 -4-       derivative_integrator(dIl);

 -5-       I0  = (Vc+Il*Rc)/(R0+Rc);
 -6-       V0  = I0*R0;
 -7-       Ic  = Il-I0;
 -8-       dVc = Ic/C;

           /*... Vc  = integrator(50.0,dVc);*/
 -9-       initial_integrator(50.0);
 -10-      Vc = state_integrator();
 -11-      derivative_integrator(dVc);

 -12-      groot1 = Il-0.0;
 -13-      idiode = lroot;
 -14-      if(transistor_on) {
 -15-         Vin = Vs;
           } else {
 -16-         if (idiode) {
 -17-             Vin = 0.0;
              } else {
 -18-             Vin = V0;
              };
           };
    End dynamic region;
```

Its associated submodel digraph is represented in figure C.22.

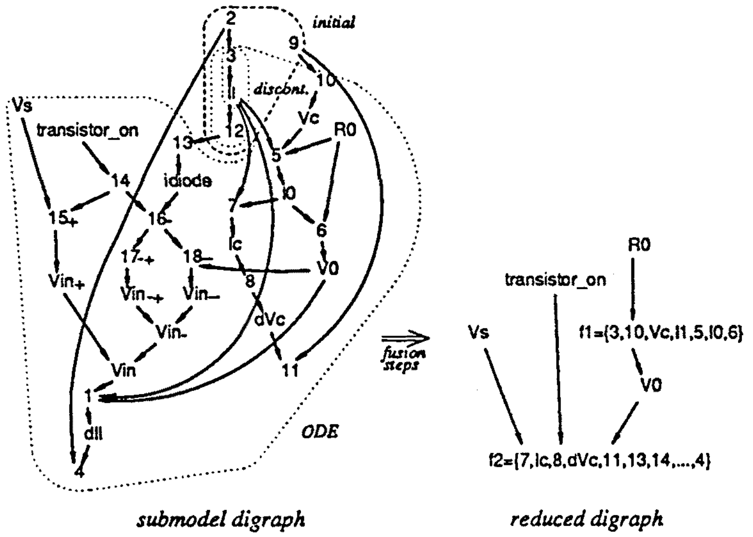

Figure C.22: *Power_circuit submodel digraph and segmentation process.*

The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:*
   $$Vs_{gl} = \{Vs, R0, transistor\_on, V0, I0, Ic, Il, Vc, dIl,$$
   $$dVc, idiode, L, Rl, Rc, C\}$$
   - *parameter vertices:* $Vs_p = \{L, Rl, Rc, C\}$
   - *input vertices:* $Vs_i = \{Vs, R0, transistor\_on\}$
   - *output vertices:* $Vs_o = \{V0\}$
   - *local vertices:* $Vs_{lo} = \{Vin_+, Vin_-, Vin_{-+}, Vin_{--}\}$
   - *if vertices:* $Vs_{if} = \{Vin_+, Vin_-, Vin_{-+}, Vin_{--}\}$
2. Subsets of Ve
   - *derivative vertices:* $Ve_d = \{4, 11\}$

- *discontinuous vertices:* $Ve_g = Ve_{gf} = \{12\}$
- *initial vertices:* $Ve_n = \{2, 9\}$
- *if vertices:* $Ve_{if} = \{13, 14, 16_-\}$

<u>Initial segment digraph</u>

- *Executable vertices:*

$$Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{2,9,12\}) \cap Ve = \{2,3,12,9\}$$

- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \emptyset$
- *Output symbol vertices:* $Ivs_o = \emptyset$

<u>Discontinuous segment digraph</u>

- *Executable vertices:* $Eve = Q(Ve_g) \cap Ve = Q(\{12\}) \cap Ve = \{12,3,2\}$
- *Input symbol vertices:* $Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \emptyset$
- *Output symbol vertices:* $Evs_o = \emptyset$

<u>ODE segment digraph</u>

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n) \cup (Ve - (Eve \cup Ve_n)) =$$
$$(Q(4, 11, V0) \cap Ve) \cup \emptyset \cup \{10,5,6,7,8,13,14,15_+,16_-,17_{-+},18_{--},1,4,11\} =$$
$$\{10,5,6,7,8,13,14,15_+,16_-,17_{-+},18_{--},1,4,11\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{transistor\_on, Vs, R0\}$
- *Output symbol vertices:* $Ovs_o = Vs_o = \{V0\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.22.

The ODE subdigraphs are:

- Algebraic segment:
  - $Ave = \{3, 10, 5, 6\}$
  - $Avs_i = \{R0\}$
  - $Avs_o = \{V0\}$
- Derivative segment:
  - $Dve = \{7, 8, 11, 13, 14, 15_+, 16_-, 17_{-+}, 18_{--}, 1, 4\}$
  - $Dvs_i = \{Vs, transistor\_on\}$
  - $Dvs_o = \emptyset$

The power_circuit segment-link digraph is represented in figure C.23 and the higher level calls to its segments will look like,

```
initial_power_circuit();
V0 = algebraic_power_circuit(R0);
derivative_power_circuit(Vs,transistor_on);
discontinuous_power_circuit();
```

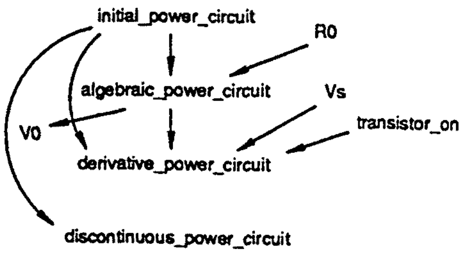

Figure C.23: *Power_circuit segment-link digraph.*

C.2.8 SMPR circuit

Code

```
Continuous submodel SMPR is
      Static region
            inputs  {real Vs,R0,Vref;}
```

```
              outputs {real V0;}
              auxiliary variables {
                   real e;
                   logical transistor_on; }
              submodels called {
                   power_circuit;
                   control_circuit;}
        End static region;
        Dynamic region
              V0 = power_circuit(Vs,R0,transistor_on);
              e  = Vref - V0;
              transistor_on = control_circuit(e);
        End dynamic region;
  End submodel SMPR;
```

Code C.8 *SMPR submodel.*

Analysis

Although SMPR submodel is placed on top of the submodel hierarchy, its analysis is as simple as the lower level submodels analysis.

The executable vertices of the submodel digraph (figure C.24) are:

```
    Dynamic region
          /*... V0 = power_circuit(Vs,R0,transistor_on);*/
 -1-      initial_power_circuit();
 -2-      V0 = algebraic_power_circuit(R0);
 -3-      derivative_power_circuit(Vs,transistor_on);
 -4-      discontinuous_power_circuit();

 -5-      e  = Vref - V0;

          /*... transistor_on = control_circuit(e);*/
 -6-      initial_control_circuit();
 -7-      transistor_on = state_control_circuit();
 -8-      derivative_state_control_circuit(e);
 -9-      discontinuous_control_circuit();
    End dynamic region;
```

1. Subsets of Vs

   - *global vertices:* $Vs_{gl} = \{Vs, R0, Vref, V0, e, transistor\_on\}$

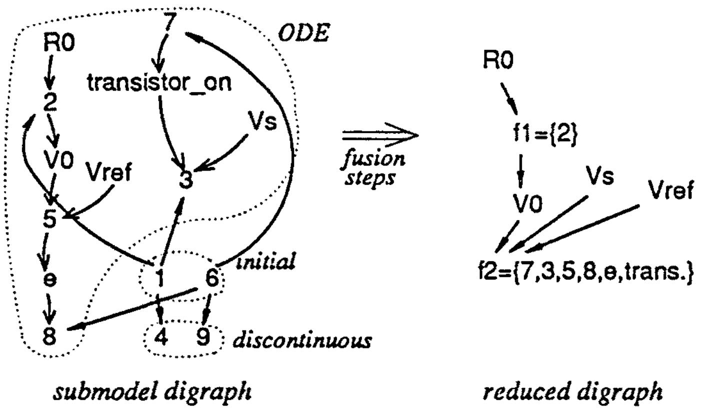

Figure C.24: *SMPR submodel digraph and segmentation process.*

- *input vertices:* $Vs_i = \{Vs, R0, Vref\}$
- *output vertices:* $Vs_o = \{V0\}$

2. Subsets of Ve

- *derivative vertices:* $Ve_d = \{3, 8\}$
- *discontinuous vertices:* $Ve_g = Ve_{gf} = \{4, 9\}$
- *state vertices:* $Ve_s = \{7\}$
- *initial vertices:* $Ve_n = \{1, 6\}$

Initial segment digraph

- *Executable vertices:* $Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1, 6\}) \cap Ve = \{1, 6\}$
- *Input symbol vertices:* $Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \emptyset$
- *Output symbol vertices:* $Ivs_o = \emptyset$

Discontinuous segment digraph

- *Executable vertices:* $Eve = Q(Ve_g) \cap Ve = Q(\{9, 4\}) \cap Ve = \{9, 4\}$
- *Input symbol vertices:* $Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \emptyset$

- *Output symbol vertices:* $Evs_o = \emptyset$

ODE segment digraph

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n) \cup (Ve - (Eve \cup Ve_n)) =$$

$$(Q(3,8,V0) \cap Ve) \cup \emptyset \cup \{7,3,2,5,8\} = \{7,3,2,5,8\}$$

- *Input symbol vertices:* $Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{Vs, R0, Vref\}$

- *Output symbol vertices:* $Ovs_o = Vs_o = \{V0\}$

The ODE segment digraph and its associated reduced digraph are represented in figure C.24.

The ODE subdigraphs are:

- Algebraic segment:
  - $Ave = \{2\}$
  - $Avs_i = \{R0\}$
  - $Avs_o = \{V0\}$
- Derivative segment:
  - $Dve = \{7,3,5,8\}$
  - $Dvs_i = \{Vs, Vref\}$
  - $Dvs_o = \emptyset$

The SMPR segment-link digraph is represented in figure C.25 and the higher level calls to its segments will look like,

```
initial_SMPR()
V0 = algebraic_SMPR(R0)
derivative_SMPR(Vs,Vref)
discontinuous_SMPR()
```

$$\begin{tikzcd}
 & \text{initial\_SMPR} \arrow[d] \arrow[ddd, bend right, to=4-1] \arrow[dddd, bend left, to=5-3] & & \\
 & \text{discontinuous\_SMPR} & & \\
 & \text{R0} \arrow[dl] & \text{Vs} \arrow[dd] & \text{Vref} \arrow[ddl] \\
\text{algebraic\_SMPR} \arrow[d] \arrow[drr] & & & \\
\text{V0} & & \text{derivative\_SMPR} &
\end{tikzcd}$$

Figure C.25: *SMPR segment-link digraph.*

Test

Figure C.26 shows the response of the SMPR circuit corresponding to a rise in the supply voltage V_s from 70 *volt.* to 90 *volt.* at time equal zero. Initial conditions were given for the steady state.

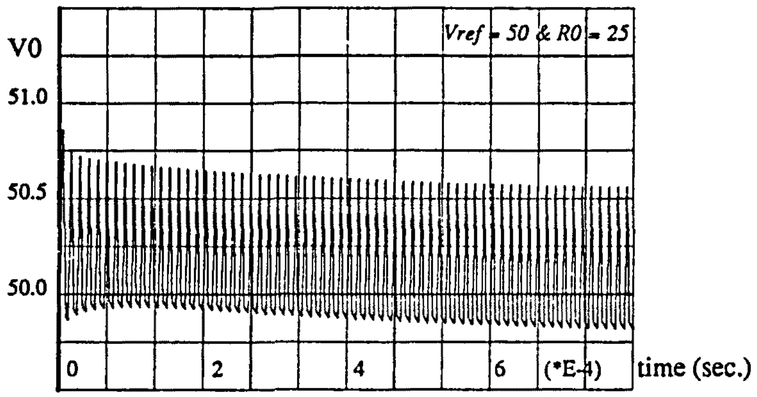

Figure C.26: *SMPR test results.*

C.3 Summary

The switched-mode power regulator circuit has been selected as a case study because in spite of its academical flavour, it is complex enough to increase the understanding on the proposed analysis and segmentation methods.

Besides testing the code of the segments, this case study has also been used to successfully test the management and execution of hierarchical models in the defined *MUSS* environment.

Appendix D

Acronyms

ACSL : Advanced Continuous Simulation Language.

CAMAS : Computer Aided Modelling, Analysis and Simulation environment.

COSMOS : Combined System Modelling and Simulation.

COSY : Combined Continuous and Discrete Systems.

CSMP : Continuous System Modelling Program.

CSSL : Continuos System Simulation Languages.

DEC : Digital Equipment Corporation.

DISCO : Discrete and Continuous system simulation.

ESL : European Simulation Language.

GEST : General System Theory implementator.

IMACS : International Association for Mathematics and Computers in Simulation.

LEX : Lexical analyzer.

MCL : MUSS Command Language.

MUSS : Modular Simulation System.

ODE : Ordinary Differential Equations.

SCS : Society for Computer Simulation.

SIDOPS : Structured Interdisciplinary Description of Physical Systems.

SMPR : Switched-Mode Power Regulator.

TC3 : Technical Committee on Simulation Software (IMACS).

YACC : Yet another Compiler Compiler.

Bibliography

[Aho77a] Aho A. V. and Ullman J. D. in *Principles of compiler design*, Addison-Wesley (1977).

[Baer80a] Baer J. L. in *Computer systems architecture*, ed. Friedman A. D., Computer Science Press (1980).

[Baker83a] Baker N.J. and Smart P.J. "The SYSMOD simulation language" in *The European Simulation Congress ESC 83* pp. 281-286, Aachen. (Sept. 1983).

[Birta85a] Birta L. G.,Ören T. I. and Kettenis D. L. "A Robust Procedure for Discontinuity Handling in Continuous System Simulation" in *Transactions of the Society for Computer Simulation* Vol. 2 (3), pp. 189-206 (September 1985).

[Breiteneck83a] Breitenecker F. "The Concept of Supermacros in Today's and Future Simulation Languages" in *Math. and Comp. in Sim.* pp. 279-289 (June 1983).

[Brennan68a] Brennan R. D. "Continuous Systems Modelling Programs : State-of-the Art and Prospects for Development" in *Proc. of the IFIP Working Conference on Simulation Programming Languages* (1968).

[Broenink85a] Broenink J. F. and Twilhaar D. N. G. "Camas, a computer aided modelling, analysis and simulation environment" in *Fourth International Conference and Exhibition* (June 1985).

[Broenink85b] Broenink J. F. "Sidops, a bond graph based modelling language" in *11th IMACS World Congress*, Oslo. (August 1985).

[Carver75a] Carver M. B. "Efficient Integration Over Discontinuities in Ordinary Differential Equations" in *IMACS International Symposium on Simulation Software for Differential Equtions* (March 1975).

[Carver79a] Carver M. B. "The FORSIM IV Distributed System Simulation Package" in *Methodology in Systems Modelling and Simulation* pp. 249-267, ed. Zeigler et al. North Holland (1979).

[Cellier76a] Cellier F. E. and Blitz A. E. "GASP-V : a universal simulation package" pp. 391-342 in *Simulation of systems : proceedings of the 8th AICA congress*, ed. North Holland pub. co. (August 1976).

[Cellier79a] Cellier F. E. "Combined continuous/discrete system simulation languages... usefulness, experiences and future development" in *Methodology in systems modelling and simulation* pp. 201-220, ed. Zeigler et al. North Holland (1979).

[Cellier79b] Cellier F. E. and Bongulielmi A. P. "On the Usefulness of Deterministic Grammars for Simulation" in *Association for Computing Machinery Special Interest Group on Simulation* (1979).

[Cellier79c] Cellier F. E. and Bongulielmi L. P. "The COSY simulation language" pp. 271-281 in *Simulation of systems : proc. of the 9th IMACS Conf. on Simulation of Systems*, ed. North Holland pub. co., Sorrento, Italy. (1979).

[Cellier84a] Cellier F. E. "How to Enhance the Robustness of Simulation Software" pp. 519-536 in *Simulation and Model-Based Methodologies: An Integrative View*, ed. Ören T. I. (1984).

[Chu69a] Chu in *Digital Simulation of Continuous Systems*, McGraw-Hill Book C. (1969).

[Clancy65a] Clancy J. J. and Fineberg M. F. "Digital Simulation Languages, a critique and a guide" Vol. 27 in *AFIPS Conference Proceedings* (1965).

[Crosbie82a] Crosbie R. E. and Hay J. L. "Towards new standards for continuous-system simulation languages" in *Proceedings of the 1982 summer Computer Simulation Conference* pp. 186-190, Denver. (July 1982).

[Crosbie82b] Crosbie R. E. and Cellier F. E. "Progress in simulation language standards = An activity report for Technical Committee TC3 on Simulation Software 1979-82" Vol. 1 pp. 411-412 in *Proc. 10th IMACS World Congress on Syst. Simul. and Scient. Comp.*, Montreal. (Aug. 1982).

[Crosbie82d] Crosbie R. E. "Interactive and Real-Time Simulation" pp. 393-406 in *Progress in Modelling and Simulation*, ed. Cellier F. E., Academic Press (1982).

[Crosbie83a]	Crosbie R. E.,Hay J. L.,Savey S. and Pearce J. G. "ESL - A new continuous-system simulation language" in *Summer Computer Simulation Conference* (1983).
[Crosbie86a]	Crosbie R. E. and Hay J. L. "Description and processing of discontinuities with the ESL simulation language" pp. 30-38 in *Languages for continuous system simulation*, ed. Cellier F. D. (1986).
[Dahl66a]	Dahl O. J. and Nygaard K. "SIMULA - An ALGOL- based Simulation Language" pp. 671-678 SIMULA simulation in *CACM* (1966).
[Dec84a]	Dec "VAX/VMS DCL Commands and Lexical Functions" in *VAX/VMS Software*, Massachusetts. (1984).
[Ellison81a]	Ellison D. "Efficient Automatic Integration of Ordinary Differential Equations with Discontinuities" in *Mathematics and Computers in Simulation* Vol. 23 (1), pp. 12-20 (March 1981).
[Elmqvist78a]	Elmqvist H. in *DYMOLA - A Structured Model Language for Large Continuous Systems*, Lung, Sweden. (1978).
[Elmqvist80a]	Elmqvist H. "Manipulation of Continuous Models Based on Equations to Assignment Statements." pp. 15-21 in *Simulation of Systems' 79*, North-Holland (1980).
[Elzas79a]	Elzas M. S. "What is needed for robust simulation ?" in *Methodology in Systems Modelling and Simulation*, ed. Zeigler et al., North-Holland (1979).
[Freeman84a]	Freeman T. G. and Benyon P. R. "Comments on the proposal for a new simulation language standard" in *TC3-IMACS Simulation Software Committee Newsletter No. 12* (August 1984).
[Golden85a]	Golden D. G. "Software engineering considerations for the design of simulation languages" in *Simulation* (4), pp. 169-178 (October 1985).
[Guasch84a]	Guasch A.,Huber R. and Ilari J. "ICDSL (Instituto de Cibernetica Digital Simulation Language)." pp. 349-356 in *IFAC Simposium, Automatica en la Industria*, Zaragoza. (Noviembre 1984).
[Guasch85d]	Guasch A. "Aplicación de la teoría de grafos a la ordenación de un submodelo MUSS" in *Memoria interna*, Barcelona. (octubre 1985).

[Guasch86a] Guasch A. and Huber R. "Precompiled Submodels: A General Sorting Procedure" pp. 172-178 in *Proc. of the 2nd European Simulation Congress*, Antwerp, Belgium. (Sept. 1986).

[Guasch86b] Guasch A. "A draft of the *MUSS* grammar." in *Internal Report ICIN - 860715/04*, Barcelona. (July 1986).

[Hay84a] Hay J. L.,Crosbie R. E. and Pearce J. G. "ESL: A Simulation Language for the Space Industry" in *ESA Journal* Vol. 8 (1984).

[Hay85a] Hay J. L.,Pearce J. G.,Turnbull L. and Crosbie R. E. in *ESL software user manual* (April 1985).

[Helsgaun80a] Helsgaun K. "DISCO-SIMULA-based language for continuous combined and discrete simulation" in *Simulation* pp. 1-12 (Julay 1980).

[Hindmarsh82a] Hindmarsh A. C. "Towards a Systematized Collection of ODE Solvers" Vol. 1 pp. 427-429 in *Proc. 10th IMACS World Congress on Syst. Simul. and Scient. Comp.*, Montreal. (1982).

[Hoare81a] Hoare C. A. "The emperor's old clothes" in *Communications of the ACM* Vol. 24 (2), pp. 75-83 (February 1981).

[Hollander81a] Hollander E. H. D.,Spriet J. A. and Vanteenkiste G. C. "The 'Algebraic Loop' problem in continuous system simulation" pp. 368-379 in *Proc. of the UKSC conference on computer simulation*, Harrogate. (May 1981).

[Horst86a] Horst A. "Model validation in a modern simulation environment" pp. 64-72 in *Languages for continuous system simulation*, ed. Cellier F. E., SCS (1986).

[Huber82b] Huber R. M.,Basanez L.,Juan R. A. and Fernandez R. O. "Hybrid computation experience at the Instituto de Cibernetica, Spain" Vol. 4 in *Proc. 10th IMACS World Congress on Syst. Simul. and Scient. Comp.*, Montreal. (Aug. 1982).

[Huber86a] Huber R. M. and Guasch A. "Towards a specification of the structure for Continuous System Simulation Languages: Evolution and a proposal draft" Vol. 2 in *Computer Systems: Architecture, Applications*, ed. M. Ruschitzka, North-Holland (1986).

[Hurst73a] Hurst N. R. in *GASP IV: Combined Continuous/Discrete Fortran Based Simulation Language*, Purdue University. (1973).

[Jackson60a] Jackson A. S. in *Analog Computation*, McGraw-Hill (1960).

[James67a] James M. L.,Smith G. M. and Wolford J. C. in *Analog Computer Simulation of Engineering Systems*, International Textbook Company (1967).

[James77a] James M. L.,Smith G. M. and Wolford J. C. in *Applied Numerical Methods for Digital Computation*, Harper and Row (1977).

[Johnson75a] Johnson S. C. "Yacc: Yet Another Compiler Compiler" in *Computing Science Technical Report N. 32*, Bell Laboratories, Murray Hill, NJ. (1975).

[Karplus82a] Karplus W. J. and Makoui A. "The Role of Dataflow Methods in Continuous System Simulation" in *Proceedings of the 1982 summer Computer Simulation Conference* pp. 13-16, Denver. (July 1982).

[Karplus82b] Karplus W. J. "Software for distributed system simulation" pp. 293-308 in *Progress in modelling and simulation*, ed. Cellier F. E., Academic Press (1982).

[Kernighan78a] Kernighan B. W. and Ritchie D. in *The C programming language*, Prentice-Hall (1978).

[Kernighan81a] Kernighan B. W. "Why Pascal is not my favorite programming language" in *Computing Science Technical Report No. 100*, Bell-Laboratories, July 18. (1981).

[Kettenis83a] Kettenis D. L. "The COSMOS modelling and simulation language" pp. 249-260 in *First european simulation congress ESC 83*, Aachen. (Sept. 1983).

[Kettenis86a] Kettenis D. L. "The Simulation Environment of the COSMOS-language" in *European Simulation Meeting*, Boeblingen. (May 1986).

[Kettenis86b] Kettenis D. L. "Cosmos: A Member of a New Generation of Simulation Languages" pp. 263-269 in *Proc. of the 2nd European Simulation Congress*, Antwerp, Belgium. (Sept. 1986).

[Knuth73a] Knuth D. E. Vol. 1 in *The art of computer programming*, ed. Varga R. S., Addison-Wesley Company (1973).

[Korn87a] Korn G. A. "A new software technique for sub-model invocation" in *Simulation* Vol. 48 (3), pp. 93-97 (March 1987).

[Kreutzer86a] Kreutzer W. in *System simulation: programming styles and languages*, ed. McGettrick A.D. and Leeuwen J., Addison-Wesley Publishing Company (1986).

[Ledgard81a] Ledgard H. in *ADA An Introduction and Ada Reference Manual*, Springer-Verlag (1981).

[Lehmann87a] Lehmann A. "Taxonomy and Application of Expert Systems in Simulation" pp. 1-7 in *Preprints of Intern. Symp. on AI, Expert Systems and Languages in Modelling and Simulation* (June 1987).

[Lesk75a] Lesk M. E. "Lex - A Lexical Analyzer Generator" in *Computing Science Technical Report N. 39*, Bell Laboratories, Murray Hill, NJ. (March 1975).

[Locke56a] Locke J. "An Essay Concerning Human Understanding" pp. 33 in *The Age of Enlightenment*, ed. Berlin I., New York. (1956).

[McCormack83a] McCormack J. and Gleaves R. "MODULA-2. A Worthy Successor to Pascal" in *BYTE Publications Inc.* pp. 385-395, Del Mar, CA. (April 1983).

[Mitchell81a] Mitchell E. E. L. and Gauthier J. S. "ACSL User Guide/Reference Manual", Mitchell and Gauthier, Assoc., Inc. (1981).

[Mitchell82a] Mitchell E. E. L. "Advanced continuous simulation language (ACSL) : an update" pp. 462-464 in *Proc. 10th IMACS world congress*, Montreal. (1982).

[Mitchell84a] Mitchell E. E. L. and Gauthier in *ACSL Newsletter* (1 dec 1984).

[Nilsen82a] Nilsen R. N. "Macros Make More Meaningful Models" in *Proceedings of the 1982 summer Computer Simulation Conference* pp. 17-23, Denver. (July 1982).

[Nilsen83a] Nilsen R. N. "CSSL-IV : a software system for computer aided analysis" in *Advances in simulation software technology (meeting)*, Atlanta, Georgia. (October 20-21, 1983).

[Ören79a] Ören T. I. and Zeigler B. P. "Concepts for Advanced Simulation Methodologies" in *Simulation* Vol. 34 (3), pp. 49-82 (March 1979).

[Ören79b] Ören T. I. "Concepts for Advanced Computer Assisted Modelling" pp. 29-55 in *Methodology in Systems Modelling and Simulation* (1979).

[Ören84a] Ören T. I. "GEST - A modelling and simulation language based on system theoretic concepts" pp. 281-335 in *Simulation and Model-Based Methodologies: An Integrative View*, Ören T. I. (1984).

[Reiss87a] Reiss S. P. "Automatic compiler production: the front end" in *IEEE transactions on software engineering* Vol. SE-13 (6), pp. 609-627 (June 1987).

[Richards78a] Richards C. J. "What's Wrong with my Model" pp. 223-228 in *2nd. UKSC Conference*, United Kingdom. (1978).

[Riera85a] Riera J. "Anàlisi i modelització de la regulació frequencia-potencia de sistemes elèctrics interconectats" in *Tesi doctoral UPC* (Octobre 1985).

[Rigas76a] Rigas H. B. "Software for Hybrid Computation-Past,Present and Future" in *Computer* Vol. 9 (7), pp. 26 (July 1976).

[Runge77a] Runge T. F. "An Universal Language for Continuous Network Simulation" in *Ph. D. Thesis*, University of Illinois at Urbana-Champaign, Urbana, Illinois. (1977).

[Sedgewick83a] Sedgewick R. in *Algorithms*, Addison-Wesley Publishing Company (1983).

[Selfridge55a] Selfridge R. G. "Coding a General-Purpose Digital Computer to Operate as a Differential Analyzer" pp. 82-84 in *Proc. Western Joint Comp. Conf.* (1955).

[Shah76a] Shah M. J. "Engineering Simulation Using Small Scientific Computers", Prentice-Hall (1976).

[Smart84a] Smart P. J. and Baker N. J. C. "SYSMOD - An environment for modular simulation" pp. 77-82 in *Proc. 1984 Summer Computer Simulation Conference*, Boston. (July 1984).

[Stein60a] Stein M. L. and Rose J. "Changing from Analog to Digital Programming by Digital Techniques" Vol. 7 pp. 10-23 in *JACM* (1960).

[Strauss67a] Strauss J. C. "The SCi Continuous-System simulation language" in *Simulation* Vol. 9 (6), pp. 281-303 (Dec. 1967).

[Symons86a] Symons A. "Summary of some current issues" in *TC3-IMACS Simulation Software Committee Newsletter No. 86/01* (January 1986).

[Symons87a] Symons A. "Model Structuring Philosophies" in *TC3-IMACS Simulation Software Committee Newsletter No. 87/01* (January 1987).

[Syn85a] Syn W. M. and Dost M. H. "On the dynamic simulation language (DSL/VS) and its use in the IBM corporation." in *11th IMACS World Congress*, Oslo. (August 1985).

[Tarjan72a] Tarjan R. "Depth-First Search and Linear Graph Algorithms" in *SIAM J.* Vol. 1 (2), pp. 146-160 (June 1972).

[Thompson85a] Thompson S. "Rootfinding and Interpolation with Runge-Kutta-Sarafyan Methods" in *Transactions of the Society for Computer Simulation* Vol. 2 (3), pp. 207-218 (September 1985).

[Troch86a] Troch I. "There is a Need for Simulation Based on Implicit Models" pp. 157-162 in *Proc. of the 2nd European Simulation Congress*, Antwerp, Belgium. (Sept. 1986).

[Vaucher84a] Vaucher J. "Futute directions in simulation software (panel)" Vol. 13 (2), pp. 122 in *Simulation in strongly typed languages: ADA, PASCAL, SIMULA...*, ed. Bryant R. and Unger B. W., Society for Computer Simulation, San Diego, California. (February 1984).

[Wirth83a] Wirth N. pp. 10-11 in *Programming in Modula-2*, Springer.Verlag (1983).

[Wolverton86a] Wolverton F. B. and Hedrick E. L. "Code optimization speeds C throughput" in *Computer Design* pp. 117-120 (November 1986).

[Wymore76a] Wymore A. W. in *Systems Engineering Methodology for Interdisciplinary Teams*, John Wiley, New York. (1976).

[Zeigler76a] Zeigler B. P. "Theory of Modelling and Simulation", John Wiley and Sons, New York. (1976).

[Zenor87a] Zenor J. J. and Zenor J. J. "Software engineering for real time simulation environments" pp. 143-151 in *Preprints of Intern. Symp. on AI, Expert Systems and Languages in Modelling and Simulation*, Barcelona. (June 1987).

Index

Emphasized items are original terms introduced in this thesis.

ACSL 213
Artificial intelligence 144
CAMAS 48, 213
COSMOS 12, 14, 18, 213
COSY 12, 14, 213
CSMP 213
CSSL 213
DEC 213
DISCO 13, 14, 213
ESL 13, 14, 213
GEST 13, 14, 213
IMACS 213
LEX 148, 213
MACRO 10, 17
MCL commands 149, 150
  calculator 151
  comment line 152
  control 152
  create 132, 153
  do 132, 153
  edit 153
  exit 155
  external file 156
  help 156
  parameter 157
  prepare 157
  print 158
  remove 132, 159
  scope 159
  set 132, 160
  show 132, 161
  type 132, 163
MCL grammar 147
MCL 110, 131, 149, 213
MUSS 213
MUSS architecture 7
MUSS grammar 147
MUSS simulation system 4
ODE segment digraph 68, 87
ODE segment 38, 68, 87, 129
ODE 213
PI controller submodel 184
SCS 213
SIDOPS 48, 214
SMPR circuit submodel 206
SMPR circuit 171
SMPR 214
SYSMOD 12, 13
SYSMOD 14
TC3 214
YACC 147, 214
address() 122
algebraic segment digraph 89
algebraic segment 89, 94, 113
analysis, phases 67
analysis, submodel digraph 66
analysis, submodel 106
biplanar representation 34, 43
block, experiment 18, 8, 111, 131, 166
block, study 8, 18, 20, 131, 166

block, submodel 8, 109, 111, 166
calling sequence 25
combined simulation 145
complete graphs 49
connected graph 48
constant symbol vertex 50
constr. rules 53
  declarative assig. st. 59
  if stat. 62
  initial region 56
  static region 56
  when stat. 60
continuous process 111
control circuit submodel 197
control circuit 174
control_circuit derivative segment 128
control_circuit discontinuous segment 128
control_circuit state segment 128
control_circuit submodel 123
coupled model 11
definition digraph 113, 117
definition digraph 125
definition routine 113, 117, 119
dense graphs 49
derivative executable vertex 51
derivative segment digraph 89
derivative segment 113, 89
derivative segment 94
derivative symbol vertex 50
dialog level 111
digraph,
  ODE segment 68, 87
  algebraic segment 89
  definition 113, 125
  derivative segment 89
  discontinuous segment 68, 82
  initial segment 68, 78
  reduced 90
  segment-link 43, 54
  segment 68
  state segment 89
  submodel 17, 33, 42, 50, 108
digraph 48
directed acyclic graphs 49
directed cycle 48
directed path 48
discontinuous executable vertex 51
discontinuous functions 82
discontinuous segment digraph 68, 82
discontinuous segment 38, 68, 82, 113, 129
distributed parameters submodel 16, 146
dynamic continuous region 15
dynamic initialization 28, 74, 110, 124
dynamic sampled region 16
dynamic sequence 113
environment generator 130
executable vertex 50, 59
  derivative 51
  discontinuous 51
  if 51, 63
  initial 51, 56
  state 51
  when 51, 60
executable vertex
exp_instance data structure 131
exp_table data structure 130, 134
experiment block 18, 8, 111, 131, 166
experiment instance 131
filter submodel 181
flat model 11
fused vertices 92
fusion 90
generic model 113
global data 25
global symbol vertex 50, 56
grammar,
  MCL 147
  MUSS 147
hidden edges 44
hierarchical model 11
if executable vertex 51, 63

if statement 36, 54, 55
if symbol vertex 51, 64
information loops 34, 88
initial executable vertex 51, 56
initial segment digraph 68, 78
initial segment 38, 68, 78, 113, 124, 129
initial symbol vertex 50, 57
initialization sequence 113, 124
initialization,
    dynamic 28, 74
    static 26, 74
initpar() 122
input symbol vertex 50
input weight 90
integrator submodel 176
limiter submodel 187
local symbol vertex 50, 57, 61, 64
lower_submodels data structure 119
lumped parameters submodel 16
memory management 110, 113
metalanguage 147
model initialization 113
model instance 111, 113
model,
    flat 11
    hierarchical 11
    modular coupled 11
model 109
modular coupled model 11
modularity 109
non linear system submodel 39
non linear system 38
object oriented languages 146
output symbol vertex 50
output weight 89
parameter symbol vertex 50
paths 48
planar representation 34, 42
power circuit submodel 202
power circuit 172
pulse-width modulator submodel 76, 191
reachable vertex 49
real time simulation 145
real_pole submodel 115, 119
reduced digraph 90
reentrance 112
sampled submodel 16
scope of variables 25
second order submodel 40
segment digraph,
    ODE 68, 87
    algebraic 89
    derivative 89
    discontinuous 68, 82
    initial 68, 78
    state 89
segment,
    ODE 38, 68, 129
    algebraic 89, 94, 113
    derivative 89, 94, 113
    discontinuous 38, 68, 113, 129
    initial 38, 68, 113, 124, 129
    state 89, 94, 113
    terminal 129
segment-link digraph 43, 54, 96
segments 34, 38
semipath 48
simulation environment 9, 110, 175
simulation experiment 18
simulation program 8
simulation study 18
sort 38, 117
sparse graphs 49, 51
spring and mass system submodel 98
spring-and-mass system 97
state executable vertex 51
state segment digraph 89
state segment 89, 94, 113
state symbol vertex 50
static initialization 26, 74, 113
strongly connected graph 48
structured programming 109

study block 18, 20, 8, 131, 166
submodel analysis 106
submodel block 8, 14, 109, 111, 166
submodel data structure 113
submodel definition routine 122
submodel digraph analysis 66
submodel digraph construction 51
submodel digraph 17, 33, 42, 50, 108
submodel instance 110, 113, 116
submodel segment 53
submodel,
    PI controller 184
    SMPR circuit 206
    control_circuit 197, 123
    filter 181
    integrator 176
    limiter 187
    non linear system 39
    power circuit 202
    pulse-width modulator 76, 191
    real_pole 115, 119
    second order 40
    spring and mass system 98
submodel,
    distributed parameters 16
    lumped parameters 16
    sampled 16
submodels 10
symbol vertex, 50
    constant 50
    derivative 50
    global 50, 56
    if 51, 64
    initial 50, 57
    input 50
    local 50, 57, 61, 64
    output 50
    parameter 50
    state 50
    when 51
symbolic access 109, 112, 117
terminal segment 129
transformation rules 53
visit() 122
weakly connected graph 48
when executable vertex 51 60
when statement 37, 55
when symbol vertex 51