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

A la meva esposa Maria, pel seu ajut i encoratjament
Als meus pares, per tot el que he rebut d'ells

Acknowledgments

This work has been possible thanks to support got from a research grant from the *'Ministerio de Educación y Ciencia'* and from the *'Institut de Cibernètica' (IC).*

I also wish to express my appreciation to the *'Universitat Politècnica de Catalunya'* and to the *'Fulbright-La Caixa'* institution for its support to the scientific exchanges with the *University of Salford (UK)* and the *California State University at Chico.*

I am particularly indebted to Professor Rafael M. Huber, my thesis supervisor, who made valuable criticism and suggestions while carrying out this research and throughout the entire preparation and proofreading of this thesis. Moreover, during the nine years period I have been with the *IC*, he has provided continuous challenge and inspiration for the furthering of my research and work.

I am grateful to Professor Roy E. Crosbie for his accurate suggestions at the very outset of this work. In addition, I wish to thank the claiming *ICDSL* users and specially Professor Jordi Riera, the unquestionable leader of that group, for always asking me for new, knotty and more powerful features.

Thanks to my friends Joan Ilari, Lluis Perez and Jordi Valls for the exciting midday tennis games and to Xavier Ros for all the fantastic down-hill skiing days we have spent together. All of them have helped me to get rid of my stress.

I appreciate the encouragement from my colleagues at the *IC* and from my former colleague Joan Juan.

Finally, I express my gratitude to my wife Maria for her relaxing happiness and comprehension. There is a little doubt that without her constant encouragement this work would not have been possible.

Abstract

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

by

Antoni Guasch

In contrast with classical simulation languages, the present trends are evolving towards fully integrated interactive modelling and simulation environments. These environments have to combine interdisciplinary techniques such as AI, object oriented programming, data base management and system modelling.

To achieve the above objectives, the architecture of the simulation programming language and that of the run-time simulation environment (in this thesis, simulation environment) which exercises the models should be designed allowing modularity and flexibility. Furthermore, the robustness of the environment should be reinforced.

The thesis proposes the *MUSS* simulation system, emphasizing the innovative concepts put forward: the hierarchical architecture of the *MUSS* simulation language, the preprocessor analysis and segmentation phases and the structure of the run-time simulation environment and related management mechanisms.

A *MUSS* environment may enclose studies, experiments and sumbodels; continuous submodels may have initial, dynamic and static regions.

Submodels are translated to object code in isolation thanks to the proposed segmentation procedure which in turn is based on the defined submodel digraph concept. The segmentation methodology increases the modularity and flexibility of the system avoiding to restrict the general structure of the submodels. The submodel digraph is also used to test the consistency of the code.

Keywords : *combined models, continuous system simulation languages, data structures, declarative language, digraphs, LALR grammars, ODE computation methods, root finder algorithms, simulation languages, simulation environments, sorting.*

Thesis supervisor : Rafael M. Huber, Professor.

Contents

List of Figures

Chapter 1

Introduction

"I expect simulation languages to disappear in favour of interpretive interactive simulation environments with graphic capability. These systems will have expert capability and will often anticipate user directives and suggest alternatives. I also expect data bases to integrate many simulation models as images of reality."

[Vaucher84a]

1.1 Background and current trends

The early stages of the digital simulation languages date back [Selfridge55a] to the mid-fifties and were supported by the apparition at the beginning of the decade of the first generation of digital computers [Baer80a] and by the numerical calculus emerging field (figure 1.1).

Meanwhile, analog computers were widely used in the scientific community. Since the analog computers voltage is limited, usually to 10 or 100 volts, the scaling of the problem is a necessity [Jackson60a]. In the early sixties, simulation languages which emulate the behaviour of the analog computers were introduced to solve this arduous problem [Rigas76a]. This oriented application and the new features added to the simulation languages favoured the spread and development of the digital simulation techniques.

At that time, the FORTRAN and ALGOL scientific programming languages become popular, leading to more powerful simulation languages [Clancy65a] and therefore, in the late sixties, several languages were in the market. However, there was not an

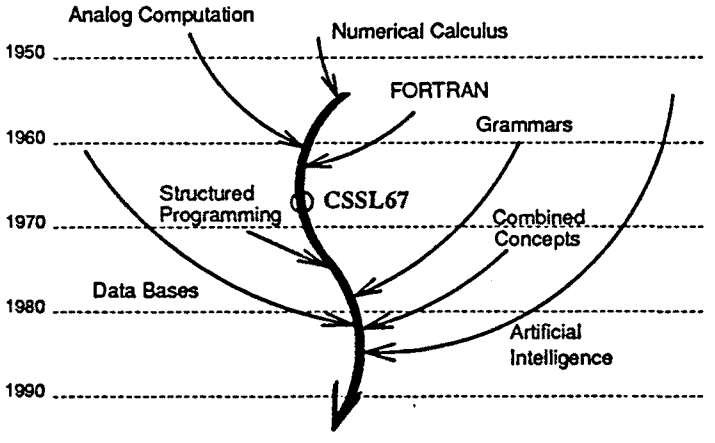

Figure 1.1: *Evolution of CSSL.*

agreement about the prevailing basic simulation features and the SCi Committee on Simulation Software was aware that some direction was needed to promote an orderly development in the field. In consequence, the conclusions of numerous meetings were agglutinated in the "SCi Continuous System Simulation Language" (referred herein as *CSSL67*) proposal [Strauss67a].

As a result, the family of *CSSL67*-like languages appeared. Without doubt, the SCi effort was worthy since nowadays the majority of the digital continuous system simulation languages available in the market still belong to this group.

Structured programming and automatic compiler generation techniques have been used in the decade of the seventies to increase the reliability and robustness of the simulation software [Cellier79b]. At the same age, the first combined —continuous/discrete— system simulation packages were released to the market [Hurst73a] [Cellier76a].

At the beginning of the present decade the necessity of introducing discrete features in the continuous system simulation domain comes forth in the scientific community. However, this aspect is still controversial, see [Crosbie82b] and the *TC3-IMACS Simulation Software Committee Newsletter N°12*, August 1984.

Furthermore, new simulation languages have given up the monolithic *CSSL67* structure and modular structures have been adopted. Those languages support higher modularity in, at least, three aspects: the programming of the simulation study itself; allowing program blocks consistent with the division of the physical system into subsystems;

and automatic building of the model by means of submodels.

At present, fields such as Expert Systems, Data Base Management Systems, Concurrent Programming and Object Oriented Programming are contributing to the development of *simulation environments* which, in contrast with the simulation languages, integrate the software support.

1.2 Motivations and objectives of the research

The research lines of the *Institut de Cibernètica (IC)* are strongly influenced by the demands of the industrial and scientific communities in Catalunya. Among them, those areas mainly related with simulation are Automatic Control (Electrical Engineering) and Bioengineering.

In the seventies, simulation in the *IC* was based on hybrid techniques [Huber82b]. Thereafter, the simulation group started to work in pure digital simulation languages. As a result, the *ICDSL* CSSL-like simulation language was designed [Guasch84a].

About 1983, the lack of modularity of the *ICDSL* language was noticed as a significant constraint in the simulation of large systems. Therefore, the design and implementation of the Modular Simulation System *MUSS* got under way.

A hierarchical architecture sustaining a modelling, coding and testing bottom-up approach has been chosen and the research guide line has been to provide the theoretical and practical background needed to support a modular structure without restricting the following general objectives:

- The language should be declarative and should manage a modularity coherent with the division of the physical system into subsystems through a minimal but sufficient number of different blocks.
- The separation between the model description and the experiment should be done in such a way that the model remains unchanged along the experiments and ready to be used from another model (submodel) at the end of the validation and verification experiences.
- It should be possible to build a system model in a bottom-up way relating two or more submodels from a model or submodel of higher hierarchical level.
- Isolated preprocessing of submodels as well as run time symbolic access to all the variables should be able.

- The language should be designed in order to be easily extendible to real time applications.
- The usual design engineering steps concerning a prototype which has to work on-line with an existing hardware system should be liable to be performed in the following sequence:
  - Hardware system modelling and model validation.
  - Prototype modelling.
  - Whole system model evaluation.
  - Prototype building.
  - Evaluation of the physical prototype on line with the model of the hardware system.
  - Evaluation of the physical prototype on line with the hardware system.

The *MUSS* architecture proposed joins the object oriented language concept and has been conceived having in mind to support in a future concurrent programming and AI reasoning.

1.3 MUSS, towards an integrated simulation system

The MUSS simulation system includes the following modules (figure 1.2).

- *Model definition languages:*

  Simulation models may by specified using different techniques such as CSSL-like languages and bond graphs.

  MUSS will allow several model definition languages besides the native one.

- *Translator:*

  It embraces two modules: the *preprocessor* which translates MUSS source code into C language code and the *C compiler* which translates the C code to object code. The MUSS preprocessor stores and gets information from the model data base.

- *Model data base:*

  Stores information about preprocessed and compiled submodel, experiment and study blocks. Compiled blocks are stored in system or user defined object libraries.

- *Environment generator:*

  Using this module, the user may chose the set of experiments and studies to be included in the user defined interactive simulation environment.

- *Simulation environment:*

  An interactive monitor controls the execution of selected studies and experiments.

  Simulation results are stored in a simulation data base.

- *Simulation data base:*

  Stores simulation results and experimental results.

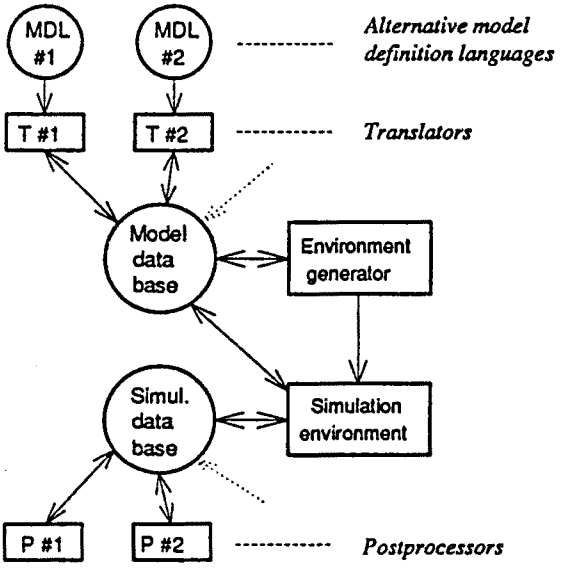

Figure 1.2: *General architecture of the MUSS system.*

- *Postprocessors:*

  Postprocessors are used to analyze simulation results stored in the data base or to store experimental results in the data base.

1.4 Organization

The general structure of this thesis is as follows :

This chapter is a review of the historical evolution as well as the present state of continuous system simulation languages. Furthermore, it describes the MUSS simulation system proposed. The research outcome is mainly centered in two system modules : the *MUSS* preprocessor and the *MUSS* simulation environment plus the *MUSS* model definition language.

Chapter 2 describes the architecture of a *MUSS* program. Only three type of blocks have been defined in contrast with other simulation languages which use a large set of blocks, being some of them defined because of programming requirements instead of methodological ones.

In **Chapter 3**, the submodel digraph concept on which the proposed continuous submodel analysis and segmentation algorithms rely is defined and its characteristics are outlined.

In **Chapter 4**, the structure of the simulation environment is presented. The user selects the set of models, study and experiment blocks to be included in the environment. Although *MUSS* program blocks can be preprocessed and compiled in isolation, variables from any block can be accessed symbolically as well as block information stored in the model data base.

Chapter 5 contains the conclusions derived from the theoretical developments and the obtained experimental results. Possible improvements are discussed and open future research lines are pointed out.

In **Appendix A**, the metalanguage used to specify in Appendix B the *MUSS* language and in Appendix C the *MCL* language is presented. *MUSS* and *MCL* are lookahead-left-to-right languages.

In **Appendix B** the *MUSS Command Language —MCL—* is introduced and its grammar rules are listed. This language has been designed to allow users to interact with the simulation environment.

Appendix C contains a simulation example to illustrate the top-down modelling and bottom-up coding and testing approach which is strongly supported by the *MUSS* simulation environment. The analysis and segmentation performed to each submodel is outlined.

Finally, **Appendix D** contains a short list of used acronyms.

Chapter 2

MUSS architecture

"Conventional simulation techniques have three shortcomings when applied to large-scale modelling: They provide an inadequate man-machine interface, they provide a poor conceptual framework, and they lack needed tools for managing data and model. These shortcomings may be ameliorated by developing a new simulation languages that differentiate the functional elements of a simulation program and by recognizing the goals of these functional elements."

[Ören79a]

2.1 Introduction

The architecture of the MUSS language and related language constructs converge with the trends on piecewise-continuous system simulation languages **[Crosbie82a]** **[Hay85a]** and with the state of the art on combined simulation languages **[Cellier79c]** **[Smart84a]** **[Ören84a]** **[Kettenis86b]**.

Although the *MUSS* language has been designed to initially support the continuous time modelling formalism, the simulation environment has been conceived to easily expand the language to combined models. As an example, the *class concept* which allows the generic instantiation of processes is supported although at present, only continuous processes are handled.

The formal design is carried out using syntax analysis tools **[Lesk75a]** **[Johnson75a]**.

2.2 Architecture

2.2.1 Simulation program

The definition of a *MUSS* simulation program is as follows[1]:

```
MUSS_program
        : set_of_MUSS_blocks
        ;

set_of_MUSS_blocks
        : MUSS_block
        | set_of_MUSS_blocks MUSS_block
        ;

MUSS_block
        : study_block
        | experiment_block
        | submodel_block
        ;
```

A *MUSS* program is composed of a set of blocks whose structure guarantees the proposed objective. Three types of blocks may be present in a program:

- *submodels:* provide the user with mechanisms to describe a physical subsystem. A submodel block may call and may be called by none, one or more submodel blocks. A *MUSS* simulation model is composed by a set of submodels. A model is a relative concept which depends on the experiment block being executed.
- *experiments:* control the execution of a single evolution. None, one or more models can be called from the dynamic region of an experiment. In the experiment block, mechanisms for performing a set of evolutions —multi-run study— are not provided. Experiments can not call one each other.
- *studies:* A study block controls the execution of a set of evolutions —experiments—. One or more experiments can be invoked from the dynamic region of the study.

A program alone does not have necessarily to define a complete environment, neither a study or experiment ready to be executed, neither a model. The set of

[1]The metalanguage used to define the *MUSS* language is specified in appendix A

preprocessed and compiled blocks belonging to a program are put together in a chosen library. Later on, a given environment will be set up by the environment generator which selects study or/and experiment blocks from object libraries.

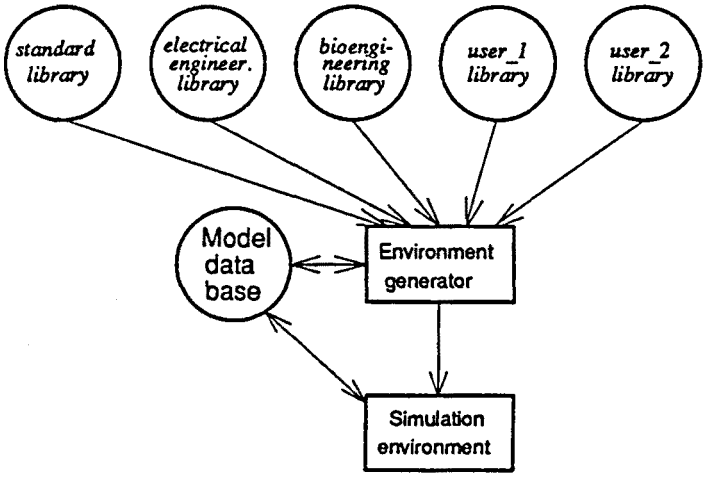

Figure 2.1: *The model data base keeps information about preprocessed and compiled submodel, experiment and study blocks. Compiled blocks are stored in system or user defined object libraries.*

The *MUSS* system is flexible enough to allow users to create and share oriented libraries. For example, the Institut de Cibernètica uses simulation in two main fields: Electrical Engineering and Bioengineering. Therefore, besides the standard simulation library, it is possible to define two libraries, the first one oriented to the simulation of Electrical Engineering models and the second one oriented to the simulation of Bioengineering models. Moreover, every user may define his own library (figure 2.1).

Using the environment generator module the user may select the set of experiments and studies[2] to be included in the user defined simulation environment. Selected blocks may be stored in different libraries. Furthermore, a block from a given library may call blocks stored in other libraries.

An example follows, lets suppose that *program A* from figure 2.2 is preprocessed, compiled and included into the standard simulation library (figure 2.1). An environment can be defined by selecting *experiment_1* and *experiment_2* experiments from the

[2]When an experiment or a study is flagged for its inclusion in a simulation environment, all the called lower level blocks are implicitly selected.

standard library. Other environments would have been defined selecting *experiment_1* or selecting *experiment_2*.

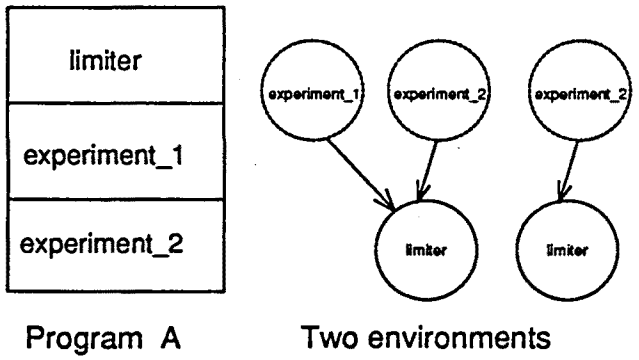

Figure 2.2: *User defined simulation environments selected from the blocks in program A.*

Notice that the limiter submodel block has to be preprocessed before higher level blocks calling it. This is extensible to all program blocks.

To continue the example, in figure 2.3 it is shown how new blocks from *program B* might be added to the environment after being preprocessed, compiled and stored in the *user_1* library.

2.2.2 Submodels

Most of the commercial currently available continuous system simulation languages are based on the SCi CSSL report [Strauss67a]. In its implementation the most important part is concerned with numerical algorithms while the programming structures are relatively poor [Brennan68a] [Chu69a]. In these languages, the only way to achieve modularity consists on the use of MACRO pseudoblocks or preprocessor target language subroutines (most often, FORTRAN subroutines). Although the use of MACRO blocks as a basic element to achieve model modularity still have adepts [Nilsen82a] [Breiteneck83a], the general feeling is that MACROS are still needed because independent translation of submodel code is not always possible [Cellier79a] [Baker83a] [Mitchell84a] [Freeman84a] [Korn87a].

In contrast, new simulation languages offer a higher level of modularity in, at least, two aspects :

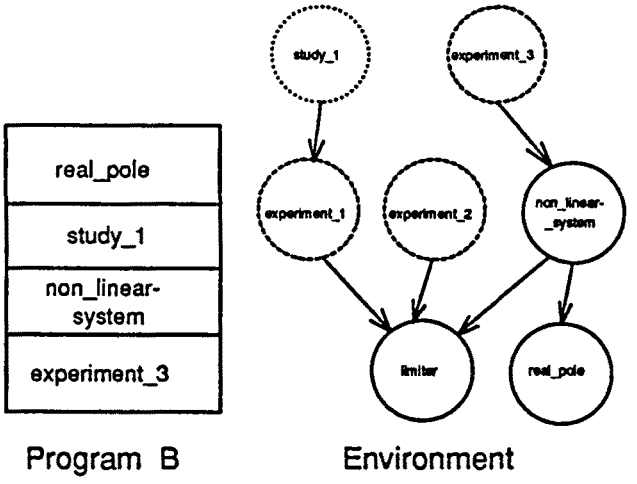

Figure 2.3: *Blocks from new programs can be added to a previous user defined simulation environment. A user defined simulation environment may include a large set of study, experiment and model blocks.*

- They allow program blocks consistent with the division of the physical system into subsystems.
- They allow the automatic building of the model by the use of submodels.

Oren [Ören79b] defined the concept of *modular coupled model.* A modular coupled model consist of several submodels where coupling specifications define the input/output relationships between submodels. Depending on the disposition of each submodel relative to other submodels, two types of modular coupled models can be defined:

- *Hierarchical model:* several coupled submodels where at least one of the submodels is itself a coupled model.
- *Flat model:*[3] several coupled submodels. The submodels are not themselves coupled models.

[3]The flat model concept proposed here is different from the flat model concept presented by Symons [Symons86a], [Symons87a].

Comparative study

Different blocks have been proposed to code submodels and a first remark is that the bracket structure statements are misleading because some of them may have different meaning according to the different languages. For example, the MODULE concept in *COSY* [Cellier79c] is quite different from the MODULE concept in *SYSMOD* [Smart84a]:

- In *COSY*, a MODULE block must be stored in source form. In its definition, formal parameters need no longer be separated into input and outputs; upon usage the mapping of the actual to formal parameters is specified by explicit assignments; besides replacement of parameters, it uses formulae manipulation to reorganize the statements. This concept was formulated in [Runge77a] and in [Elmqvist78a].
- *SYSMOD* uses the MODULE concept for SUBMODEL invocation when information loops are found. When a submodel is called as a MODULE it is handled as a MACRO. The MODULE invocation ensures that compatibility with SUBMODELS is maintained at run-time.

Up to now, there is not a general agreement about the basic blocks that a simulation language should have in order to achieve modularity in the model description. What follows is a description of the blocks of some modern simulation languages. Only the blocks that support the piecewise continuous modelling formalism are described:

COSMOS [Kettenis83a] [Kettenis86a] :

- CONTINUOUS PROCESS : describe processes of entities whose behaviour is governed by differential equations. Each continuous process may have an *initial*, a *continuous* and a *terminal segment*. Statements written in the continuous segment are sorted by the compiler.
- SYSTEM : The SYSTEM-segment is used to specify the model. The parameter passing between processes is derived from *GEST* and will be given in a COUPLING-segment in the SYSTEM-segment of a simulation program. In the literature read there is not information about how information loops in the COUPLING-segment are handled.

COSY [Cellier79c] :

- MODEL : The concept of MODEL is similar to the PROCED pseudoblock proposed in the CSSL definition. Contrary to the PROCED construct, the

statements of the model are parallel code. MODELs can be preprocessed and used as full-MODELs.

- MACRO : This facility is often needed to obtain a sorted executable sequence of statements. MACROs should be stored in source form.
- MODULE : It is more general than the MACRO definition in two senses :
  1. Besides replacement of parameters, it uses formulae manipulation to reorganize the statements.
  2. A MODULE definition may contain an *initial block*, a *terminal block* and a *discrete block* in addition to the *continuous block*.

DISCO [Helsgaun80a] :

- CONTINUOUS PROCESS : In a continuous process the order in which the equations have to be solved is left to the user because *DISCO* does not change the execution order of the equations. The user has to determine the order of evaluation within each continuous process, and the continuous process themselves have to be ranked by giving each a priority.

ESL [Crosbie83a] [Hay85a] :

- SUBMODEL : In *ESL*, a SUBMODEL may call lower level submodels and may be called from higher level submodels or from a MODEL. Sorting algorithms have been implemented lately in the language but up to now we do not know their characteristics and restrictions.
- MODEL : The model provides the user with the ability to describe the physical system. SUBMODEL sorting notes are applicable to the MODEL dynamic code.

GEST [Ören84a] :

- COMPONENT MODEL : In *GEST*, a model may be composed by a number of component models. A component model may be *continuous*, *memoryless* or *discrete*. Each component model is composed by a number of other component models; the coupling may be hierarchical.

SYSMOD [Baker83a] [Smart84a] :

- MODEL : In *SYSMOD* a model is a relative concept and it has a hierarchical structure. The SUBMODEL name is used for calling lower level MODEL blocks. The statements of a MODEL are parallel code and can be preprocessed, compiled and stored in libraries.

  When sorting parallel code within a MODEL, the sorter, which has no knowledge of the internal structure of lower level MODELs, may report

information loops. In such a case, lower level MODELs in the loop must be called using the MODULE clause instead of the SUBMODEL one. The MODULE invocation forces the preprocessor two handle the called MODEL as a MACRO.

Several aspects can be stressed about the above simulation languages:

- *COSY, ESL, GEST* and *SYSMOD* use the hierarchical modelling approach where as *COSMOS* and *DISCO* use the flat modelling approach.
- *COSY* and *SYSMOD* use MACRO-like facilities to obtain a sorted executable sequence of the code when information loops have to be avoided. *DISCO* does not sort the sentences, leaving this job to the users.

  GEST uses a much more formal approach, based on the axiomatic theory of Wymore [Wymore76a]. In *GEST* a component model may be continuous, discrete or memoryless. The dynamic structure of a continuous component model is divided in two segments, the *derivative segment* and the *output segment.* This formal separation between the derivative and the output segments allows the sorting of coupled continuous submodels. In contrast, *GEST* does not take in account that logical states associated to discontinuous functions are in fact state variables [Horst86a].

MUSS submodel blocks

MUSS uses the hierarchical modelling approach. The structure of *MUSS* submodel blocks is represented in figure 2.4. Submodel blocks may be translated in isolation which helps to make the modelling turnaround time shorter than languages that need to translate all the submodel and experiment blocks each time a change is made on a submodel or experiment block.

The submodel block consists of the classical two regions, an *initial region* and a *dynamic region* plus a *static region* which is equivalent to the *static structure* of *GEST* [Ören84a]. The static region is described basically in terms of model descriptive variables such as state, input and output variables and constants and parameters.

Although the inclusion of the classical *terminal region* has been rejected because the calculation to be performed when a finish condition is meet can always be included in the experiment, its incorporation in the submodel block can be done without restricting the proposed objectives (page 3).

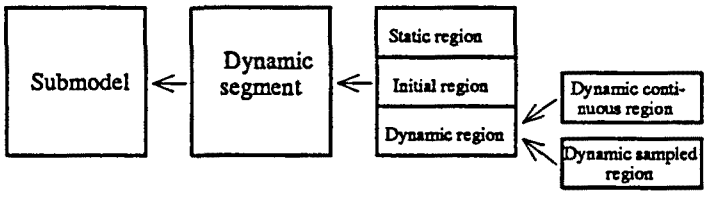

Figure 2.4: *Submodel.*

The type of the equations coded in the *dynamic* region characterizes the nature of the submodel as *continuous* (lumped or distributed parameters) or *sampled.*

The invocation of the contributory submodels is done from the dynamic region.

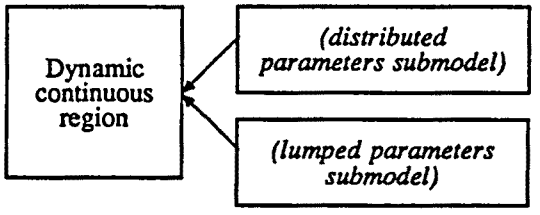

Figure 2.5: *Dynamic continuous region.*

Dynamic continuous region

If the syntax of some representation statement corresponds to a partial differential equation, the whole model block will be considered as representing a distributed parameter submodel.

The whole dynamic continuous region is declarative (non procedural). The preprocessor clusters code which need not be executed in every integration step but only at each communication interval because it is able to detect the variables not contributing to the computation of derivatives, discontinuous functions or variables in the output string. Hence, it makes unnecessary to define derivative segments.

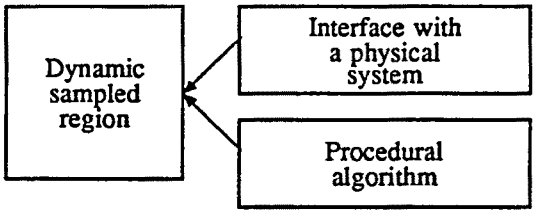

Figure 2.6: *Dynamic sampled region.*

Dynamic sampled region

It is characterized by the fact that it communicates with the other dynamic regions only at fixed intervals.

Its dynamic section can be of two kinds:

- A procedural algorithm representing submodels of computer based controllers sampling some continuous system variables and sending back control commands at fixed time intervals.
- An interface with a hardware subsystem. This kind of dynamic section differs from all the others in two main aspects: it depends on the hardware configuration of the host computer and it requires real time computation [Cellier84a].

Conclusions

Four types of submodels have been described, depending on the nature of the dynamic region:

- Lumped parameters submodel.
- Distributed parameters submodel.
- Sampled submodels representing controllers.
- Sampled submodels that interface with a hardware subsystem.

In this thesis, we will focus in the analysis of lumped parameters submodels with discontinuities. As a remark, sampled submodels representing controllers can easily be implemented in the system but to include in the environment sampled submodels that interface with a hardware system and distributed parameters submodels requires a depth analysis [Crosbie82d].

The structure of the lumped parameters submodel block is not innovative, it follows the classical pattern of Continuous System Simulation Language models. The main important contributions are derived from the analysis done at preprocessing time, they are:

- *Code robustness:*

  The formalization in the next chapter of the *submodel digraph* concept constitutes a powerful base for improving the robustness of the model code in a way still not achieved in present simulation languages.

- *Declarative code:*

  MACRO-like facilities or a strict separation between derivative computation and output computations are not needed to solve the problem of information loops in the model code. Through a formal analysis of the submodel digraph, the submodel code is automatically sorted and partitioned into a set of segments. This segmentation avoids information loops and opens the door to work without restriction with libraries of precompiled submodels.

- *C Target code with a neat structure:*

  The following text reproduced from [Clancy65a] is taken as an introduction to this point.

> *The field of digital simulation languages, although barely ten years old, has shown a remarkable growth and vigor. The very number and diversity of languages suggests that the field suffers from a lack of perspective and direction...*
>
> *In Locke's words, "everyone must not hope to be ... the incomparable Mr. Newton..., it is ambition enough to be employed as an under-labored in clearing the ground a little, an removing some of the rubbish that lies in the way to knowledge* [Locke56a]".

  Nowadays, more than thirty years later from the beginning of the digital simulation languages field, the situation is similar. Moreover, the number of directions is much greater than before. Fields such as Artificial Intelligence, Data Base Management Systems, Compiler Design and Abstract Data Types are increasing

their influence in the development of simulation languages. Related with it, in our opinion, simulation languages designers must use existing algorithms and packages designed by experts from the different fields they need, and integrate them in a structured manner to develop powerful simulation environments.

Following this strategy and focusing our attention in integration and discontinuity handling packages. Reputed packages, as for example LSODAR (ODEPACK) [Hindmarsh82a], distinguish code needed to compute derivatives from code needed to computed discontinuous functions; in fact, the code must be stored in different routines. A fruitful work done by the submodel digraph segmentation algorithm consists on separating code for derivative computations from code for discontinuity function computations.

2.2.3 Experiment

A simulation experiment is defined as *a simulation run over a period of time from a known initial frame* [Symons86a]. Unlike currently developed simulation languages, the *MUSS* experiment block monitors the execution of a single simulation run in contrast with the other languages whose experiment descriptions may monitor the execution of a set of runs.

We rather distinguish between a simulation experiment and a simulation study. A simulation study is defined as *a set of related experiments* [Symons86a]. The study block described in the next section provides the mechanisms to perform a set of related experiments.

A model without an experiment can not be executed. Even though an experiment may be called by a study. The experiment can always be optionally executed in independence with respect to the study.

In the most general case an experiment block may have three segments: a dynamic segment, a control segment and an output segment

Dynamic segment

The *dynamic segment* has the classic three regions : *initial*, *dynamic* and *terminal* regions plus a *static* region.

As in *COSMOS* [Kettenis83a], its aim is to allow the inclusion in the experiment of the dynamics external to the system model —but in most cases necessary for its

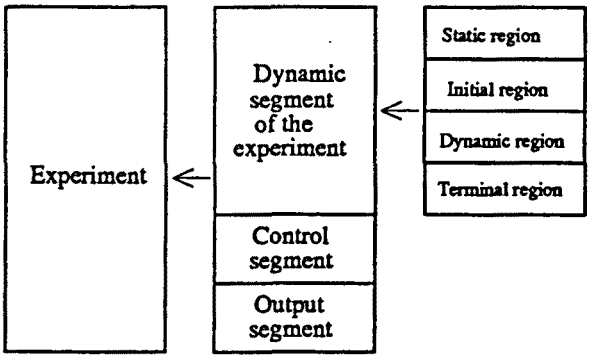

Figure 2.7: *Experiment block.*

evolution— or continuous cost function computations. Without this segment it would be necessary to handle the model code for that purposes.

In some simple studies this segment could represent the system model but this is not at all its purpose. The only difference from the dynamic segment of submodel blocks is that a *terminal region* is provided to specify the calculations to be performed when a finish condition is met. For example, in the pilot ejector study, a warning message can be written from the *terminal region* if the pilot will strike the vertical stabilizer. The terminal region is not intended for controlling the execution of a set of evolutions. This task corresponds to the study block.

The *static region* of the experiment block is equivalent to the *static region* of the submodel block.

In most frequent situations, one or more models will be called from the dynamic region, the latter option opening the possibility of simultaneously performing the same experiment on different independent models.

Control segment

The *control segment* is intended for simulation run parameters (in short, simulation parameters) specification such as initial time, and finish time, and termination conditions. From this segment it is also possible to set model parameters.

Output segment

The *output segment* clusters the output control statements. Its objective is to make explicit the variables whose evolution has to be recorded.

The control segment, as well as the output segment, are stored in the model data base at preprocessing time and it is allowed to edit both segments from a user defined simulation environment. When creating an experiment instance, the control segment and the output segment are executed. The simulation user has the option to modify control and output specifications before invoking a simulation experiment (single run).

2.2.4 Study

The study block monitors the execution of a set of experiments —simulation study—. Usually, the study block will be called from the *MUSS* simulation environment. The study block may be optionally called from sophisticated main programs coded by the users. Moreover, the study itself may be supplied by the user in C target code, which in turn, calls the experiment blocks through a clear set of interfacing routines.

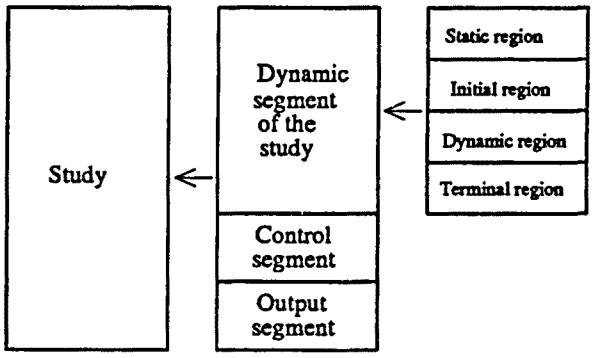

Figure 2.8: *Study block.*

Like the experiment block, in the most general case a study block may have three segments: a dynamic segment, a control segment and an output segment

Dynamic segment

The *dynamic segment* has four regions : *initial, dynamic, terminal* and *static.*

Unlike the *dynamic region* of experiment or submodel blocks, the *dynamic region* of the study block is procedural. Therefore, it will not be sorted by the preprocessor.

Static region

Like the *static region* of the submodel block or the experiment block, it is used for declaring constants and variables.

The main differences with respect to the submodel and experiment static blocks are :

- In the study block state variables are not allowed.
- The names of experiment instances may be declared in the static region although they can also be symbolically passed through the input list in a call sequence to the study.

Initial region

The *initial region* describes the procedural computations to be performed for initializing the study. The statements written in this region will be executed only once.

Dynamic region

It describes the computations to be performed between experiments by means of procedural statements. The execution of experiments is sequential and not parallel. Therefore, when an experiment is called for execution, the execution of the study is frozen until the simulation run finish.

Any experiment block present in the simulation environment which has been declared in the static region or symbolically passed through the input list when invoking the study can be called for execution from this region.

The possibility of passing symbolically the experiment name to the study, opens the door to coding general studies such as parameter sweeping, parameter optimization or boundary condition problems whose code is independent with respect to the dynamics and outputs.

MUSS simulation, model and experiment parameters may be updated from the *initial, dynamic* and *terminal regions* of the study.

Terminal region

It describes the actions to be done after the execution of the study. It can be used for documentation purposes.

Control segment

The *control segment* specifies simulator parameters such as initial time, finish time and termination condition specification. Remember that simulation parameters can also be set in the dynamic segment of the study. It is also possible from this segment to set model, experiment and study parameters.

The control segment, as well as the output segment, are stored in the model data base at preprocessing time and it is allowed to edit both segments from a user defined simulation environment. The updated segments may be stored for later executions.

Output segment

The *output segment* clustering the output control statements. Its objective is to make explicit the variables whose evolutions has to be recorded.

When creating a study instance, the control segment and the output segment are executed. The simulation user has the option to modify control and output specifications before invoking a simulation study.

2.2.5 Example

Van der Pol's equation

$$\frac{d^2y}{dt^2} + \mu(y^2 - 1)\frac{dy}{dt} + y = 0$$

can be investigated in a straightforward manner. Let

$$\frac{dy}{dt} = y_1$$

Then

$$\frac{dy_1}{dt} = -y - \mu(y^2 - 1)y_1$$

and the *Van_der_Pol* submodel may be written

```
Continuous submodel Van_der_Pol is
      Static region
            outputs (real y;)
            state (real y,y1;)
            parameters ( real mu = 0.0, y0 = 1.0,
                         y10 = 1.0;)
      End static region;
      Initial region
            y  = y0;
            y1 = y10;
      End initial region;
      Dynamic region
            y'  = y1;
            y1' = -y - mu*(y**2.0-1.0)*y1;
      End dynamic region;
End submodel Van_der_Pol;
```

Code 2.1 *Van_der_Pol submodel.*

In order to perform the power spectral density analysis of Van der Pol's equation, the coincident and the quadrature spectral density functions P and Q have to be computed:

$$P = \int_0^{2\pi\, ncycl/w} y(t)\, cos(w\, t)$$

$$Q = \int_0^{2\pi\, ncycl/w} y(t)\, sin(w\, t)$$

In the experiment block the computation of *one point* (for a w) of the co-spectrum and quad-spectrum can be set:

```
Experiment armonic_contents is
      Static region
            inputs (real w;)
            outputs (real P,Q;)
```

```
            state {real P,Q;}
            auxiliary variable {real y;}
            submodels called
                  {Van_der_Pol;}
            functions called
                  {real sin,cos;}
      End static region;
      Initial region
            P   = 0.0;
            Q   = 0.0;
      End initial region;
      Dynamic region
            y   = Van_der_Pol();
            P'  = y*cos(w*sy_time);
            Q'  = y*sin(w*sy_time);
      End dynamic region;
      Control segment
            control si_finti=48.0;
            parameter Van_der_Pol.mu = 5.0;
      End control segment;
      Output segment
            prepare/output=(file1.dat)/delt=(0.02) y,
                  Van_der_Pol.y1,P,Q;
      End output segment;
End experiment armonic_contents;
```

Code 2.2 *Armonic_contents experiment.*

The power spectral density analysis is controlled from the study block. The finish time (*si_finti*) is not constant and depends on w:

$$si\_finti = 2\pi\, ncycl/w$$

The above equation as well as the sweeping of w will be included in the dynamic region of the study to control the experiments.

```
Study power_spectral_density_analysis is
      Static region
            experiments called {armonic_contents;}
            auxiliary variables {real DEP, P, Q, Pm, Qm, w;}
            parameters { real w0 = 0.75, wf = 2.0,
                              wincr = 0.01; int ncycl = 20;}
      End static region;
      Dynamic region
            Pm = 0.0;
```

```
            Qm = 0.0;
            for (w = w0 to wf by wincr) {
                  si_finti = 2.0*si_Pi*ncycl/w;
                  P,Q = armonic_contents(w);
                  DEP = w/(ncycl*2.0*si_Pi)*
                        sqrt((P-Pm)**2.0+(Q-Qm)**2.0);
                  Pm = P;
                  Qm = Q;
            }
      End dynamic region;
      Control segment
            parameter armonic_contents.Van_der_Pol.y10 = 1.0;
      End control segment;
      Output segment
            prepare/cross/output=(file2.dat) DEP
      End output segment;
End study power_spectral_density_analysis;
```

Code 2.3 *Power_spectral_density_analysis study*

An interactive monitor language (*MCL*) has been defined to control the execution of experiments and studies present in the simulation environment. Thus, the user may optionally invoke for execution the *armonic_contents* experiment or *power_spectral_density_analysis* study.

2.3 Scope of variables, parameters and constants

In high-level procedural languages there are two ways of passing data between subprograms: calling sequence and global data.

- *Calling sequence:* Although there are exceptions, the use of the calling sequence mechanism for parameter passing is the preferred method to pass information into a subprogram, and out of a subprogram. This tends to improve modularity of the code, and the possibility of undesirable side effects are reduced [Kettenis86b].
- *Global data:* Unrestricted communication between program modules through global data blocks leads to increased complexity and confusion within the system [Golden85a] ("back-door" programming [Cellier79c]).

Besides the above opinions, we are faced with the following requirements and restrictions:

- Symbolic access to the all submodel constants, parameters and variables must be allowed for recording.
- If isolate preprocessing of submodels is to be maintained, the calling sequence mechanism must be forced for submodel variables communication.
- Using the calling sequence mechanism for constants and parameters communication unnecessarily increases the time overhead.
- From the study block, is should be able to access any parameter from the experiments and models called. This option is important, for example, in sensitivity analysis or in optimization studies.

Data communication between program blocks is carefully controlled to meet the above requirements. From the simulation users perspective, the use of global data has been restricted to parameters and constants and the calling sequence mechanism is mandatory for the communication of variables. From the internal perspective, global data facilities provided by the C language are not used to implement global data mechanisms between program blocks; the mechanisms used are explained in chapter 4.

The rules imposed to global data communications are emphasized in figure 2.9 and can be cleared up looking to the sequence of steps done when performing an experiment or a study:

- **Experiment:**
  - First, when creating an experiment instance[4], parameters are initialized in a bottom up approach (*static initialization*). Parameters in a given block can be initialized within the *static region* of the block or from the *static region* of higher level blocks. The highest level initialization of a given parameter overrides lower level initializations of the same parameter.
  - Second, after the static initialization, the *control segment* of the experiment is executed. The control segment of an experiment may update the value of the parameters in the static region of the experiment or the value of the parameters of the models called.
  - Third, the *output segment* of the experiment is executed to specify the set of variables to be recorded, printed or represented graphically at periodic intervals. All the variables from the experiment and from the called models are visible from this segment.

[4]The creation of an instance of an experiment implicitly implies the creation of an instance for each model called. If a model is called twice, two instances of this model will be created.

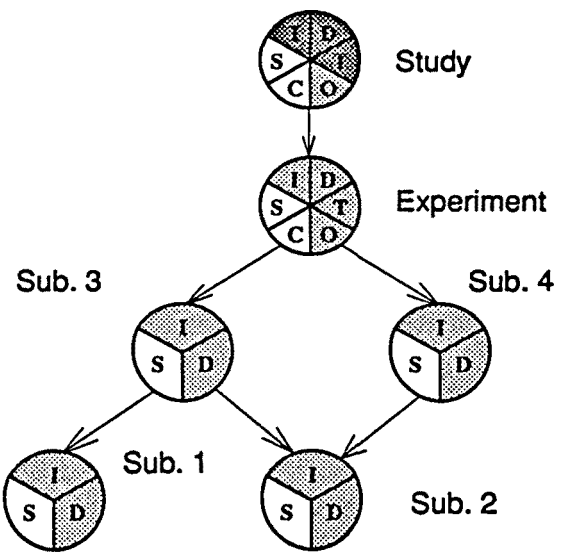

Figure 2.9: *Communication through global data.* **S, I, D** *and* **T** *represent the static, initial, dynamic and terminal regions respectively and* **O** *and* **C** *represent the output segment and the control segment respectively. From the white colored region or segment of a given block constants and parameters of the block can be set and initializations of parameters in lower level blocks can be overrided. In the gray regions or segments of a given block any parameter or constant of the block or any parameter or constant of lower level blocks can be used but not modified. In the dark regions of the study any parameter of the study or any parameter of lower level blocks can be used or updated.*

- Fourth, the user may optionally change, using the *MCL* monitor language, any parameter of the models involved in the experiment or any parameter of the experiment itself (this option is not represented in figure 2.9). Moreover, control and output specifications may be updated.
- Fifth, the next step consists on executing the experiment. First, the initial region of the experiment and the initial regions of the called submodels are executed[5] (*dynamic initialization*). The initial conditions of the state variables must be set in the *initial region*. After the dynamic initialization has concluded, a simulation run is conducted.

  In the *initial region* or in the *dynamic region* of a block we can use but not update parameters and constants from the block or from lower level blocks.
- Last, after the simulation run is finished, the *terminal region* of the experiment is executed. In the terminal region we can use any parameter, constant or variable from the experiment or from the called models.

- **Study:** From the study block we can invoke for execution one or more experiment blocks. The steps involved in the execution of a single experiment have been already explained. Now we will focus our attention in the additional steps which involve study segments or regions.

  - First, when creating a study instance, copies of the experiments and models which are involved in the study are implicitly created. From the *static region* of the study block we can optionally set the value of any parameter of the study or the value of any parameter of the experiments and models involved.
  - Second, after the static initialization, the *control segments* of the experiments and, afterwards, the *control segment* of the study are executed. Thus, from the control segment of the study we can override parameter initializations done in the control segments of the experiments.
  - Third, the *output segment* of the study is executed (the *output segments* of the experiments are not executed). All the variables, parameters and constants of the study and experiments and models involved in the study are visible from this segment.
  - Fourth, the user may optionally change any parameter of the experiments and models involved in the study or any parameter of the study itself. Moreover, control and output specifications may be updated.

[5] In fact, the initial segment associated to the experiment and the initial segments of the called lower level submodels blocks are executed to initialize the experiment and the called models (chapter 3). The initial segment of a given block is a routine that includes, besides the code of the initial region needed to initialize the block, code from the dynamic region needed to initialize lower lower blocks. Moreover, it includes code needed for computing the initial value of the discontinuous functions of the block and of the discontinuous functions of lower level blocks.

- Last, the next step consists on executing the study. First of all, the initial region of the study is executed. Afterwards, control is passed to the *dynamic region* o the study which monitors the execution of a set of experiments. At the end, the *terminal region* of the experiment is executed.
  In the *initial region*, *dynamic region* and *terminal region* of a study we can use and update parameters and use constants and variables from the study block or from lower level blocks.

2.4 MUSS preprocessor

The *MUSS* preprocessor has been designed using automatic compiler production techniques. *Automatic compiler production* is the task of automatically constructing practical compilers from minimal specifications of the source and target languages [Reiss87a]. Its aim is to ease the construction and update of compilers for new or existing languages, and to assist on the grammar definition.

Besides automatic compiler construction techniques, *MUSS* relies on modular programming methodologies to improve the robustness of the software.

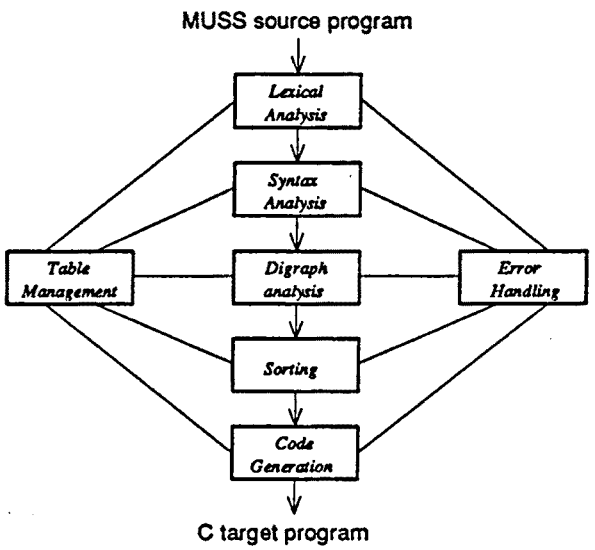

Figure 2.10: *Preprocessor structure.*

The modular structure of the *MUSS* preprocessor is represented in figure 2.10. The

main modules of the *MUSS* preprocessor are:

- *Lexical analyzer:* produces a sequence of input symbols (tokens) for the syntax analyzer. The input to the lexical analyzer is a stream of characters.
- *Syntax analyzer:* maps the token representation of the source program into an attributed abstract syntax tree.
- *Submodel digraph analyzer:* besides constructing the submodel digraph (subsection 3.2.3), it checks using the submodel digraph, the code consistency and extracts from the submodel digraph the initial, discontinuous and ODE segment digraphs (section 3.3).
- *Sorter:* to solve the hierarchical sorting problem, the ODE segment digraph is splitted into the state, algebraic and derivative segment digraphs. Thereafter, the code in each segment is sorted using classical techniques (section 3.4).
- *Code generator:* creates the initial, discontinuous, state, algebraic and derivative segments as well as the associated data structures (chapter 4). The chosen target language is *C*. Since statements in the *MUSS* language are similar to statements in the *C* language, the translation of the statements is simple.
- *Table manager:* stores and retrieves program symbols from a table. A balanced tree technique has been chosen to implement the symbols table.
- *Error handler:* this module is intended for precisely inform the user about the errors in the source program code.

The final goal of the *MUSS* preprocessor is to produce a well-conditioned target code from a correct *MUSS* source program.

MUSS source program

In the design of the language it has been tried to extract the best from GEST [Ören84a], COSMOS [Kettenis86b] ,ESL [Hay85a], C [Kernighan78a] and ADA [Ledgard81a]

A draft of the *MUSS* language specification is presented in [Guasch86b]. The language has been defined with a LALR(1) grammar [Cellier79b]. By its use, it is possible to ensure that no syntactic errors will propagate to the *C* target code.

C target program

In chapter 3 the structural transformations to be performed to the *MUSS* continuous submodel blocks are presented. Thereafter, in chapter 4, the *C* object code structure of this block is shown.

This point is exclusively centered around the following question: *Why the C language has been chosen as the basic MUSS system software support?*

This question may be answered weighting the following aspects: *availability, suitability, reliability, transportability* and *availability of supporting tools.*

When the development of the *MUSS* system started, five general purpose programming languages were analyzed: *Ada, C, Fortran, Modula-2,* and *Pascal.*

In spite of the large amount of software in numerical analysis written in *Fortran,* this language was rejected because it is not suitable for:

- Coding a preprocessor.
- Code the algorithms and the data structures involved in the management of the simulation environment.

Ada was rejected mainly because at the time the coding of the *MUSS* system started, *Ada* compilers were not available for VAX computers. Other objections concerning reliability and transportability [Hoare81a] supported the decision.

Although *Modula-2* [McCormack83a] seems a very suitable language and it supports object-oriented programming methodologies. It has not been chosen because of the lack of software support. Moreover, *Modula-2* is not a native language processor in VAX systems.

Even though *Pascal* is a reliable language, its standard set is not suitable for coding the *MUSS* system [Kernighan81a]. Some relevant disadvantages of *Pascal* are its lack of standard facilities for separate compilation and the fact that records can not hold references to executable functions or procedures.

One primary reason why the *C* language has been selected is that the *LEX* (lexical analyzer generator) and *YACC* (compiler-compiler) software tools use and generate *C* code. Moreover it is a suitable and transportable language.

However, the clarity of the code and associated data structures should be enforced using good programming methodologies because *C* allows users to write 'cryptic' programs.

2.5 Summary

In this chapter the architecture of the *MUSS* language which agrees with the proposed objectives (page 1.2) has been introduced.

The grammar selected for *MUSS* is of the lookahead-left-to-right (LALR(1)) type. LALR techniques of parser construction are chosen because the tables obtained are considerably smaller than the canonical LR tables [Aho77a]. The formal design of the *MUSS* language is carried out using syntax analysis tools.

The formalization, in the next chapter, of the submodel digraph concept constitutes the base for improving the robustness of the submodel code. The next chapter will be mainly concerned about the formal analysis of lumped parameters submodels.

Chapter 3

Continuous submodel analysis

"Most compilers for computer languages available today do some (marginal) program verification besides the translation from source-code to run-time object-code. Most of the checking, however, is limited to verification of adherence of the program to the semantic and syntactic rules laid down in the language definition, and restrictions added to them by the compiler limitations. One could call this type of error detection text verification because it provides no information at all on the integrity of the program."

[Elzas79a]

3.1 Introduction

Monolithic simulation languages are not flexible enough to easy the task of studying models of high complexity, even for expert users. New structures —macro, sample, submodel, module, model— have been added to simulation languages to increase its modularity, but this effort to increase the modularity has not been extended to increment the "intelligence" of preprocessors and compilers for simulation languages.

This chapter is concerned with the analysis of continuous time lumped parameters submodels which can be performed by the *MUSS* preprocessor. It has three main parts:

- The first one is devoted to present the submodel digraph concept (section 3.2). The submodel digraph shows the relationships between submodel variables, declarative sentences and declarative sentences and variables.

Two types of representation are used. The first one, named *planar representation* , emphasizes the program flow —i.e. the relationships between sentences according to its input/output variables—. The second one, named *biplanar representation* , emphasizes the relationships between variables, parameters and constants, which are represented in a plane (*symbols plane*), and the relationships between sentences, represented in another plane (*statements plane*); the relationships between sentences and variables are represented by edges going from one plane to the other.

- The second main part (section 3.3), is principally concerned with the robustness study of the submodel code. Through a formal analysis, a high degree of robustness can be assured.

- In the third part (section 3.4), a global solution to the sorting problem of the submodel code relatively to other submodels, is proposed.

  So far, the absence of a general sorting method to arrange the statements, defined in a submodel, relatively to other submodels, constraints the degree of modularity of the simulation languages[1]. Using classical methods, *information loops* [Baker83a], *those that disappear if the called lower level submodels are handled like MACROS*, appear in the calling submodel.

  The proposed solution, based on a segmentation of the submodels supported by a graph analysis, breaks the submodel code into a set of related blocks (segments).

  Each submodel call is splitted into a set of segment calls, one call for each segment, when called from higher level submodels.

3.1.1 Continuous submodel

Grammar

A summary of the grammar of *MUSS* continuous submodels is included below. The metalanguage used to specify the grammar is described in appendix A.

```
MUSS_continuous submodel
        : continuous_submodel_heading
          continuous_submodel_regions
          continuous_submodel_end
        ;
```

[1] If either the input variables or the output variables are all states, there is no problem in sorting submodels as an entity. To avoid sorting conflicts, restrictions are necessary on the form the model code can take.

```
continuous_submodel_heading
        : CONTINUOUS SUBMODEL IDENTIFIER IS
        ;

continuous_submodel_regions
        : static_region other_regions
        ;

other_regions
        : initial_region dynamic_region
        | dynamic_region
        ;

initial_region
        : INITIAL REGION procedural_block END INITIAL REGION ';'
        ;

procedural_block
        : set_of_procedural_statements
        ;

set_of_procedural_statements
        : procedural_statement
        | set_of_procedural_statements procedural_statement
        ;

procedural_statement
        : if_statement
        | procedural_assignment_statement
        ;

dynamic_region
        : DYNAMIC REGION declarative_block END DYNAMIC REGION ';'
        ;

declarative_block
        : set_of_declarative_statements
        ;

set_of_declarative_statements
        : declarative_statement
        | set_of_declarative_statements declarative_statement
        ;

declarative_statement
        : if_discontinuous_statement
        | when_discontinuous_statement
        | declarative_assignment_statement
        ;
```

```
continuous_submodel_heading
        : END SUBMODEL IDENTIFIER ';'
        ;
```

Several aspects of the grammar can be stressed:

- The `declarative_assignment_statement` grammar rule includes submodel calls (i.e. `y = limiter(ll,ul,x);`).
- Two declarative statements to describe discontinuities have been initially conceived within the dynamic region:

  1. `if` statement : It is used to handle decisions which affect the choice of a specific set of declarative statements. The formal definition is,

```
if_discontinuous_statement
        : IF '(' logical_expression ')'
             '{' limited_declarative_block '}'
          else ';'
        ;

else
        : /* empty */
        | ELSE '{' limited_declarative_block '}'
        | ELSE IF '(' logical_expression ')'
              '{' limited_declarative_block '}'
          else
        ;
```

  As a first remark, see that statements within the `if` statement (`declarative_block`) are declarative in contrast with other languages `if` statements where the code inside them is procedural. In fact, the `if` statement chooses the set of statements ('equations') which are valid over each span of time, where each span of time is delimited by two events.

  `If` statements are not allowed within an `if` statement. That is, the `limited_declarative_block` is equal to the `declarative_block` with the exception that it does not include `if` statements[2]

  A sequence of `else if`'s in the construction

```
if(logical_expression) {
    ..........
```

[2]The formal definition of the `if` declarative statement can be extended to allow `if` statements in it.

```
    ..........
} else if(logical_expression) {
    ..........
    ..........
} else if(logical_expression) {
    ..........
    ..........
} else {
    ..........
    ..........
};
```

is a general way of writing a multi-way decision. For analysis purposes we work with the following equivalent construct,

```
if(logical_expression) {
    ..........
    ..........
} else {
      if(logical_expression) {
          ..........
          ..........
      } else
            if(logical_expression) {
                ..........
                ..........
            } else {
                ..........
                ..........
            };
      };
};
```

2. When statement : It describes actions having to take place at event occurrences. Code within the when statement is assumed to be procedural. It is equivalent to the AS SOON AS statement of *COSMOS* or to the WHEN statement of *ESL*.

Analysis aspects

Along the analysis of continuous submodels, the following points have to be taken into account:

- An analysis of the robustness of the submodel code is wanted.
- All the submodel variables may be able to be accessed symbolically.

- Many software packages —i.e. LSODE [Hindmarsh82a] DISCO [Birta85a], [Thompson85a]— for solving ODE's models with discontinuities require two different subprograms.

  - A subprogram defining the ODE system (ODE subprogram).
  - A subprogram defining de discontinuity functions —wanted implicit solutions of the ODE system— (discontinuous subprogram).

  If such a structure is to be preserved[3] the discontinuity functions should be computed within the discontinuous subprogram and the derivative computations within the ODE subprogram. Thus, for each continuous submodel, it will be necessary to split the code in the dynamic region into two subprograms, which will be called segments: the *discontinuous segment* and the *ODE segment.*

- The initialization of a continuous submodel needs special care. Besides checking that all state variables are initialized, the analysis algorithm must check that all the discontinuous functions can be computed at initial time, this is needed to set the logical states associated to each discontinuous function before calling the ODE segment. Furthermore, the analysis algorithm clusters in a subprogram (*initial segment*) code from the initial region, code needed to compute the discontinuous functions at initial time, dynamic code needed to initialize lower level continuous submodels.

- Code in the dynamic region is declarative. Besides sorting the code, the sorting algorithm must restructure the dynamic code to allow higher level submodels to be sorted.

3.1.2 Introductive example

System

Figure 3.1 shows a non linear system presented in block-diagram form.

[3]Preserving this structure improves the readability and robustness of the preprocessor C target code. Furthermore, the computation time overhead is reduced mainly because when precisely locating discontinuities the root finder algorithm only calls the discontinuous subprogram instead of the whole dynamic code.

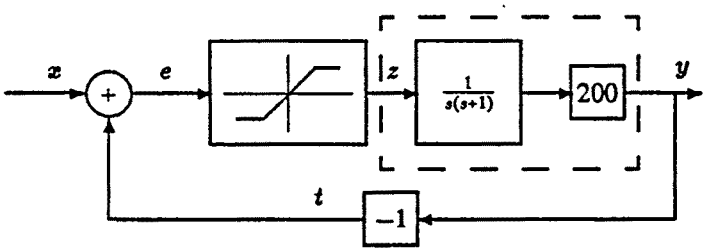

Figure 3.1: *Non linear system.*

Model code

The MUSS code for simulating the system has been subdivided in two submodels. The *non_linear_system* submodel (code 3.1) represents the behavior of the whole system and calls *second_order* submodel (code 3.2) which simulates the blocks included within the dash box.

```
Continuous submodel non_linear_system is
      Static region
            inputs {real x;}
            outputs{real y;}
            auxiliary variables {real e,t,z;}
            submodels called {
                  second_order;
            }
      End static region;
      Dynamic region
          e  = x+t;
          if (e>=2.0) {
             z = 2.0;
          } else if (e<-2.0) {
             z = -2.0;
          } else {
             z = e;
          };
          y = second_order(z);
          t = -y;
      End dynamic region;
End submodel non_linear_system;
```

Code 3.1 *Non_linear_system submodel.*

Using conventional sorting algorithms [Zeigler76a] a loop will be found in *non_linear_system* because $y = f_1(f_2(f_3(x, f_4(y)))$ since the sorter has no knowledge of the *second_order* internal structure, this is reflected in the digraph of figure 3.2.

```
Continuous submodel second_order is
      Static region
            inputs {real z;}
            outputs{real y;}
            state  {real r,s;}
      End static region;
      Initial region
          r  = 0.0;
          s  = 0.0;
      End initial region;
      Dynamic region
          r' = z;
          s' = r-s;
          y  = 200.0*s;
      End dynamic region;
End submodel second_order;
```

Code 3.2 *Second_order submodel.*

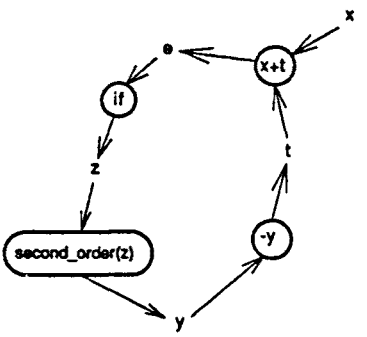

Figure 3.2: *Digraph associated to the non_linear_system submodel. An implicit loop appears due to the lack of knowledge about* second_order *submodel.*

Second_order submodel code can be divided in two subprograms:

- One clustering the sentences which compute the output variables of the submodel depending on state variables and constants.

- The other grouping the sentences involved in the derivative computations.

Therefore, the *second_order* call can be replaced by two calls:

```
y = state_second_order();
derivative_second_order(z);
```

and the loop will disappear (figure 3.3).

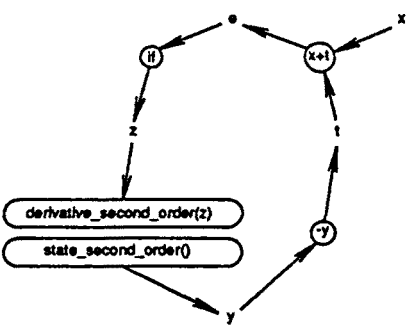

Figure 3.3: *New digraph associated to the non_linear_system submodel after splitting lower level submodel calls.*

Surprisingly, digraphs which represent the relations between the declarative sentences that describe the model have just been used to sort the declarative code or to make simple tests on it. A formal representation of the relationships between sentences by means of a digraph is needed in declarative statements for sorting purposes and it can be very useful for analyzing in more detail the model code. In general, simulation languages designers have limited the complexity of the structure and grammar rather than improving the analysis and sorting methods.

In contrast, in the *MUSS* system the digraph representation of the relations involved in the submodel code are mainly used for code analysis and marginally for sorting purposes. The digraph which will be called *submodel digraph*, represents in great detail the relations involved in the submodel declarative code.

Submodel digraph: analysis and sorting

During the preprocessing stage of a continuous submodel, the preprocessor builds the submodel digraph, which is an internal representation of the continuous submodel. The submodel digraph is built after performing two transformations in the code:

- Trigger conditions associated to discontinuous statements are translated into expressions.
- Calls to lower level submodels are expanded into a call for each submodel segment.

Afterwards, the preprocessor will create the submodel digraph where each declarative statement has a vertex (*executable vertex*)[4].

```
-1-  Initial region
         r  = 0.0;
         s  = 0.0;
     End initial region;
     Dynamic region
-2-      r' = z;
-3-      s' = r-s;
-4-      y  = 200.0*s;
     End dynamic region;
```

Code 3.3 *Labeled executable vertices of second_order submodel.*

Continuing with the previous example, code 3.3 shows the number of the executable vertices and Figure 3.4 represents the two isomorphic representations —planar and biplanar— of the *second_order* submodel digraph:

- *Planar representation:* It is used to emphasize the general program flow and to show the different segments which are the output results of the analysis algorithms. The ODE segment (see page 38) will be splitted by the sorting algorithm into *state, algebraic and derivative* segments in order to solve the problem of information loops.

  In the example the segments are:

[4]The initial region and when statements are viewed as a single executable vertices

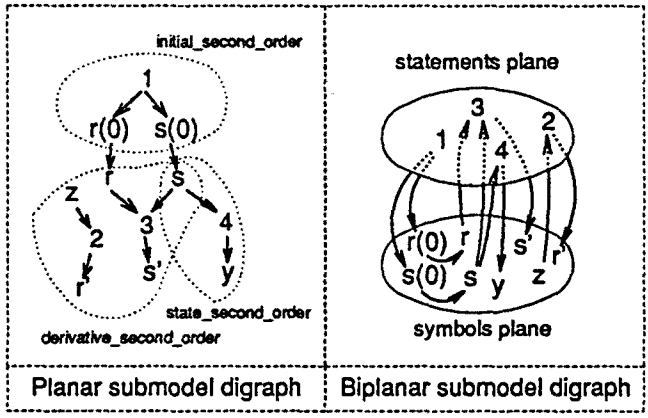

Figure 3.4: *Planar and biplanar representations of second_order submodel digraph.*

- initial_second_order: It is the initial segment associated to *second_order*. It must be called before any other *second_order* segment.
- state_second_order: The state segment computes the output variables which do not depend on input variables through algebraic chains.
- derivative_second_order: The derivative segment includes the code needed to compute derivatives which is not included in the state segment or in the algebraic segments.

- *Biplanar representation:* It emphasizes the relationships between variables in the symbols plane and the relationships between executable statements in the statements plane. As a first remark, note that the relation between the state variables s and r and their initial conditions is explicitly represented in the symbols plane. The meaning of the symbol s, r in the initial region (code 3.3) is conceptually different from that in the dynamic region. Each state variable in the dynamic region must be explicitly initialized, in the initial region. The absence of an edge between the state variable and its initial condition is erroneous. This can be easily detected by the analysis algorithm.

A *segment-link digraph* is defined to show the relationships between submodel segments (figure 3.5 represents the segment-link digraph associated to *second_order*). Any call to a submodel from a higher level submodel will be splitted into several calls, one for each submodel segment.

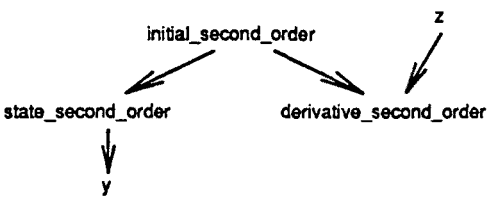

Figure 3.5: *Second_order segment-link digraph.*

The edges of the segment-link digraph are called *hidden edges* because they can not be inferred from the higher level submodel through the input-output variables in the segment calls.

To build the higher level submodel digraph, the segment-link digraphs of the called lower level submodels are needed in order to properly relate the expanded calls. This knowledge about submodels interfacing is stored in the model data base.

Once *second_order* has been preprocessed without errors, it is allowed to preprocess *non_linear_system* or any other higher level submodel calling it. When preprocessing *non_linear_system* submodel, the call to *second_order* submodel is divided into a call for each submodel segment (see code 3.4).

```
     Dynamic region
-1-      e       = x+t;
-2-      groot1  = e-2.0;
-3-      groot2  = -2.0-e
-4-      if(lroot1) {
-5-         z = 2.0;
         } else {
-6-         if(lroot2) {
-7-            z = -2.0;
            } else {
-8-            z = e;
            };
         };
-9-      initial_second_order();
-10-     y = state_second_order();
-11-     derivative_second_order(z);
-12-     t = -y;
     End dynamic region;
```

Code 3.4 *Labeled executable vertices of non_linear_system submodel.*

Furthermore, to build the *non_linear_system* submodel digraph, it is necessary to add the hidden edges of *second_order* segment digraph to those associated to the explicit knowledge about *non_linear_system* submodel code.

Notice that the piece of code from *MUSS* submodel code 3.1:

```
 if(e>=2.0) {
.............
.............
 } else {
.............
.............
 };
```

has been decomposed by the preprocessor in

```
 groot1  = e-2.0;
 if(lroot1) {
.............
.............
 } else {
.............
.............
 };
```

as shown in code 3.4.

`Groot1` stores the value of the discontinuous function and `lroot1` is the logical state associated to the discontinuous function.

The cause of an event is independent from its effect due to the fact that logical states associated to discontinuous functions are state variables [Horst86a]. In spite of this, edges from discontinuous function vertices (3 and 2) to their associated discontinuous effect ((2,4) and (3,6_)) are present in the submodel digraph. These edges are only considered in the submodel initialization analysis.

Figure 3.6 and figure 3.7 represent the biplanar and planar representation respectively of *non_linear_system* submodel digraph:

- *Biplanar representation:* The edges (z_+, z) and (z_-, z) represent that the variable z is formed by the union of variables which are defined, each one, over different spans of time. Moreover, the edges (z_{-+}, z_-) and (z_{--}, z_-) represent that the

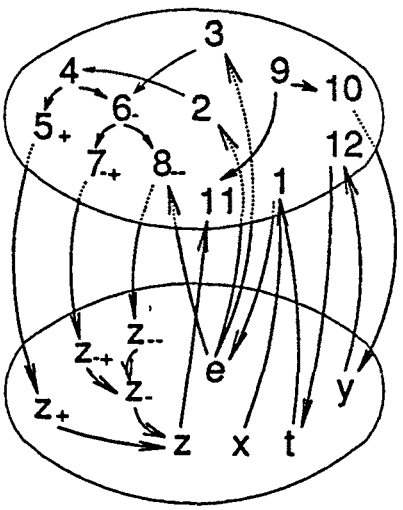

Figure 3.6: *Biplanar representations of non_linear_system digraph. Edges* (2,4) *and* (3,6_) *are only used for initialization analysis.*

variable z_- is composed by the union of z_{-+} and z_{--} being these continuous variables defined over different spans of time.

In the statements plane, the edges (9, 10) and (9, 11) are the hidden edges associated to *second_order* segment-link digraph. Moreover, edges (4,5+) and (4,6_) represent the relationships between the `if` statement and the statements included within it. The same for the edges (6_,7_+) and (6_,8__).

- *Planar representation:* In this case the segments are:
  - initial_non_linear_system: includes code to initialize state variables and logical states associated to discontinuous functions. Note that vertex 10 which is a call to the lower level *second_order* state segment must be included in this segment in order to compute the initial values of the discontinuous functions.
  - state_non_linear_system: the code in this segment does not depend on input variables. Therefore, output variables can be handled like state variables.
  - derivative_non_linear_system: the statements in this segment does not contribute to output computations. Thus, input variables can be handled like derivative variables.
  - discontinuous_non_linear_system: includes code needed to compute

$$\overline{g} = f(\overline{y}, t)$$

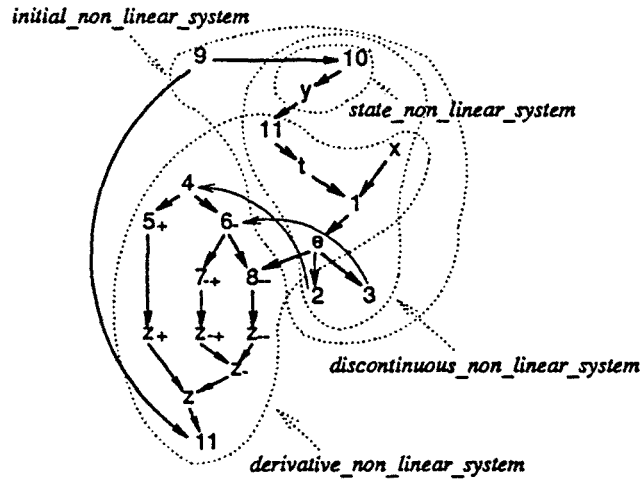

Figure 3.7: *Planar representation of non_linear_system digraph.*

being $\overline{y}$ the state vector, t the simulation time and $\overline{g}$ the discontinuous vector.

To get the right execution sequence (procedural code), each submodel segment is sorted with independence from the other segments. Afterwards, the segments are sorted according to the segment-link digraph.

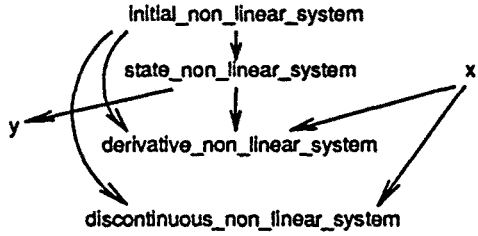

Figure 3.8: *Non_linear_system segment-link digraph.*

The non_linear_system segment-link digraph is represented in figure 3.8. Higher level models might be coded calling the *non_linear_system* submodel. These calls would be splitted into a call for each segment at preprocessing time.

3.2 Submodel digraph

Graph theory has been successfully applied to study and analyze computer programs since the early years of computer programming. The aim of such analysis might be to subdivide a large program into a number of subprograms or segments, to detect structural errors in the program, to document a program or to optimize the code.

Simulation packages designers started using graph theory to analyze code associated with the dynamics of the model. Stein [Stein60a] provided the theoretical and practical background needed to write a sorting algorithm implicitly based on graph theory.

Later on, Elmqvist [Elmqvist80a] employed graph theory to sort the model equations and transform them into assignment statements by automatic formula manipulation. Furthermore, the role of dataflow methods in Continuous System Simulation Languages have been introduced in [Karplus82a], in data flow methods all the information needed to execute the program must be contained in its data flow graph.

More recently, a computer aided modelling, analysis and simulation environment (*CAMAS*) [Broenink85a] which accepts as input a bond graph modelling language (SIDOPS) [Broenink85b] have been developed.

In this thesis, graph theory is used to analyze submodel code consistency as well as to solve the problem of submodel sorting when the hierarchical modelling and coding approach is utilized.

3.2.1 Elementary graph concepts

A *digraph* (directed graph) is a collection of *vertices* $V = v_1, v_2, ...$, a collection of *edges* $E = e_1, e_2, ...$ and a mapping Ψ that maps every edge onto some *ordered* pair of vertices (v_i, v_j). Vertices are simple objects and an edge is a connection between v_i and v_j with an arrow directed from v_i to v_j.

A *path* from vertex v_i to v_j in a graph is a list of vertices in which successive vertices are connected by edges in the graph. A path is directed —*directed path*— if the edges have the same orientation, otherwise, it is a *semipath*. A graph is *connected* if there is a path from every vertex to every other vertex in the graph. A graph which is not connected is made up of *connected components*. A connected graph is *strongly connected* if there is at least one directed path from every vertex to other vertex, otherwise, it is *weakly connected*. A *simple directed path* is a directed path in which no vertex is repeated and a *directed cycle* is a directed path which is simple except that the first and last vertex are the same.

A vertex v_j is *reachable* from a vertex v_i if there is a directed path from v_i to v_j.

A graph without cycles is called a *tree*. A group of disconnected trees is called a *forest*.

Graphs with all edges present are called *complete* graphs; graphs with relatively few edges are called *sparse*; graphs with relatively few of the possible edges missing are called *dense*.

In order to be segmented and sorted, graphs have to be directed and acyclic. These graphs are called *directed acyclic graphs* (*dags*).

Some formal definitions

Definition 3.1 *Given a vertex v_i belonging to V, $R(v_i)$ is the set of vertices reachable from the vertex v_i.*

Definition 3.2 *Given a vertex v_i belonging to V, the reaching set $Q(v_i)$ is the set of vertices which can reach vertex v_i.*

Definition 3.3 *Given a vertex v_i belonging to V, $\Gamma(v_i)$ is the set of successors from v_i*

Definition 3.4 *Given a vertex v_i belonging to V, $\Gamma^{-1}(vi)$ is the set of predecessors of v_i.*

3.2.2 Submodel digraph definition

Definition 3.5 *Given a MUSS lumped-parameter continuous submodel, its associated submodel digraph is* $G = (V, E)$, *where:*

$V = (Vs, Ve)$ *is the set of vertices,*

being:

- Vs*: set of* **symbol vertices** *associated to variables, parameters or constants.*
- Ve*: set of* **executable vertices**. *Each executable vertex is associated to one declarative statement. The initial region is viewed as a single executable vertex.*

Different subsets of Vs and Ve can be defined:

1. Subsets of Vs

   Vs_{gl} : *global vertices*. All the variables, constants and parameters declared in the submodel static region have an associated global vertex.

   Vs_p : *parameter vertices*, each parameter has a vertex associated to it ($Vs_p \in Vs_{gl}$).

   Vs_c : *constant vertices*, each constant has a vertex associated to it ($Vs_c \in Vs_{gl}$).

   Vs_i : *input vertices*, each formal parameter in the submodel input list (input variable) becomes an input vertex ($Vs_i \in Vs_{gl}$).

   Vs_o : *output vertices*, each formal parameter in the submodel output list (output variable) becomes an output vertex ($Vs_o \in Vs_{gl}$).

   Vs_d : *derivative vertices*, each state variable has an associated derivative vertex ($Vs_d \in Vs_{gl}$).

   Vs_s : *state vertices*, each state variable has an associated state vertex ($Vs_s \in Vs_{gl}$).

   Vs_{lo} : *local vertices*, $Vs_{lo} = Vs_n \cup Vs_{wh} \cup Vs_{if}$. Are those created by the preprocessor as a consequence of the analysis of the submodel initial and dynamic regions.

   Vs_n : *initial vertices*, each variable initialized within the initial region has one associated initial vertex ($Vs_n \in Vs_{lo}$).

Vs_{wh} : *when vertices*, each variable defined in a procedural statement included in a when statement has one associated when vertex ($Vs_{wh} \in Vs_{lo}$).

Vs_{if} : *if vertices*, set of symbol vertices defined to represent the relations involved in `if` declarative statements ($Vs_{if} \in Vs_{lo}$).

2. Subsets of Ve

Ve_d : *derivative vertices*, each call to a lower level derivative segment has an associated derivative vertex.

Ve_g : *discontinuous vertices*, each call to a lower level discontinuous segment or each discontinuous function has an associated discontinuous vertex.

Ve_g can be subdivided in two sets: vertices associated to calls to lower level discontinuous segments (Ve_{gs}) and vertices associated to discontinuous functions (Ve_{gf}).

Ve_s : *state vertices*, each call to a lower level state segment has an associated state vertex.

Ve_n : *initial vertices*, each call to a lower level initial segment or the initial region has an associated initial vertex.

Ve_{if} : *if vertices*. Set of vertices associated to `if` statements.

Ve_{wh} : *when vertices*. Set of vertices associated to when statements.

Submodel digraphs are sparse. Thus, with relatively few edges ($< |V| \times log(|V|)$). Therefore, the submodel digraph is implemented using adjacency lists.

3.2.3 Construction of the submodel digraph

Instead of generating cross tables along with the syntactic analysis and afterwards, test and sort the code scanning repetitively the internal tables, the *MUSS* preprocessor speeds up the analysis and sorting phases through the construction of a submodel digraph.

The submodel digraph can be built in a single pass, along the syntactic analysis, supported by the global vertices defined through the static region analysis.

The following code is included to illustrate the above ideas,

```
          ::::::::::::::
      Static region;
            inputs  {real R0, ... ;}
            outputs {real V0;}
            state {real Vc,Il;}
            auxiliar variables {real I0,Ic; ... }
            parameters {real Rc = 0.1, ... ;}
      End static region;
      Dynamic region
-1-         I0  = (Vc+Il*Rc)/(R0+Rc);
-2-         V0  = I0*R0;
-3-         Ic  = Il-I0;
              ::::::::::::::
      End dynamic region;
         ::::::::::::::
```

During the syntactic analysis of the static region, for each declared variable[5] a global symbol vertex is defined. The symbols with their associated vertices are stored in the *symbols table*[6] (figure 3.9).

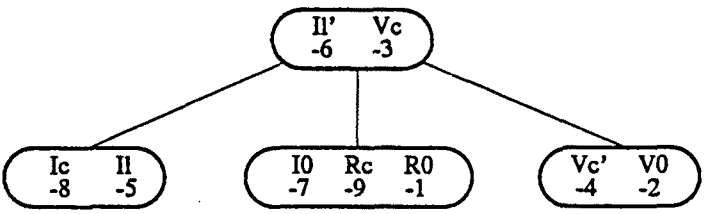

Figure 3.9: *Balanced tree used to store the symbols table.*

Afterwards, for each statement being preprocessed an executable vertex in the statements plane is defined. For each variable in the statement a search into the symbols table is made to find the global vertex in the symbols plane associated to it. In this way, the edges from the global vertices to the executable vertices or form the executable vertices to the global vertices are defined.

Two adjacency lists have been defined for implementing the submodel digraph (figure 3.10): the *symbols adjacency list* and the *statements adjacency list.*

[5]The term *variable* embraces constants, parameters and variables.

[6]The symbols table is stored in a balanced tree which has been implemented using the top-down 2-3-4 tree technique [Sedgewick83a].

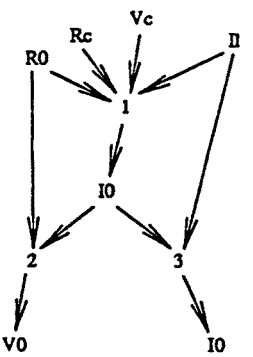

Figure 3.10: *Since global symbol vertices ($Vs_{gl} = \{R0, Rc, Vc, I1, I0, V0\}$) have been defined when performing the syntactic analysis of the static region, the inclusion of executable vertices and its associated edges from or to global symbol vertices is straightforward.*

(SYMBOLS ADJ. LIST)

Directed edges from the symbols plane to the symbols or statements plane.

```
-1:  1  2
-2:
-3:  1
-4:
-5:  1  3
-6:
-7:  2  3
-8:
-9:  1
```

(STATEMENTS ADJ. LIST)

Directed edges from the statements plane to the statements or symbols plane.

```
1:  -7
2:  -2
3:  -8
```

To get the submodel digraph, two sets of rules are used: the *transformation rules* and the. Both sets are used along the syntactic analysis of the code in a single pass.

Transformation rules

Rule T.1 : Submodel calls are splitted into a call for each submodel segment. The

segment-link digraphs of the called lower level submodels are inserted in the submodel digraph.

Let be, the submodel call,

```
y = second_order(z);
```

included in the *non_linear_system* submodel coded in page 39 (submodel code 3.1). The preprocessor will split it into three segment calls (code 3.4 in page 44).

```
-9-   initial_second_order();
-10-  y = state_second_order();
-11-  derivative_second_order(z);
```

The edges of the *second_order* segment-link digraph are afterwards inserted in the submodel digraph of the calling program (figure 3.11).

The segment calls as well as the sorting relationships between them, already shown in figure 3.5, are retrieved from the model data base by the preprocessor at *non_linear_system* preprocessing time.

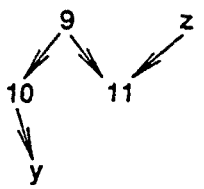

Figure 3.11: *Edges* (9, 10) *and* (9, 11) *are called hidden edges because they can not be gathered through the input-output variables in the segment calls. These edges represent the implicit sorting relationships between the segments of the called lower level* second_order *submodel.*

Rule T.2 : Sequences of `else if` fields of an `if` statement are replaced by a sequence of embedded `if` statements (see page 36). This replacement affecting the submodel digraph is useful for analysis purposes and will not appear in the preprocessor C object target code.

Let be, the following piece of code to illustrate the rule[7].

[7]This example is fully explained in section 3.5.

```
   ::::::::::::::
Dynamic region
        ::::::::::::::
       if(x<A) {
           fx = K1*(A-x);
       } else if (x>=B & (v>=0.0 | ac<0.0)) {
           fx = K2*(B-x)-C*v;
       } else {
           fx = 0.0;
       };
        ::::::::::::::
```

Code 3.5 *The symbols '&' and '|' are the* **and** *and* **or** *logical operators respectively.*

For analysis purposes, the `if` statement is transformed into a sequence of embedded `if` statements,

```
   ::::::::::::::
Dynamic region
        ::::::::::::::
       if(x<A) {
           fx = K1*(A-x);
       } else {
           if(x>=B & (v>=0.0 | ac<0.0)) {
                fx = K2*(B-x)-C*v;
           } else {
                fx = 0.0;
           };
        };
        ::::::::::::::
```

Code 3.6 *The replacement of* `else if` *fields for* `if` *statements is made to ease the submodel digraph representation and analysis.*

Rule T.3 : The transformed `if` statements (rule T.2) and `when` statements are decomposed into a statement for each associated discontinuous function and a statement for the event effect.

An edge from each discontinuous function vertex to its associated event effect is inserted in the submodel digraph (figure 3.12). Although the cause of an event is independent from its effect (see page 45) from the sorting perspective, these edges should be considered in the submodel initialization analysis (subsection 3.3.2).

Continuing with the example introduced in rule T.2, for every 'relational expression' embedded in the `if` declarative statement, its associated discontinuous function (`groot1`, `groot2`, `groot3` or `groot4`) is inserted in the

model code and the 'relational expression' is replaced by its related logical state (`lroot1, lroot2, lroot3` or `lroot4`).

```
        ::::::::::::::
      Dynamic region
            ::::::::::::::
-5-       groot1 = A-x;          \
-6-       groot2 = x-B;           |  discontinuous functions
-7-       groot3 = v;             |  (event causes)
-8-       groot4 = -ac;          /
-9-       if(lroot1) {                              \
-10-         fx = K1*(A-x);                          |
          } else {                                   | (event
-11-         if (lroot2 & (lroot3 | lroot4)) {       |  effects)
-12-             fx = K2*(B-x)-C*v;                  |
             } else {                                |
-13-             fx = 0.0;                           |
             };                                      |
          };                                        /
            ::::::::::::::
      End dynamic region;
        ::::::::::::::
```

Code 3.7 *The substitution of relational expressions by their associated discontinuous functions and logical states lets the analysis algorithms to put on one side code needed for discontinuous function computation and on the other side code needed for derivative computations.*

Construction rules

Static region rule:

Rule S.1 : A global symbol vertex is created in the submodel digraph for each variable declared in the static region. These symbol vertices are defined to be at level zero and they embrace different subsets as stated in subsection 3.2.2.

Initial region rules:

Rule I.1 : A single executable vertex is flagged as initial if it is associated to the initial region.

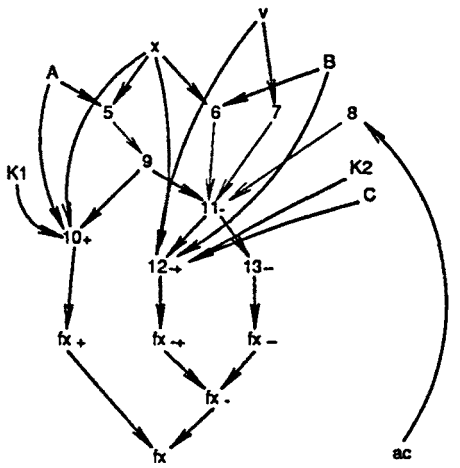

Figure 3.12: *Edges* $(5,9)$, $(6,11\_)$, $(7,11\_)$ *and* $(8,11\_)$ *are only used in the initialization analysis.*

Rule I.2 : A local symbol vertex is inserted in the submodel digraph for each variable initialized in the initial region. This symbol vertex is flagged as initial.

The initial local symbol vertices will be labelled using the variable symbol plus the extension (0). Hence, if the state variable x is initialized in the initial region, the associated symbol vertex will be labelled $x(0)$.

Rule I.3 : In the submodel digraph the following directed edges are created.

- From the initial executable vertex to the local symbol vertices.
- From each initial local symbol vertex to its associated global symbol vertex.

Rule I.4 : Edges from input, parameter and constant global symbol vertices related to the initial region to the initial executable vertices are included in the submodel digraph.

Code 3.8 is part of the static and the initial regions of the *pilot_ejector* submodel.

```
Continuous submodel pilot_ejector is
   Static region
      inputs {real vel_of_aircraft,altitude;}
      outputs{real horiz_disp,vert_disp;}
      parameters {
         real
           pilot_mass      = 7.0,  drag_coefficient = 1.0,
           pilot_drag_area = 10.0, eject_velocity   = 40.0,
           eject_deg_angle = 15.0, rail_height      = 40.0;
         }
      constants {
         real g                  = 32.2,
              deg_per_radian     = 57.2958;
         }
      tabular 2d functions {
         real air_density = (0.0,2.377E-3)...
                            (60000.0,0.2238E-3);
      }
      functions called { real sin,cos,polar;}
      state {real vel,flight_angle, horiz_disp, vert_disp;}
      auxiliar variables {real eject_angle,horiz_velocity,
                          vert_velocity,
                          drag_at_altitude,...;}
   End static region;
   Initial region
      horiz_disp     = 0.0;  vert_disp      = 0.0;
      flight_angle   = 0.0;  vel            = 0.0;
      eject_angle    = eject_deg_angle/deg_per_radian;
      horiz_velocity = vel_of_aircraft-eject_velocity*
                       sin(eject_angle);
      vert_velocity  = eject_velocity*cos(eject_angle);
      vel, flight_angle  = polar(horiz_velocity,
                                 vert_velocity);
      drag_at_altitude   = 0.5*drag_coefficient*
                   pilot_drag_area*air_density(altitude);
   End initial region;
     ..............
     ..............
End submodel pilot_ejector;
```

Code 3.8 *Procedural blocks are represented with single executable vertices. This restricts the possibility of checking the procedural code through the submodel digraph analysis. For this reason, other techniques should be applied to analyze the procedural code.*

Figure 3.13 represents the vertices and edges of the *pilot_ejector* submodel digraph related with the initial region.

Declarative assignment statements rule:

Rule D.1 : For every assignment declarative statement, an executable vertex is inserted in the submodel digraph. Declarative assignment statements are those declarative statements which are not discontinuous (`if` and `when` statements).

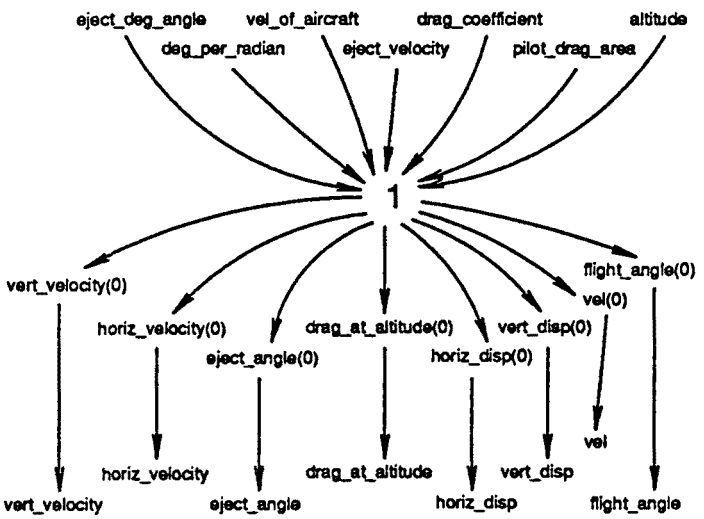

Figure 3.13: *The initial region is represented with the single executable vertex* 1. *For each variable initialized in the initial region, an initial symbol vertex is defined* (Vs_n = $\{vert\_velocity(0), ..., flight\_angle(0)\}$).

Let be the declarative statement,

```
ac = fm/M;
```

Previously to the analysis of this statement, the variables ac and fm and the parameter M already have global symbol vertices associated to them. To include in the submodel digraph the relations involved in the statement the executable vertex associated to the statement is created first; afterwards the directed edges connecting the symbol vertices with the executable vertex are added (figure 3.14).

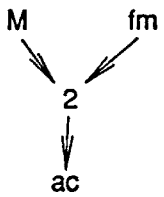

Figure 3.14: *Since global symbol vertices are present in the submodel digraph before the syntactic analysis of the initial and dynamic regions, edges from executable vertices to global symbol vertices can be directly inserted in the submodel digraph.*

When rules:

The code into a when statement has to be executed once at implicit solution points of the ODE.

Several strategies are possible:

- To define a specific segment for the when clauses. This code would be called only at event occurrences governed by the discontinuous segment.
- To cluster the when clauses in the discontinuous segment.
- To cluster the when clauses in the ODE segment (state or algebraic or derivative).

The selection of any of the above strategies influences the design of the sequence of calls to the discontinuous and ODE segments from the ODE solver at implicit solution points.

Rule W.1 : A when executable vertex is assigned to each *when* clause.

This is shown in the following code:

```
-1-  Initial region
        start  = 0.0;
        y      = FALSE;
     End initial region;
     Dynamic region
       ::::::::::
-11-    groot2 = ramp-1.0;
```

```
-12-    when(lroot2) {
           start  = start+period;
           y      = TRUE;
        };
        ::::::::::
```

Code 3.9 *A when executable vertex is assigned for each* when *clause.*

whose relations are represented in figure 3.15.

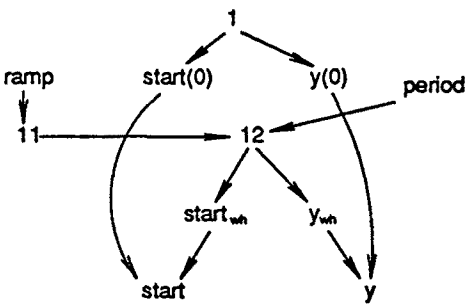

Figure 3.15: *Vertex number* 12 *belongs to the set of when executable vertices* ($12 \in Ve_{wh}$ *and* $\Gamma(12) \in Vs_{wh}$).

Rule W.2 : For each variable defined in a when statement a new when local symbol vertex is inserted in the submodel digraph. The when symbol vertices will be labelled using the symbol plus the extension *wh*.

Rule W.3 : For each input variable to the when clause an edge from its associated global symbol vertex ($v_i \in Vs_{gl}$) to the when executable vertex ($v_j \in Ve_{wh}$) is created in the submodel digraph if $v_i \notin \Gamma^2(v_j)$.

Rule W.4 : If:

$$\{v_j, v_k\} \in \Gamma^{-1}(v_i)$$

and

$$v_i \in Vs,\ v_j \in Ve,\ v_k \in Vs_{wh}$$

then the edge $(v_j, \Gamma^{-1}(v_k))$ must be inserted in the submodel digraph in order to ensure the proper sequencing of the declarative code.

To emphasize the above rule, submodel codes 3.10 and 3.11 modelling a pulse generator can be considered. In order both to behave properly it is mandatory that vertex 4 is executed before vertex 3, see code 3.12. To achieve it, according

to the above rule an edge from vertex 4 to vertex 3 has to be added in the submodel digraph in figure 3.16.

```
Dynamic region
   y    = FALSE;
   ramp = (si_time-start)/per;
   when(ramp>=1.0) {
      start = start+per;
      y     = TRUE;
   };
End dynamic region;
```

Code 3.10 *Impul submodel code.*

```
Dynamic region
   ramp = (si_time-start)/per;
   when(ramp>=1.0) {
      start = start+per;
      y     = TRUE;
   };
   y    = FALSE;
End dynamic region;
```

Code 3.11 *Impul submodel code.*

```
   Dynamic region
-1-   ramp = (si_time-start)/per;
-2-   groot1 = ramp-1.0;
-3-   when(lroot2) {
         start = start+per;
         y     = TRUE;
      };
-4-   y    = FALSE;
   End dynamic region;
```

Code 3.12 *The relations between procedural statements included in the* when *clause are not represented in the submodel digraph. Other techniques should be applied to analyze in more detail this procedural code.*

If rules:

In contrast with other languages restrictive definition of the `if` declarative statement [Crosbie86a], the `if` declarative statement defined in the *MUSS* language allows declarative blocks (see page 36). In a fast reading through the below construction rules it can be observed that it is more difficult to properly represent the relations involved into the *MUSS* `if` statement than in other languages restrictive `if` statements.

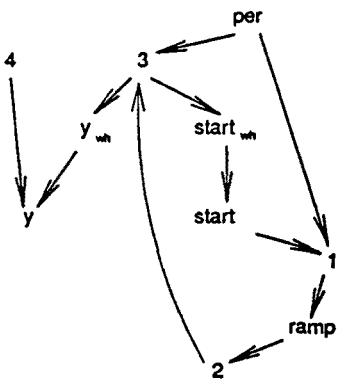

Figure 3.16: *If vertex number* 4 *is executed after the when executable vertex number* 3, *the effect of the when vertex over the local symbol vertex* y_{wh} *will not be reflected in the global symbol vertex* y.

Rule F.1 : An *if executable vertex* is assigned to each `if` statement.

Rule F.2 : The *executable vertices* within an `if` block are labelled with the vertex number followed by an extension coding the TRUE-FALSE sequence leading to it. Notice that in general some executable vertices can be `if` or `when` executable vertices.

An `if` statement selects, in general, one of the two declarative blocks embedded in the statement. The first one is chosen if the logical expression related to the statement is true. Otherwise, the second declarative block is selected. To distinguish the executable vertices included in the first declarative block from those in the second one, the executable vertices related to the true state are flagged with a "+" extension. Executable vertices related to the false state are flagged with a "-" extension. This can be recursively extended to embedded sequences of `if` statements (see code 3.13 and figure 3.17).

Rule F.3 : Edges are created from each if executable vertex to the executable vertices associated to the `if` blocks.

Therefore, the set of successors of an if executable vertex ($\Gamma(v_i)$, $v_i \in Ve_{if}$) corresponds with the set of executable vertices included within the `if` statement.

As an example, the submodel digraph associated to the following code is represented in figure 3.17.

```
          ::::::::::::::
      Dynamic region
               ::::::::::::::
-3-           if(lroot1) {
-4-               fx = K1*d;
-5-               d  = (A-x);
              } else {
-6-               if (lroot2)) {
-7-                  d  = (B-x);
-8-                  fx = K2*d-C*v;
                  } else {
-9-                  fx = 0.0;
                  };
              };
-10-          fm  = fe+fx;
               ::::::::::::::
      End dynamic region;
          ::::::::::::::
```

Code 3.13 *Blocks included in an* `if` *statement are declarative.*

In this figure, it can be seen that the set of successors of the if executable vertex number 3 are:

$$\Gamma(3) = \{4_+, 5_+, 6_-\}$$

which correspond to the vertices associated to the statements included in the `if` statement.

Moreover, vertex number 6_- is an if vertex. Thus,

$$\Gamma(6_-) = \{7_{-+}, 8_{-+}, 9_{--}\}$$

Rule F.4 : A local symbol vertex is created in the submodel digraph for each variable defined into the `if` declarative blocks. This if symbol vertices as well as the edges from the executable vertices defining them are labelled with the name of the symbol followed by an extension like the one defined in rule F2 (see code 3.13 and figure 3.17).

Rule F.5 : If a if symbol vertex such as x_{+---+} is created in the submodel digraph, the if symbol vertices x_{+---}, x_{+--}, x_{+-} and x_+ have to be created too as well as the corresponding edges: (x_{+---+}, x_{+---}), (x_{+---}, x_{+--}), (x_{+--}, x_{+-}), (x_{+-}, x_+) and (x_+, x) —see page 71—.

In figure 3.17, the edges (fx_+, fx) and (fx_-, fx) represent that

$$fx = fx_+ + fx_-$$

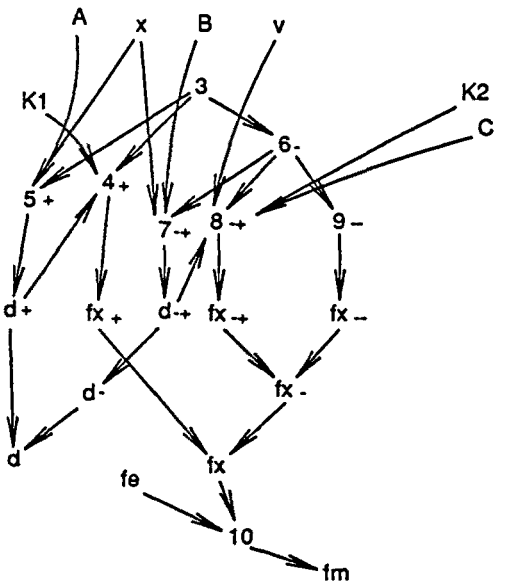

Figure 3.17: *The submodel digraph vertices can be classified in:*

- *Global symbol vertices*

$$Vs_{gl} = \{x, v, fx, fm, d, fe, K1, K2, C, A, B\}$$

- *Local symbol vertices*

$$Vs_{lo} = \{d_+, d_{-+}, d_-, fx_+, fx_{-+}, fx_{--}, fx_-\}$$

- *If executable vertices:* $Vs_{if} = \{3, 6_-\}$.

- *Declarative executable vertices "included" in the if executable vertex* 3*:* $\Gamma(3) = \{4_+, 5_+, 6_-\}$ *and* 6_-*:* $\Gamma(6_-) = \{7_{-+}, 8_{-+}, 9_{--}\}$

Note that the executable vertex 4_+ *has as input the local symbol vertex* d_+ *instead of* d*. This is because* d_+ *is defined in the same declarative block.*

and given that

$$fx_{-} = fx_{-+} + fx_{--}$$

it results

$$fx = fx_{+} + fx_{-+} + fx_{--}$$

Rule F.6 : In case an input variable to a statement in an `if` block has a if symbol vertex *in the same block* associated to it, an edge is included from the if symbol vertex to the executable vertex.

Otherwise, an edge from the global symbol vertex to the executable vertex is included.

3.3 Submodel digraph analysis

Different aspects regarding the robustness of simulation software have been described in [Elzas79a] an expanded later in [Cellier84a]. These aspects can be grouped and summarized as follows:

- *Simulation languages:* the simulation language should follow a structured approach compatible with structure elements of the real system under investigation. Furthermore, the simulation language definition must contain sufficient redundancy so that the software is able to detect as many programming errors as possible. Moreover, the use of formal grammars to define the simulation language is strongly requested.

- *Simulation compilers:* the simulation compiler should perform extensive error testing while parsing the application program. Furthermore, the simulation compiler should ensure that syntactic errors will not propagate to the next compiler stage. Moreover, the use of formal grammars increases the robustness of simulation compilers with respect to their maintainability.

- *Simulation run-time systems :* the user should not be forced to bother about details related to the run-time system implementation. Furthermore, the numerical algorithms should detect numerical instabilities, properly report numerical errors and work for a large set of applications.

- *Simulation data:* perhaps the most important concept for enhancing the robustness related to the simulation data is the use of relational simulation data bases to store and retrieve simulation and experimental results.

- *Simulation systems:* The term simulation system denotes the union of simulation language, simulation compiler, simulation run-time system, simulation data and the documentation. System documentation must be updated in parallel with program code improvement.

- *Model robustness:* Consists in ensuring that a particular application program performs properly. It is a consequence of the mathematical abstraction methodology used and the robustness of the simulation software.

In the design of the *MUSS* system and special care has been given to its robustness:

- The hierarchical structure of the *MUSS* simulation language is suitable for the division of the real system into subsystems.

- Redundancy is introduced in the submodel code. For example, the user is forced to declare all the submodel variables.

- The use of LALR(1) grammars to specify the *MUSS* language increases the robustness of the *MUSS* preprocessor with respect to its maintainability.

- The *MUSS* preprocessor ensures that syntactical errors will not propagate from the preprocessor to the *C* compiler stage.

- The *MUSS* preprocessor performs extensive error checking looking for model consistency and completeness.

- *MUSS* relies on reputed numerical algorithms increasing the robustness of the run-time system (*MUSS* simulation environment).

- Model and simulation data bases fall into the robustness of the data handling mechanisms.

This section is devoted to the submodel digraph analysis techniques employed in the *MUSS* preprocessor to check the model consistency and completeness.

In the analysis of the submodel digraph four functional main phases can be distinguished:

1. *Code consistency checking:* This stage, which can partially be carried on in parallel with the submodel digraph construction, looks at the code consistency.

2. *Submodel dynamic initialization analysis:* In this phase, besides checking that the submodel can be properly initialized, the executable code needed for initializing the current submodel (this includes the discontinuous functions initializations) and the called lower level submodels is grouped into the *initial segment.*

3. *Discontinuous function computations analysis:* During this phase, the code needed to evaluate the discontinuous functions of the current submodel and those in the called lower level submodels is grouped into the *discontinuous segment.* Moreover, discontinuous functions are classified in order to generate run code reducing the time overhead at event occurrences.

4. *Dynamic computations:* During this phase, the code needed to calculate derivatives is grouped into the *ODE segment* (Ordinary Differential Equations segment).

Before continuing, some more definitions are needed:

Definition 3.6 *A* segment digraph $Sg = (Sv, Se)$ *is defined as a subdigraph of the submodel digraph.*

Definition 3.7 *Following run-time structural requirements, tree types of segment digraphs can be defined. According to the definition 3.6 this segment digraphs can overlap and this will be often the case.*

- **Initial segment digraph** $(Ig = (Iv, Ie))$*: Includes all the executable vertices needed to initialize the current submodel and the lower level submodels.*

- **Discontinuous segment digraph** $(Eg = (Ev, Ee))$*: Includes the executable vertices needed for evaluating the discontinuous functions of the current submodel and those of the lower level submodels.*

- **ODE segment digraph** $(Og = (Ov, Oe))$*: Clusters the executable vertices involved in the computation of the derivative and output variables of the current submodel as well as the derivative vertices associated to the called lower level submodels*[8].

[8]Auxiliary vertices that do not contribute to discontinuous, derivative or submodel output computations are included in this segment. The related code may only be executed at communication intervals. When executable vertices are also added to this segment but the associated code has to be executed at event occurrences.

3.3.1 Code consistency checking

What follows, is a preliminary list of error conditions which may be used in *MUSS* for testing the source code consistency.

(1) A parameter or a constant can not be updated in the initial or dynamic region. Thus, an error occurs if

$$\Gamma^{-1}(v_i) \neq \emptyset \text{ and } v_i \in Vs_p$$

or

$$\Gamma^{-1}(v_i) \neq \emptyset \text{ and } v_i \in Vs_c$$

(2) A parameter or constant declared in the static region should be used as input in some statement. Hence, a warning error should be issued if

$$\Gamma(v_i) = \emptyset \text{ and } v_i \in Vs_p$$

or

$$\Gamma(v_i) = \emptyset \text{ and } v_i \in Vs_c$$

In addition, each variable in the submodel input list should be used as an input to a statement. Consequently, the warning error condition is,

$$\Gamma(v_i) = \emptyset \text{ and } v_i \in Vs_i$$

(3) All the declared submodel variables —excluding state variables and those in the input list— should be used as output in some statement. A warning error occurs if

$$\Gamma^{-1}(v_i) = \emptyset \text{ and } v_i \in Vs - (Vs_i \cup Vs_s \cup Vs_p \cup Vs_c)$$

(4) A symbol vertex can not be defined by two or more executable vertices. Therefore, there is some error if

$$|\Gamma^{-1}(v_i) \cap Ve| \geq 2 \text{ and } v_i \in Vs$$

This error condition is emphasized in the following example.

```
          ::::::::::::::
       Dynamic region
-1 -          y = x*A;
               ::::::::::::::
-5 -          if(lroot1) {
-6 -              fx = K1*(A-x);
```

```
                } else {
-7 -                fx = 0.0;
                    ::::::::::::::
-10-                fx = K2*(x-B);
                };
-11-            y = cos(z);
                 ::::::::::::::
        End dynamic region;
           ::::::::::::::
```

The submodel digraph related to the piece of code written above is represented in figure 3.18.

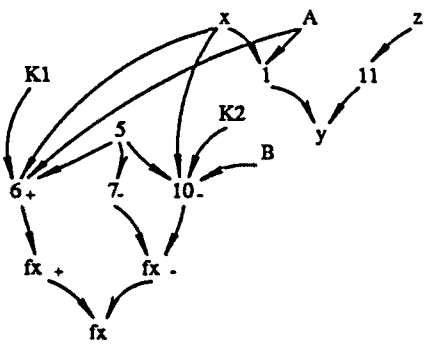

Figure 3.18: *Code associated to the submodel digraph is inferred to be incorrect because the variable y has a double definition. Furthermore, the variable $fx\_$ defined for the FALSE state of the* `if` *statement has a double definition, too.*

The submodel code is erroneous because:

$$|\Gamma^{-1}(y) \cap Ve| = |\{1, 11\}| = 2 \text{ and } y \in Vs$$

and

$$|\Gamma^{-1}(fx\_) \cap Ve| = |\{7\_, 10\_\}| = 2 \text{ and } fx\_ \in Vs$$

(5) A submodel variable can not be simultaneously defined in an `if` clause and in a declarative assignment statement outside the `if` clause.

Given a symbol vertex v_i, an error is inferred when,

$$\Gamma^{-1}(v_i) \cap Vs_{if} \neq \emptyset$$

and

$$\Gamma^{-1}(v_i) \cap Ve \neq \emptyset$$

The following code, whose submodel digraph is represented in figure 3.19 is wrong because the variable y is defined twice over time spans.

```
     Dynamic region
-1-     if(...) {
-2-        y = ...;
        } else {
-3-        y = ...;
        };
-4-     y = ...;
     End dynamic region;
```

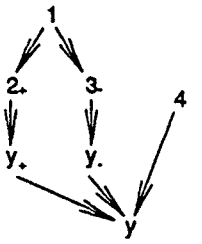

Figure 3.19: *A variable can not be defined in an* `if` *clause and simultaneously in a declarative statement placed outside the* `if` *clause.*

In this case

$$(\Gamma^{-1}(y) = \{y_+, y_-, 4\}) \cap Vs_{if} = \{y_+, y_-\} \neq \emptyset$$

and

$$(\Gamma^{-1}(y) = \{y_+, y_-, 4\}) \cap Ve = \{4\} \neq \emptyset$$

(6) Variables defined in an `if` clause, and used outside it should be defined for every execution path through the `if` clause.

Hence, given

$$v_i \in Vs_{gl} \text{ and } (v_i \in Vs_o \cup Vs_d \text{ or } \Gamma(v_i) \cap Ve \neq \emptyset)$$

and v_j belonging to the set of if local symbol vertices weakly connected to v_i then,

a warning error occurs if

$$|\Gamma^{-1}(v_i) \cap Vs_{if}| = 1$$

or

$$|\Gamma^{-1}(v_j) \cap Vs_{if}| = 1$$

The error condition is illustrated in the following example:

A way to code a zero-order hold with a logical input control signal could be,

```
Continuous submodel zero_order_hold is
      Static region
            inputs {logical logical_variable; real x,IC;}
            outputs{real y;}
      End static region;
 -1-  Initial region
            y  = IC;
      End initial region;
      Dynamic region
 -2-        if(logical_variable) {
 -3-            y  = x;
            };
      End dynamic region;
End submodel zero_order_hold;
```

Code 3.14 *Zero_order_hold submodel.*

Where the associated submodel digraph is represented in figure 3.20.

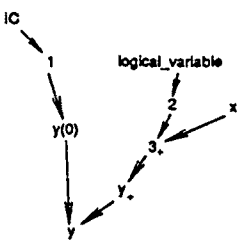

Figure 3.20: *The analysis algorithm checks that variables are defined for each logical state of a* `if` *discontinuous statement. In this case, variable* `y` *is only defined for the TRUE state.*

It can be seen that the variable y is only defined for the TRUE state of the `if` statement. In this case the code is right but in other cases it might be wrong. Therefore, a warning message error may help the users to detect programming errors.

In this case a warning error will be given because

$$y \in Vs_o \; and \; |(\Gamma^{-1}(v_i) = \{y(0), y_+\}) \cap Vs_{if}| = |\{y_+\}| = 1$$

However, notice that variables not defined in an `if` clause for all its execution paths but not used outside the `if` clause are not suspect of an error condition. This is shown in the next example,

```
      Dynamic region
-1-     grootl = x-A;
-2-     if(lrootl) {
-3-        fx = K1*x1;
-4-        x1 = (A-x);
        } else {
-5-        fx = 0.0;
        };
      End dynamic region;
```

The related submodel digraph is represented in figure 3.21.

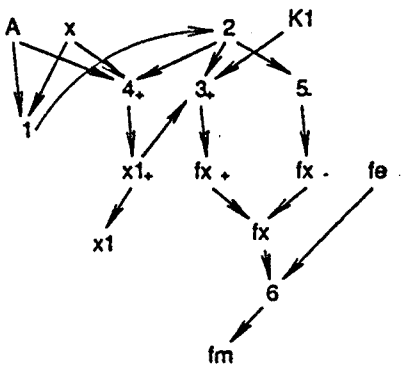

Figure 3.21: *Variable $x1$ is only used in a declarative block embedded in the `if` statement. Therefore, its use is always correct although it is not defined for all the logical states of the `if` declarative discontinuous statement.*

In this case $x1$ is only defined and used[9] in the same declarative block which is connected to the TRUE state. So, a warning error will not be issued to the user.

[9]In this example, $x1$ is not an output, neither a derivative variable

(7) If the error condition (6) fulfills for a global symbol vertex v_i, then a fatal error is issued if

$$\Gamma^{-1}(v_i) \cap Vs_n = \emptyset$$

As the reader may have observed in code 3.14 and its associated submodel digraph represented in figure 3.20, variables not defined in an `if` clause for all its execution paths and used outside it must be initialized in the initial region.

3.3.2 Submodel dynamic initialization analysis

The model (submodel) initialization can be divided in two parts:

- *Static initialization:* Consists on setting up constants when instances of continuous models are created.

  At preprocessing time as well as at run-time, a test is undertaken to check if all the submodel parameters and constants have been initialized. At run-time, the simulation system forces the user to initialize all the submodel parameters and constants not set beforehand. See section 2.3 at page 25 for details concerning data passing mechanisms between subprograms.

- *Dynamic initialization:* Consists on executing the initial segment of each continuous submodel and setting up the logical states associated to each discontinuous function.

  The value of the variables of the initial segment in the submodel input list must be initialized from a higher level initial segment. Since the dynamic segment of the experiment is also partitioned into the inital, discontinuous and ODE segments, the initial segment of the experiment may initialize input variables of the called initial segments.

This subsection deals with the submodel consistency checking (state vector and value of the discontinuous functions) at initial time. The submodel digraph simplifications rules are stated first, thereafter, the steps of the initialization analysis are exposed.

Simplification rules:

- If there is an edge from a local initial symbol vertex Vs_n to a global symbol vertex, all other edges concurrent to this symbol vertex are removed:

  $\forall v_k \in Vs_n$, the edges $(v_j, \Gamma(v_k))$ where $v_j \neq v_k$ are not considered (see figure 3.22).

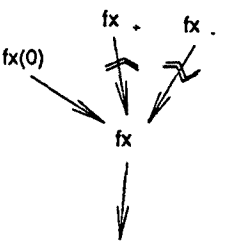

Figure 3.22: *The global symbol vertex fx is defined in the initial region. Consequently, the edges (fx_+, fx) and (fx_+, fx) are meaningless for the submodel initialization analysis.*

- When occurrences at initialization time can not afford. Therefore, actions associated to when statements do not contribute to the submodel initialization:

  Edges (v_i, v_j) such as $v_i \in Ve_{wh}$ or $v_j \in Ve_{wh}$ are removed.

Notice that edges from discontinuous executable vertices to if executable vertices are preserved. These edges are needed for the analysis because the initialization depends on the discontinuous function values.

Steps of the analysis:

In the analysis steps, two different sets can be distinguished: the first concerning the completeness of the submodel initialization which is achieved through the submodel digraph (steps 1 to 5); the second regarding the initial region (whose code will be included into the initial segment) initialization (steps 6 and 7) using specific techniques for procedural code.

(1) Check that all the state variables have been initialized in the initial region:

  For every state symbol vertex $v_i \in Vs_s$, $|\Gamma^{-1}(v_i) \cap Vs_n|$ must be equal to one.

(2) A variable defined in a when clause and used (input of a statement) outside the when clause must be initialized in the initial region if it is not defined in a declarative assignment statement. Thus, if $v_j \in \Gamma(v_i)$ such as $\Gamma^{-1}(v_j) \cap Ve = \emptyset$, $v_i \in Vs_{wh}$ and $v_j \in (Vs_o \cup Vs_d)$ or $\Gamma(v_j) \in Ve$, a vertex $v_k \in \Gamma^{-1}(v_j)$ should exists such as $v_k \in Vs_n$, otherwise there is an initialization error.

  Let be the submodel code[10] which models a square wave train:

[10]This submodel is presented in more detail in appendix C in page 193.

```
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 3.15 *Pulse_width_modulator submodel.*

The transformed code will be:

```
-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-      groot1 = ramp-ratio;
```

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

The initialization of y and ***start*** variables is necessary because (figure 3.23):

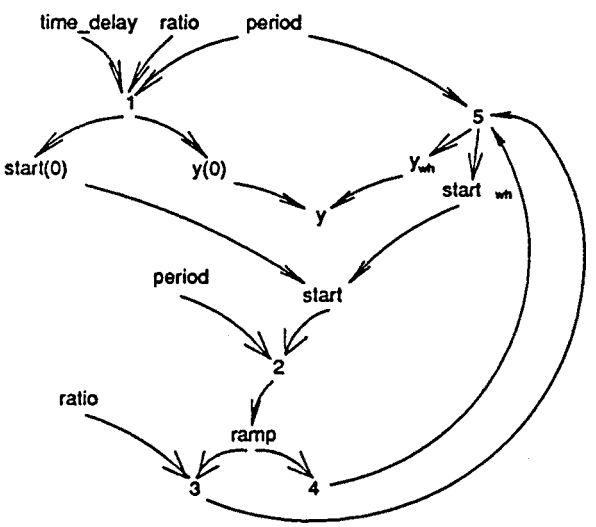

Figure 3.23: *Pulse-width modulator submodel digraph. Note that variables* ***start*** *and* y *must be initialized at initial time, otherwise they will be undefined until an event activates the* when *statement.*

- $y \in \Gamma(y_{wh})$ and $\Gamma^{-1}(y) \cap Ve = \emptyset$, $y_{wh} \in Vs_{wh}$ and $y \in Vs_o$.
- $\boldsymbol{start} \in \Gamma(\boldsymbol{start}_{wh})$ and $\Gamma^{-1}(y) \cap Ve = \emptyset$, $\boldsymbol{start}_{wh} \in Vs_{wh}$ and $\Gamma(\boldsymbol{start}) \in Ve$.

(3) Check that the logical states associated to the discontinuous functions of the submodel can be set at initial time:

The logical states can be set at initial time if the discontinuous vertices Ve_{gf} and their predecessors ($Q(Ve_{gf})$) do not belong to directed cycles.

A good example is presented in the case study of section 3.5 (see code 3.20 and figure 3.29 in pages 99 and 100). In the example, if the variable fx would not be initialized, the discontinuous vertex 8 would belong to the directed cycles formed by vertices

$$\{8, 11\_, 12\_+, fx\_+, fx\_, fx, 14, fm, 2, ac, 8\}$$

or vertices

$$\{8, 11\_, 13\_\_, fx\_\_, fx\_, fx, 14, fm, 2, ac, 8\}$$

and therefore the discontinuous functions would not be able to be evaluated at initial time.

(4) Check that the called lower level initial segments can be executed at initial time:

This case is similar to the previous one: the lower level initial segments can be executed at initial time if its initial executable vertices and their predecessors ($Q(Ve_n)$) do not belong to directed cycles.

In figure 3.24 in page 80, it can be seen that initial vertex 4 can be executed at initial time because there is an initial condition for $x0$ and therefore the edge $(10, x0)$ has been removed.

(5) If the above conditions fulfill, the next step consists in building the initial segment digraph which has to include the following vertices:

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

and the initial segment will be composed by:

- Statements associated to Ive executable vertices.
- Input variables associated to Ivs_i input symbol vertices.

(6) Check that for every execution path through the initial region, all the variables which are inputs to statements of the dynamic region will be initialized.

This objective as well as the next one can not be achieved through the submodel digraph analysis, they can be attained using techniques more suitable for procedural code [Richards78a].

(7) Being the code in the initial region, procedural, a test can be made to be sure that no variable is used before its initialization.

Aspects of the initial segment structure:

Let be the *MUSS* submodel code 3.16, where the `if` statement is used to select one of two submodels:

```
Continuous submodel dummy is
      Static region
            inputs {real y;} outputs{real x;}
            parameters {real ul=0.0, ll=-4.0, tau=1.0;}
            auxiliary variables {real x0;}
            submodels called {real_pole; limiter;}
      End static region;
      Initial region
            x0  = 3.0;
      End initial region;
      Dynamic region
           if(y>=0.0) {
                x = real_pole(x0,tau,y);
            } else {
                x = limiter(ll,ul,y);
            };
            x0 = x;
      End dynamic region;
End submodel dummy;
```

Code 3.16 *Dummy submodel. In many models the set of active equations changes at certain event occurrences. This is the case of events that bear a change in the physical system (pilot_ejector model) or the case of events that convey a change in the set of equations in order to improve the numerical behaviour (orbit and nuclear engineering models).*

The *MUSS* transformed code is shown in code 3.17.

```
-1-   Initial region
            x0  = 3.0;
      End initial region;
      Dynamic region
-2-         grootl  = y;
-3-         if(lrootl) {
-4-             initial_real_pole(x0);
-5-             derivative_real_pole(tau,y);
-6-             x = state_real_pole();
            } else {
```

```
-7-              initial_limiter(ll,ul,y);
-8-              discontinuous_limiter(ll,ul,y);
-9-              x = algebraic_limiter(ll,ul,y);
              };
-10-          x0 = x;
      End dynamic region;
```

Code 3.17 *At each event occurrence, the initial segment of the selected* `if` *block must be executed to properly initialize the associated submodel.*

The *dummy* submodel digraph is represented in figure 3.24.

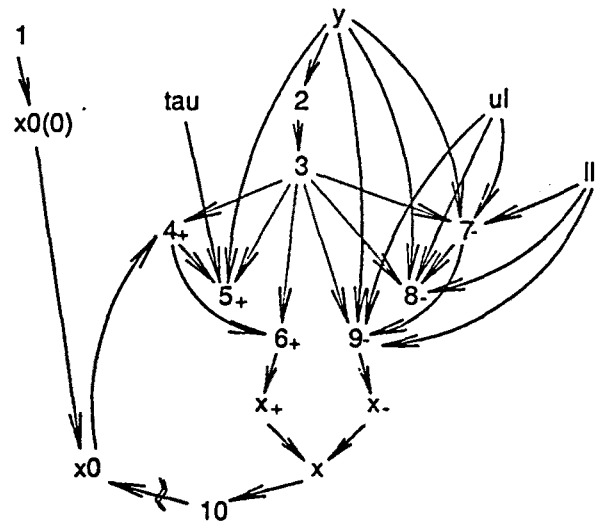

Figure 3.24: *Dummy submodel digraph. Edge* $(10, x0)$ *is ignored in the initialization analysis. Though vertices* 4_+ *and* 7_- *belong to* Ve_n *(*$Ve_n = \{1, 4_+, 7_-\}$*), only one of these vertices will be executed at initial time.*

The initial segment of the submodel *dummy* has to include the code associated with the submodel digraph executable vertices $\{2, 3, 4_+, 7_-\}$. This code will allow the computation of the executable vertices 4_+ or 7_- at initial time.

```
x0 = 3.0;
lroot1 = (groot1 = y) >= 0.0;
if(lroot1) {
    initial_real_pole(x0);
} else {
    initial_limiter(ll,ul,y);
};
```

At initial time, both submodels (`if` blocks) may be able to be initialized although only the call to the lower level initial segment of the lower level submodel selected (*real_pole* or *limiter*) will be executed. Afterwards, at event occurrences, the initial segment of the lower level submodels called in the `if` declarative blocks have to be executed depending on the discontinuous functions states.

Before performing a simulation experiment, a check must be undertaken to establish if the initial set of the defined logical states for all discontinuous functions is consistent. If it is not, an iterative method can be used to search for a consistent set of states.

The responsibility of defining the proper set of initial conditions is left to the analyst.

3.3.3 Discontinuous function computations analysis

In the dynamic initialization analysis, in fact, the completeness of the computations related with the logical states associated to the discontinuous functions has been checked and the initial segment elements have been pointed out as well.

The stress in this subsection is in how to build up the discontinuous segment (*g-root* computations). Concerning the necessary transformation rules to build the logic around the root-finder and the discontinuous segment tightly related with the ODE solver, only the guidelines are commented.

The discontinuous segment:

The purpose of the discontinuous segment is to cluster the code needed to compute the discontinuous functions of the submodel and called lower level submodels.

To achieve this objective, the submodel digraph has first to be simplified according to the following rules:

- Edges (v_i, v_j) such as $v_i \in (Vs_{wh} \cup Ve_{wh})$ or $v_j \in (Vs_{wh} \cup Ve_{wh})$ are rejected.
- Edges (v_i, v_j) such as $v_i \in (Vs_n \cup Ve_n \cup Ve_g)$ are rejected. These edges are only used in the submodel initialization analysis.

The discontinuous segment digraph has to include the following vertices:

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

and the discontinuous segment will be composed by:

- Statements associated to Eve executable vertices.
- Input variables associated to Evs_i input symbol vertices.

The discontinuous functions analysis:

The end objective of the analysis is to generate a well-conditioned code to properly handle the discontinuities.

To achieve it, two main aspects can be distinguished:

- The recognition of the discontinuous function type according to some taxonomy.
- Which is the right action to drive the integration properly after the event occurrence.

Steps 1,2 and 3 are mainly concerned with the first one whereas step 4 emphasizes the second one.

(1) According to the discontinuous *MUSS* statement type, a distinction of the discontinuous functions can be done in two classes:

1. Discontinuous functions attached to `if` statements.
   Notice that the *MUSS* statement

   ```
   l = y >= 0.0;
   ```

   involves a discontinuous function belonging to this class. It is equivalent to

   ```
   if(y>=0.0){l=TRUE;}else{l=FALSE;};
   ```

and it will be decomposed at preprocessing time into:

```
groot1 = y;
l = lroot1;
```

2. Discontinuous functions attached to `when` statements.

The main difference between both discontinuous statement types is in the event occurrence handling.

When a root of an `if` attached discontinuous function is found and the new logical state is set, only a single call to the ODE segment is needed to propagate the event effect across the ODE segment.

In contrast, since variables which are output of `when` statements are handled like state variables (i.e. memory variables), two calls to the ODE segment are needed to propagate the effect of a `when` associated event.

(2) Depending on the event effect the discontinuous functions can be grouped in:

1. Those involving state variables or derivative variables.
2. Those involving only discontinuous functions.
3. Those which do not modify state variables, neither derivatives, neither discontinuous functions.

Although an integration restart may always be done to resume the integration after an event occurrence, a best choice, which is supported by modern ODE solvers, seems to be:

- If the event belongs to the first class, a restart is necessary to proceed with the integration.
- If the event belongs to the second class, only a call to the discontinuous segment is necessary to resume the integration.
- If the event belongs to the third class, integration continues —if it is possible— from the internal mesh point.

However, to embody a discontinuous function in one of the above mentioned groups may require additional information. This is the case when the effect of an event propagates through an output variable to a higher level submodel or to a called lower level submodel through an input variable of the lower level submodel.

A deeper knowledge about submodels interfaces allows to achieve the right classification of the discontinuous functions.

(3) Another analysis objective consists on finding the discontinuous functions which only depend on the simulation time because their associated events can be handled like time events [Ellison81a].

If input variables contribute to the discontinuous function evaluations, the identification of the discontinuous functions as time-dependent-only should be hold until the calling higher level submodel(s) is(are) preprocessed.

(4) To improve the code a useful analysis objective consists in detecting, in cascaded `if (else if)` constructions, the meaningless *if* branches (*g-roots*) associated to the logic values (*l-roots*) of the higher level `if` constructs.

Notice that this analysis is achieved at *MUSS* source processing time.

This can be illustrated with the next example which is based on the case study presented in section 3.5.

Given the piece of dynamic code:

```
if(x<A) {
    fx = K1*(A-x);
} else if (x>=B & (v>=0.0 | ac<0.0)) {
    fx = K2*(B-x)-C*v;
} else {
    fx = 0.0;
};
fm  = fe+fx;
ac  = fm/M;
v'  = ac;
```

it is transformed at preprocessing time into the intermediate code:

```
 -1-    groot1 = A-x;
 -2-    groot2 = x-B;
 -3-    groot3 = v;
 -4-    groot4 = -ac;
 -5-    if(lroot1) {
 -6-       fx = K1*(A-x);
        } else {
 -7-       if (lroot2 & (lroot3 | lroot4)) {
 -8-            fx = K2*(B-x)-C*v;
           } else {
 -9-            fx = 0.0;
           };
        };
-10-    fm  = fe+fx;
-11-    ac  = fm/M;
-12-    v'  = ac;
```

It can be seen that:

- If the value of `lroot1` is TRUE, it is not needed to restart integration when a root in `groot2, groot3` or `groot4` is found.
- If the value of `lroot1` is FALSE and the value of `lroot2` is FALSE, it is not needed to worry about roots in `groot3` or `groot4` discontinuous functions.

As a consequence of the above analysis, the source code will be transformed into:

```
-1-     groot1 = A-x;
-2-     groot2 = x-B;
-3-     groot3 = v;
-4-     groot4 = -ac;
-5-     if(froot1 = lroot1) {
-6-        fx = K1*(A-x);
        } else {
-7-        if (froot2 = (lroot2 & (lroot3 | lroot4))) {
-8-            fx = K2*(B-x)-C*v;
           } else {
-9-            fx = 0.0;
           };
        };
-10-    fm  = fe+fx;
-11-    ac  = fm/M;
-12-    v'  = ac;
```

instead of the previous code.

Now, the condition to restart the integration is attached to a change on the state of `froot1` or `froot2`. To update `froot1` and `froot2`, a call to the ODE segment has to be made after a root is detected and the logical state (`lroot1,...,lroot4`) related to the root has been updated.

In this particular example, the number of integration restarts will be reduced in a 50%.

3.3.4 Dynamic computations

The continuous submodel dynamic code has to be splitted into the ODE and the discontinuous subprograms (segments) in order to interface it with the ODE solver packages (subsection 3.1.1, page 38).

In subsection 3.3.3 it has been shown how to build the discontinuous segment but the ODE segment construction has not been discussed yet.

Subsection 3.3.1 has mainly dealt with the dynamic code analysis, therefore the only remaining question is the clustering of the necessary code into the ODE segment.

In order to perform the above objective the submodel digraph has to be simplified and thereafter the ODE segment digraph has to be set.

Simplification rules:

- Directed edges issuing from initial executable vertices are removed. Therefore, given $v_i \in Ve_n$, the edges $(v_i, \Gamma(v_j))$ are suppressed.
- Directed edges issuing from initial local symbol vertices to their associated global symbol vertices are discarded. Thus, if $v_i \in Vs_n$ then edges $(v_i, \Gamma(v_i))$ are suppressed.
- Directed edges issuing from the discontinuous function vertices to their associated executable vertices are removed. Thus, if $v_i \in Ve_g$, the edges $(v_i, \Gamma(v_i))$ are suppressed.
- Directed edges issuing from when executable vertices to when local symbol vertices are removed. Thus, if $v_i \in Ve_{wh}$, the edges $(v_i, \Gamma(v_i))$ are suppressed.

The ODE segment:

The ODE segment has to assemble the code to compute the submodel output variables (those in the output list) as well as those needed at communication intervals (for recording purposes).

To achieve the above requirement, the following computations have to be performed: derivatives (state), when clauses, `if` block initialization clauses (see example in page 79) and output bounded clauses.

The set of executable vertices of the ODE segment will be:

- $Q(Vs_d) \cap Ve$: Vertices to compute the derivative symbol vertices (derivatives of the submodel).
- $Q(Ve_d) \cap Ve$: Vertices to compute the derivative vertices associated to calls to lower level derivative segments.

- $Q(Ve_{wh}) \cap Ve$: Vertices to compute the when executable vertices at event occurrences.
- $(\Gamma(Ve_{if}) \cap Ve_n)$: Vertices to compute the initial vertices which will have to be executed at event occurrences.
- $Q(Vs_o) \cap Ve$: Vertices to compute the output symbol vertices.
- $(Ve - (Eve \cup Ve_n)) - Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{wh})$: Vertices which only will be executed at communication points.

The ODE segment digraph includes the following vertices:

- *Executable vertices:*

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

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

The ODE segment will be composed by[11]:

- Statements associated to Ove executable vertices.
- Input variables associated to Ovs_i input symbol vertices.
- Output variables associated to Ovs_o output symbol vertices.

3.4 Submodel sorting

As Clancy [Clancy65a] states, the development of a sorting method by Stein [Stein60a] was an important step towards the design of more powerful and flexible Continuous System Simulation Languages (CSSL). Since then, this feature has been provided by many widely used CSSL languages, such as: MIDAS (1964), DSL/90 (1964) and successors [Shah76a] [Syn85a] ,CSSL-IV [Nilsen83a] and ACSL [Mitchell82a].

The automatic sorting of the sentences makes free the user from the responsibility of ensuring a proper execution order of the simulation model code. This important feature should be supported, in our opinion, by modern simulation languages.

[11]In fact, the ODE segment (subprogram) is not created by the preprocessor. The ODE segment digraph is directly splitted into the state, algebraic and derivative segment digraphs.

3.4.1 Statement of the problem

The *MUSS* language has been conceived as declarative and its architecture hierarchical.

The sorting algorithm which has to convert the source code into a procedural one faces a problem not found in classical monolithic architectures, that of the *information loops.*

A sentence in the dynamic code in which a submodel is invoked is formally equivalent to an assignment statement: it has a set of input and output variables (the submodel interface). The difference arises from the fact that the coupling of the variables in the interface through the called submodel code is hidden to the preprocessor sorting procedure. If that procedure detects algebraic loops involving interface variables, the loop may be really algebraic —which would be the case if the above mentioned coupling is algebraic— or merely an *information loop.*

Known approaches to avoid information loops are:

- Handle the submodels as MACRO's. The statements of the called submodels will be spread over the statements of the calling submodel. In this case, all the submodels must be able to be retrieved in source form. Moreover, MACRO-like facilities should be provided. Furthermore, the time spent at preprocessing time increases because of the necessity of translating all the lower level submodels which have to be handled like MACRO's.

- Force either the submodel input variables or the submodel output variables to be all state type. This approach, in our opinion, is restrictive because the correspondence between the physical subsystem and the submodels in the hierarchical model can be lost.

- Force the user to separate the computation of the derivatives from the output computations which are assembled in a specific block in which the outputs only depend on state variables. The main objection to this approach is that the user is forced to bother about requirement imposed by restrictions in sorting capabilities. Moreover, a different type of submodel has to be defined for coding subsystems when the submodel output variables are algebraically related to the submodel input variables.

The method proposed next, based on the segmentation of the ODE segment, does not impose restrictions on the submodel architecture neither in the hierarchy

3.4.2 Definitions

The algorithm which will be shown in the next subsection is based on a segmentation of the ODE segment into subsegments (*state, algebraic* and *derivative* segments) which have an associated subdigraph which has to be built from the ODE segment digraph.

Based on subsection 3.2.2, the following definition can be stated:

Definition 3.8 *Three types of ODE subdigraphs are distinguished according to the nature of its executable vertices:*

- **State segment digraph** *($Sg = (Sv, Se)$): The executable vertices do not depend on input symbol vertices.*
- **Algebraic segment digraph** *($Ag = (Av, Ae)$): The executable vertices belong to input/output symbol vertices directed paths.*
- **Derivative segment digraph** *($Dg = (Dv, De)$): The executable vertices do not contribute to output computations.*

Straightly, from the preceding definition, the segments (subsegments) associated to the digraphs will be characterized by the following structural properties:

- *State segment:* output variables in it will depend only on parameters, constants or state variables. Therefore, they may not exist pure algebraic chains between input and output variables.
- *Algebraic segment:* it clusters the input-output algebraic computations.
- *Derivative segment:* computations involving output variables are not allowed. Derivative computations, when clauses and computations to be performed at communication intervals will be assembled in this segment.

Before coming into the algorithm exposition, some new definitions (founded on those in subsection 3.2.1) are needed:

Definition 3.9 *The of a vertex v_i in a digraph is the number of output vertices reachable from vertex v_i.* **weight, output**

$$Ow(v_i) = |R(v_i) \cap Vs_o|$$

Definition 3.10 *The of a vertex v_i in a digraph is the number of input vertices which can reach v_i.* **weight, input**

$$Iw(v_i) = |Q(v_i) \cap Vs_i|$$

3.4.3 An algorithmic solution

To get the aforesaid subdigraphs it is proposed an algorithmic method which comprises two major procedures:

- The *fusion* process conducing to the construction of the *reduced digraph* ($Org = (Orv, Ore)$).

  Definition 3.11 *A pair of vertices v_i and v_j are said to be* **fused**, *if both are replaced by a single new vertex such that every vertex that is incident (into or out of) on v_i or v_j or both is now incident on the new vertex* [Deo 1974a].

  Definition 3.12
  The reduced digraph is the result of applying the fusion rules to an ODE segment digraph. The most extended one, besides input an output symbol vertices, has n vertices: a fused-state vertex, a fused-derivative vertex and $n-2$ fused-algebraic vertices.

- The *backwards reconstruction* process which expands the reduced digraph into the segment digraphs.

Fusion

The method proposed to get the reduced digraph is an enhanced extension of that shown in [Guasch86a] whose main restrictions were:

- The analysis dealt with the dynamic code as a whole. The initialization and the discontinuous function computations analysis were not envisaged.
- `If` clauses were handled as single vertices. Code inside it was supposed to be procedural.

It has been previously seen that the *if* clause general representation in the digraph is no more a single vertex but comprises discontinuous vertices, if executable vertices and executable vertices (rules T.2, T.3 pp. 53 , 54). This will increase the complexity of the method to get the reduced digraph although it must be reminded that there has also been an enhancement in the `if` blocks which are now declarative.

The present scope of the `if` clause does not permit to apply the method in [Guasch86a] because the if executable vertices can not be directly handled as executable vertices which have to be executed before any successor executable vertex.

This is illustrated in the next example. Figure 3.25 represents the ODE segment digraph related to code 3.18.

```
Continuous submodel dummy_2 is
      Static region
            inputs {real in;}
            outputs{real z;}
            parameters {real p; logical log;}
            state {real r;};
             ::::::::::::::::::
      End static region;
      Dynamic region
            ::::::::::::
-1-         v = f1(p);
-2-         if(log) {
-3-            x = f2(v);
-4-            y = f3(v);
            } else {
-5-            x = 4.0;
-6-            y = f4(in);
            };
-7-         z = f5(x);
-8-         r' = f6(y);
      End dynamic region;
End submodel dummy_2;
```

Code 3.18 *Dummy_2 submodel.*

Using the method presented in [Guasch86a], the derivative segment digraph does not include the if executable vertex 2. This is incorrect because, even though vertex 2 is forced to be executed before vertices 4_+ and 6_-, the execution of these vertices in the derivative segment can not be separated from the logical decision associated to the if executable vertex which is taken in the state segment. Clearly, both segment digraphs might include the if executable vertex.

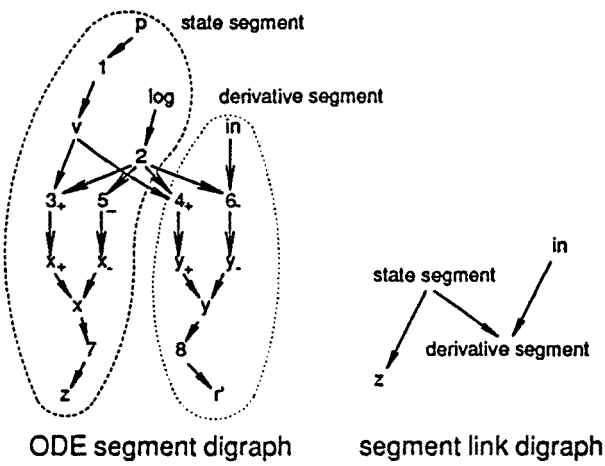

Figure 3.25: *The represented subdigraphs got from the ODE segment are incorrect because the derivative segment must include vertex 2.*

The Fusion steps

The following fusion steps are stated to get the the reduced digraph from the ODE segment digraph, they have to be applied in sequence to the vertices in the ODE digraph —the output state symbol vertices comprised— except to the input/output symbol vertices (remaining) which may not be found.

Step 1 : Fuse the weakly connected vertices that have the same output weight.

Step 2 : Once performed the previous step, fuse the weakly connected vertices that have the same input weight and whose output weight is different than zero.

Step 3 : Fuse the vertices having zero output weight.

Step 4 : Fuse the vertices having zero input weight.

Behind the conclusion of the fusion steps, the vertices in the reduced digraph which are not input or output symbol vertices (the fused vertices) are classified into the following types:

- *Fused-state vertex:* The vertex v_{fs} which has input weight equal zero.
- *Fused-derivative vertex:* The vertex v_{fd} which has output weight equal zero.

- *Fused-algebraic vertices:* Those vertices v_{fa} which belong to a directed path from an input symbol vertex to an output symbol vertex.

Backwards reconstruction

In the backwards reconstruction procedure, the state segment digraph ($(Sg = (Sv, Se)$), the derivative segment digraph ($(Dg = (Dv, De)$) and the algebraic segment digraphs ($(Ag = (Av, Ae)$) are built around the corresponding fused vertices of the reduced digraph.

The vertices composing each segment digraph will be:

- *Input symbol vertices:*

$$Vs_i^* = \Gamma^{-1}(v_i) \cap Vs_i \quad \forall v_i \in \{v_{fs}, v_{fd}, v_{fa}\}$$

For the state segment digraph:

$$v_i = v_{fs} \text{ and } Vs_i^* = \emptyset$$

- *Output symbol vertices:*

$$Vs_o^* = \Gamma(v_i) \cap Vs_o \quad \forall v_i \in \{v_{fs}, v_{fd}, v_{fa}\}$$

For the derivative segment digraph:

$$v_i = v_{fd} \text{ and } Vs_o^* = \emptyset$$

- Symbol and executable vertices.

The set of vertices which have been agglutinated in the corresponding fused vertex of the reduced digraph, as well as the if executable vertices which are predecessors in the ODE segment digraph but have been aggregated in other fused vertices.

Notice that the state segment digraph can include output state symbol vertices which have to be added to the output symbol vertices defined. Therefore, the output symbol vertices of the state segment digraph will be in general:

$$(\Gamma(v_{fs}) \cap Vs_o)$$

from the reduced digraph and

$$(Vs_o \cap Vs_s)$$

from the ODE segment digraph.

The ODE subsegments assembled from the associated sorted digraph will be:

- *State segment:* Composed by
  - Statements associated to Sve executable vertices.
  - Output variables associated to Svs_o output symbol vertices.
- *Algebraic segments:* Composed by
  - Statements associated to Ave executable vertices.
  - Input variables associated to Avs_i input symbol vertices.
  - Output variables associated to Avs_o output symbol vertices.
- *Derivative segment:* Composed by
  - Statements associated to Dve executable vertices.
  - Input variables associated to Dvs_i input symbol vertices.

Figure 3.26 represents the submodel digraph associated to *dummy_2* submodel (see code 3.18 in page 91). The vertices embraced by the dashed curved have output weight equal to one (notice that vertex z is an output symbol vertex) and the vertices embraced by the dot curve have the output weight equal to zero. Thus, after the first fusion step the reduced digraph is obtained (figure 3.26) because the successive fusion steps do not affect, in this case, the shape of the reduced digraph.

Through the backwards reconstruction, the segment digraphs are extracted:

- *Labelling of the fused vertices:*

$$Iw(f_1) = 0 \text{ and } Ow(f_1) \neq 0 \Rightarrow \; f_1 \text{ is the fused-state vertex.}$$

$$Ow(f_2) = 0 \Rightarrow \; f_2 \text{ is the fused-derivative vertex.}$$

- *State segment digraph:*

  Built around the fused-state vertex.

  Its vertices are:

  - *Input symbol vertices:* $Svs_i = \Gamma^{-1}(f_1) \cap Vs_i = \emptyset$

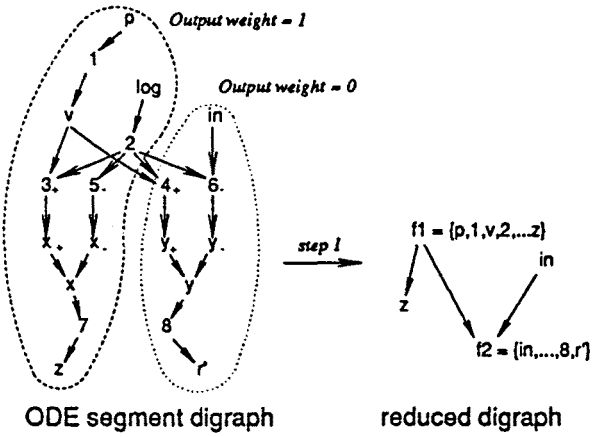

Figure 3.26: *The first fusion step consists on fusing vertices with equal output weight.*

- *Output symbol vertices:*

$$(\Gamma(f_1) \cap Vs_o) = \{z\}$$

and

$$(Vs_o \cap Vs_s) = \emptyset$$

then

$$Svs_o = \{z\}$$

- *Executable vertices:* those agglutinated in f_1.

$$Sve = \{1, 2, 3_+, 5_-, 7\}$$

- *Derivative segment digraph:*

Built around the fused-derivative vertex.

Its vertices are:

- *Input symbol vertices:* $Dvs_i = \Gamma^{-1}(f_2) \cap Vs_i = \{in\}$
- *Output symbol vertices:* $Dvs_o = (\Gamma(f2) \cap Vs_o) = \emptyset$
- *Executable vertices:* those agglutinated in $f_2 \Rightarrow \{4_+, 6_-, 8\}$ plus if executable vertices which are predecessors in the ODE segment digraph $\Rightarrow \{2\}$ Therefore,

$$Dve = \{2, 4_+, 6_-, 8\}$$

Segment-link digraph construction

Once the segment digraphs have been assembled (the fusion and backwards reconstruction processes have been performed) an aspect still remains pending: that of the proper sequence between the ODE subsegments (derivative, state and algebraic), the initial segment and the discontinuous segment.

The *segment-link digraph construction* objective is to establish the proper executable sequence minimizing the calls from a higher level submodel (Rule T.1 in page 53).

Definition 3.13 *A segment-link digraph represents the proper sequence between submodel segments (initial, discontinuous, state, algebraic and derivative segments). Its intrinsic vertices are: initial-segment, discontinuous-segment, algebraic-segment and derivative-segment.*

Two rules apply:

- As stated, the initial segment has to be executed first, at the beginning of a simulation run. Thus, an edge from the initial-segment vertex to all the remaining vertices has to be created in the segment digraph (figure 3.27).

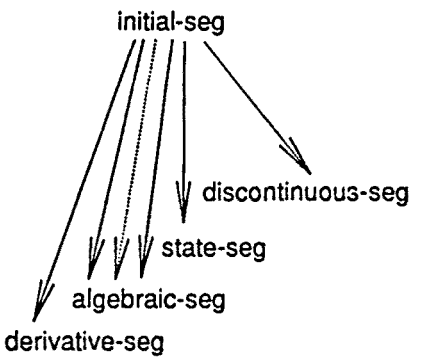

Figure 3.27: *Segment-link digraph construction. The first rule consists on creating edges from the initial-segment vertex to all the other vertices.*

- Concerning the state, algebraic and derivative segments, the proper sequence is explicit in the directed-paths between the fused vertices in the reduced digraph.

3.5 Case study

Figure 3.28 shows a spring-and-mass system, viscously damped by the dashpot shown, and containing the dead space represented by the gaps a and b [James67a]. When $t = 0$, the mass has zero displacement and an initial velocity of $0.2m/sec$, as shown. Considering the mass m as a free body in dynamic equilibrium, the differential equation of motion is

$$mx'' + cx' + f(x) = 0 \tag{3.1}$$

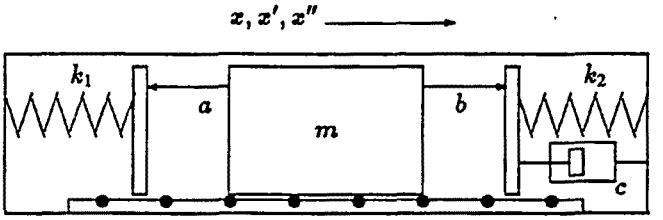

Figure 3.28: *Spring and mass system.*

where $f(x)$ represents the spring forces which acting on the mass. Examining the system, it can be seen that $f(x)$ is given by

$$\begin{aligned} f(x) &= k_2(x-b) && x > b \\ f(x) &= 0 && a \leq x \leq b \\ f(x) &= k_1(x-a) && x < a \end{aligned}$$

A MUSS submodel for simulating the spring-and-mass system follows:

```
Continuous submodel spring_and_mass_system is
      Static region
            inputs {real fe;}
            outputs{real x;}
            parameters {real K1,K2,A,B,M,V0,C;}
            state {real x;}
            auxiliary variables {real ac,fx,fm,v;}
            submodels called { integrator;}
      End static region;
```

```
        Initial region
              fx  = 0.0;
              x   = 0.0;
        End initial region;
        Dynamic region
              ac  = fm/M;
              v   = integrator(V0,ac);
              x'  = v;
              if(x<A) {
                  fx = K1*(A-x);
              } else if (x>=B & (v>=0.0 | ac<0.0)) {
                  fx = K2*(B-x)-C*v;
              } else {
                  fx = 0.0;
              };
              fm  = fe+fx;
        End dynamic region;
End submodel spring_and_mass_system;
```

Code 3.19 *Spring_and_mass_system submodel. The integrator submodel is shown in appendix C in page 177.*

The syntactic analyzer checks program correctness and translates MUSS source code into an intermediate code (code 3.20).

```
-1-   Initial region
          fx  = 0.0;
          x   = 0.0;
      End initial region;
      Dynamic region
-2-       ac  = fm/M;
-3-       initial_integrator(V0);
-4-       v = state_integrator();
-5-       derivative_integrator(ac);
-6-       x'  = v;
-7-       groot1 = A-x;
-8-       groot2 = x-B;
-9-       groot3 = v;
-10-      groot4 = -ac;
-11-      if(lroot1) {
-12-          fx = K1*(A-x);
          } else {
-13-          if (lroot2 & (lroot3 | lroot4)) {
-14-              fx = K2*(B-x)-C*v;
              } else {
-15-              fx = 0.0;
              };
```

```
          );
-16-      fm  = fe+fx;
     End dynamic region;
```

Code 3.20 *Spring_and_mass_system code expansion.*

The integrator submodel call is splitted into a call for each submodel segment (transformation rule T.1 in page 53). The sequence of `else if` fields is replaced by a sequence of embedded `if` statements (transformation rule T.2 in page 54). Discontinuous functions associated to `if` statements are pulled out into separate statements (transformation rule T.3 in page 55).

The *spring_and_mass_system* submodel digraph is represented in figure 3.29. The symbol vertices and the executable vertices can be grouped into the following subsets:

1. Subsets of Vs
   - *global vertices:* $Vs_{gl} = \{fe, x, x', K1, K2, A, B, M, V0, C, v, ac, fx, fm\}$
   - *parameter vertices:* $Vs_p = \{K1, K2, A, B, M, V0, C\}$
   - *constant vertices:* $Vs_c = \emptyset$
   - *input vertices:* $Vs_i = \{fe\}$
   - *output vertices:* $Vs_o = \{x\}$
   - *derivative vertices:* $Vs_d = \{x'\}$
   - *state vertices:* $Vs_s = \{x\}$
   - *local vertices:* $Vs_{lo} = \{fx(0), x(0), fx_+, fx_-, fx_{-+}, fx_{--}\}$
   - *initial vertices:* $Vs_{in} = \{fx(0), x(0)\}$
   - *when vertices:* $Vs_{wh} = \emptyset$
   - *if vertices:* $Vs_{if} = \{fx_+, fx_-, fx_{-+}, fx_{--}\}$
2. Subsets of Ve
   - *derivative vertices:* $Ve_d = \{5\}$
   - *discontinuous vertices:* $Ve_g = Ve_{gf} = \{7, 8, 9, 10\}$
   - *state vertices:* $Ve_s = \{4\}$
   - *initial vertices:* $Ve_{in} = \{1, 3\}$
   - *if vertices:* $Ve_{if} = \{11, 13\_\}$
   - *when vertices:* $Ve_{wh} = \emptyset$

The preprocessor analysis algorithms cluster the submodel code into the initial segment, the discontinuous segment and the ODE segment.

3.5.1 Initial segment

To get the initial segment digraph, the submodel digraph has to be simplified according the rule presented in page 74. Therefore, the edges (fx_+, fx) and (fx_-, fx) are not considered.

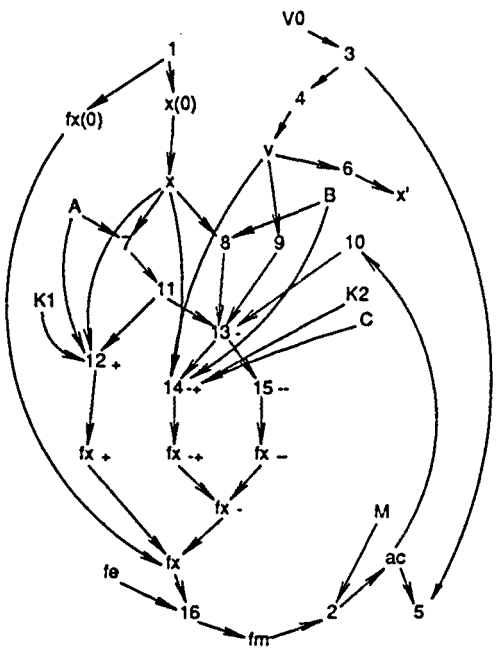

Figure 3.29: *Spring_and_mass_system submodel digraph.*

The most representative vertices of the submodel digraph are:

- *Executable vertices:*

$$Ive = Q(Ve_n \cup Ve_{gf}) \cap Ve = Q(\{1,3,7,8,9,10\}) \cap Ve =$$

$$\{1,3,7,8,4,9,16,2,10\}$$

In this case, the submodel can be initialized because the value of fx is known at initial time. Otherwise, an algebraic loop would appear in the initial segment digraph.

- *Input symbol vertices:*

$$Ivs_i = \Gamma^{-1}(Ive) \cap Vs_i = \{fe\}$$

- *Output symbol vertices:*

$$Ivs_o = \emptyset$$

The initial segment C target code includes the statements of code 3.21. Notice that the initial segment includes, besides statements associated to the initial regions, other statements included in the dynamic region of the submodel source code.

```
initial_spring_and_mass_system(fe)
::::::::::::::::::::
{
        fx   = 0.0;
        x    = 0.0;
        initial_integrator(V0);
        groot1 = A-x;
        groot2 = x-B;
        v = state_integrator();
        groot3 = v;
        fm   = fe+fx;
        ac   = fm/M;
        groot4 = -ac;
}
```

Code 3.21 *The spring_and_mass_system initial segment as well as the other segments are part of the pre-processor C object code. Because of symbolic access and uniqueness of variable names, the pre-processor C target code does not look like these segments but it is equivalent.*

The initial segment of each submodel instance is executed before each simulation run. After, the front-end integration and discontinuity finding algorithm sets the logical states (`lroot1,...,lroot4`) associated to each discontinuous function.

3.5.2 Discontinuous segment

Once again, the submodel digraph has to be simplified according the rules presented in the discontinuous function computation analysis (subsection 3.3.3 in page 81). In this case, the edges that should not be considered are:

- Edges (v_i, v_j) such as $v_i \in (Vs_n)$: $(fx(0), fx)$ and $(x(0), x)$.

- Edges (v_i, v_j) such as $v_i \in (Ve_n)$: $(1, fx(0))$, $(1, x(0))$, $(3,4)$ and $(3,5)$.
- Edges (v_i, v_j) such as $v_i \in (Ve_g)$: $(7,11)$, $(8,13\_)$, $(9,13\_)$ and $(10,13\_)$.

The most representative vertices are:

- *Executable vertices:*

$$Eve = Q(Ve_g) \cap Ve = Q(\{7,8,9,10\}) \cap Ve =$$

$$\{7,8,4,9,11,12_+,13\_,14_{-+},15_{--},16,2,10\}$$

- *Input symbol vertices:*

$$Evs_i = \Gamma^{-1}(Eve) \cap Vs_i = \{fe\}$$

- *Output symbol vertices:*

$$Evs_o = \emptyset$$

The discontinuous segment C target code includes the statements of code 3.22. The discontinuous segment is called by the discontinuity finding algorithm to precisely locate discontinuities.

```
discontinuous_spring_and_mass_system(fe)
::::::::::::::::::::::
{
        groot1 = A-x;
        groot2 = x-B;
        v = state_integrator();
        groot3 = v;
        if(lroot1) {
           fx = K1*(A-x);
        } else if (lroot2 && (lroot3 || lroot4)) {
           fx = K2*(B-x)-C*v;
        } else {
           fx = 0.0;
        }
        fm  = fe+fx;
        ac  = fm/M;
        groot4 = -ac;
}
```

Code 3.22 *Spring_and_mass_system discontinuous segment.*

3.5.3 ODE segment

To get the ODE segment digraph, the submodel digraph has to be simplified according the rules presented in the derivative computations analysis (subsection 3.3.4 in page 85). In this case, the edges that should not be considered are:

- Edges (v_i, v_j) such as $v_i \in (Ve_n \text{ and } v_i \notin \Gamma(Ve_{if}))$: $(1, fx(0))$, $(1, x(0))$, $(3,4)$ and $(3,5)$.
- Edges (v_i, v_j) such as $v_i \in (Vs_n)$: $(fx(0), fx)$ and $(x(0), x)$.
- Edges (v_i, v_j) such as $v_i \in (Ve_g)$: $(7, 11)$, $(8, 13\_)$, $(9, 13\_)$ and $(10, 13\_)$.

The most representative vertices of the ODE segment are:

- *Executable vertices:*

$$Ove = (Q(Ve_d \cup Vs_d \cup Vs_o \cup Ve_{uh}) \cap Ve) \cup (\Gamma(Ve_{if}) \cap Ve_n)$$
$$\cup (Ve - (Eve \cup Ve_n)) = (Q(5, x', x) \cap Ve) \cup \emptyset \cup \{5,6\} =$$
$$\{4,6,11,12_+,13\_,14_{-+},15_{--},16,2,5\} \cup \emptyset \cup \{5,6\} =$$
$$\{4,6,11,12_+,13\_,14_{-+},15_{--},16,2,5\}$$

- *Input symbol vertices:*

$$Ovs_i = \Gamma^{-1}(Ove) \cap Vs_i = \{fe\}$$

- *Output symbol vertices:*

$$Ovs_o = Vs_o = \{x\}$$

The ODE segment digraph is represented if figure 3.30.

```
ODE_spring_and_mass_system(x,fe)
::::::::::::::::::::::
{
        v = state_integrator();
        if(lroot1) {
           fx = K1*(A-x);
        } else if (lroot2 && (lroot3 || lroot4)) {
           fx = K2*(B-x)-C*v;
        } else {
```

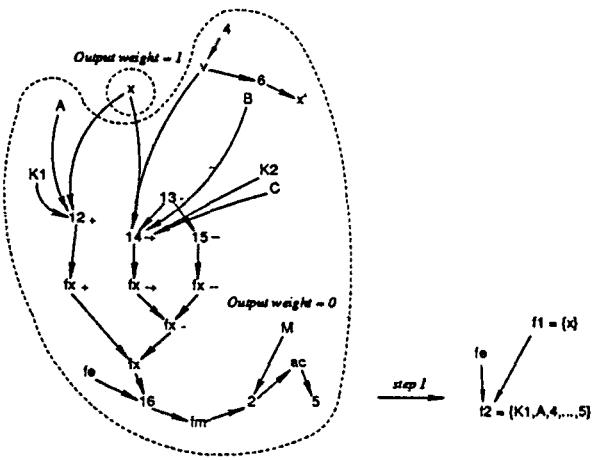

Figure 3.30: *Spring_and_mass_system ODE segment digraph and associated reduced digraph.*

```
        fx = 0.0;
    }
    fm  = fe+fx;
    ac  = fm/M;
    dx  = v;  /* x' = v */
    derivative_integrator(ac);
}
```

Code 3.23 *Spring_and_mass_system ODE segment.*

The ODE segment includes the statements of code 3.23. Nevertheless, it is not created by the preprocessor because the ODE segment digraph is splitted into a set of subdigraphs (state, algebraic and derivative segment digraphs) in order to solve the hierarchical sorting problem. For each subdigraph, a segment is created by the preprocessor.

The reduced digraph is obtained through the fusion steps explained in page 92. In the analysis of the reduced digraph, two segment digraphs are got, the state segment digraph and the derivative segment digraph (see figure 3.30).

State segment digraph

Built around the fused-state vertex $f1$.

Its vertices are:

- *Input symbol vertices:* $Svs_i = \Gamma^{-1}(f_1) \cap Vs_i = \emptyset$
- *Output symbol vertices:*

$$(\Gamma(f_1) \cap Vs_o) = \{x\}$$

and

$$(Vs_o \cap Vs_s) = \emptyset$$

then

$$Svs_o = \{x\}$$

- *Executable vertices:* those agglutinated in f_1.

$$Sve = \emptyset$$

The state segment created by the preprocessor is shown in code 3.24.

```
state_spring_and_mass_system(x)
::::::::::::::::::::::::
()
```

Code 3.24 *Spring_and_mass_system state segment.*

In this case, the state segment only pass the state variable x to the calling submodel.

Derivative segment digraph:

Built around the fused-derivative vertex $f2$.

Its vertices are:

- *Input symbol vertices:* $Dvs_i = \Gamma^{-1}(f_2) \cap Vs_i = fe$
- *Output symbol vertices:* $Dvs_o = (\Gamma(v_i) \cap Vs_o) = \emptyset$

- *Executable vertices:* those agglutinated in f_2

$$Dve = \{4, 6, 11, ..., 5\}$$

The derivative segment created by the preprocessor is shown in code 3.25.

```
derivative_spring_and_mass_system(fe)
::::::::::::::::::::::::
{
        v = state_integrator();
        if(lroot1) {
           fx = K1*(A-x);
        } else if (lroot2 && (lroot3 || lroot4)) {
           fx = K2*(B-x)-C*v;
        } else {
           fx = 0.0;
        }
        fm  = fe+fx;
        ac  = fm/M;
        dx  = v;  /* x' = v */
        derivative_integrator(ac);
}
```

Code 3.25 *Spring_and_mass_system derivative segment.*

3.6 Summary and conclusions

Summary

The whole procedure of the continuous submodel analysis is summarized in figure 3.31:

- Applying the transformation rules to the submodel source code the submodel digraph is set up (subsection 3.2.3).
- The analysis of the submodel digraph contributes to the robustness of the simulation code through a deep checking looking for inconsistencies (subsections 3.3.1 and 3.3.2).

  Furthermore, the initial, discontinuous and ODE segment digraphs are identified in order to get the suited interface with the ODE solvers (subsections 3.3.2, 3.3.3 and 3.3.4).

- The fusion and backwards reconstruction procedures achieve the state segment, derivative segment and algebraic segment digraphs construction.

  The stated procedures ensure that information loops will not appear and establish the proper execution sequence between the segments associated to the above mentioned digraphs (section 3.4).

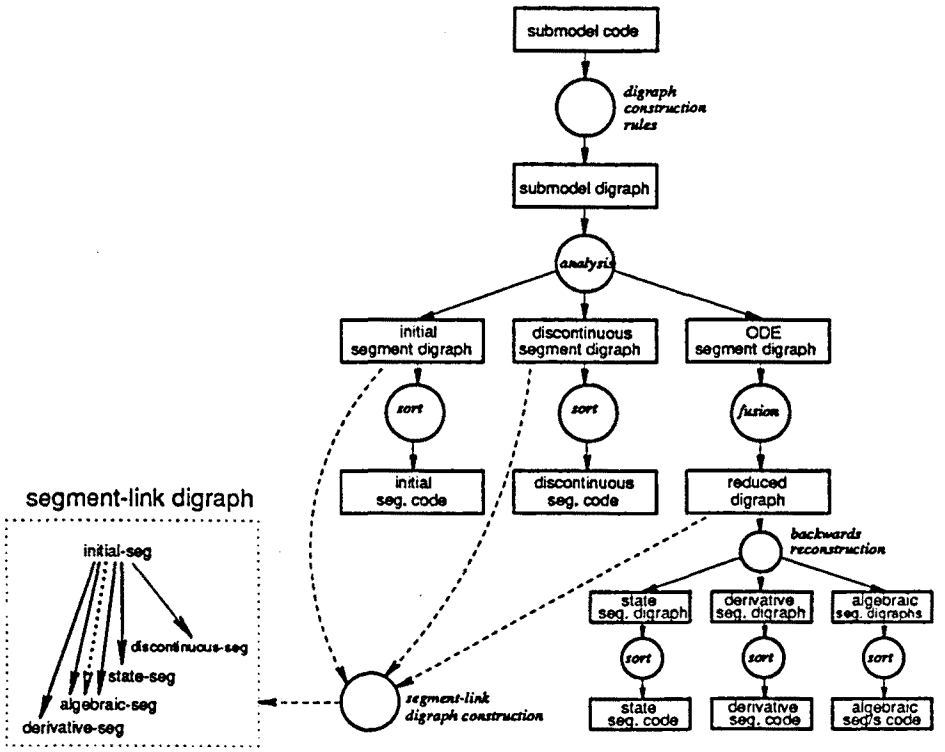

Figure 3.31: *The continuous submodel analysis: processes, digraphs and codes.*

- The intermediate code associated to each segment is sorted applying well known algorithms as the depth-first search [Tarjan72a] to the corresponding digraphs (initial, discontinuous, state, derivative, algebraic).

- The proper calling sequence to the segments in a lower level submodel is ensured by the segment-link digraph (subsection 3.4.3 in page 96).

Conclusions

The submodel digraph shows to be a powerful concept which can be used to increase the robustness of the submodel. Based on it:

- Consistency errors can be detected.
- It is possible to check if there are enough initial conditions to set the state vector and to evaluate the discontinuous functions at initial time.
- The submodel code is splitted into the initial, discontinuous and ODE segments. This segmentation is consistent with the functional tasks involved in a simulation run:
  1. Initialize the model.
  2. Compute the model derivatives.
  3. Locate discontinuities.

The subdivision of the ODE segment into the state, algebraic and derivative segments solves the hierarchical sorting problem. This approach is more general that those available in the literature.

The proposed method does not impose restrictions to the definition of new statements (`do, case, ...`) in the declarative grammar block.

In the next chapter, the *MUSS* simulation environment is presented. Special emphasis is made on the model hierarchical structure and some of the problems that will be solved are:

- Instantiation of submodels and models.
- Symbolic access to all the model variables.
- Management of the simulation environment.

Chapter 4

The simulation environment

"Software engineering has proven useful in reducing the cost of developing large and complex software systems and improving the quality of the resulting product. Since many simulation models are both large and complex, simulation-oriented programming languages should be designed to support software engineering techniques."

[Golden85a]

4.1 Introduction

In chapter 2 we have presented the MUSS architecture which has in the highest level three types of blocks: studies, experiments and submodels. A model is a set of hierarchical submodel blocks.

Following the generally accepted software engineering principles proposed by Oren and Zeigler [Ören79a], the experimentation with models has to be completely separated from models themselves. The architecture of *MUSS* goes one step forward separating experiments from studies increasing the modularity of the simulation environment. The concept of modularity is one of the most important concepts of structured programming [Golden85a]. Nevertheless, modularity alone is not enough to produce well designed programs but it helps to increase the reliability of simulation software.

In this chapter, we will analyze the structure of the *MUSS* simulation environment.

- First, we present the structure of models, which are made of hierarchical submodel blocks. The main problems to be solved are: symbolic access to the values

of the submodel variables, memory management of the submodel instances and model dynamic initialization.

- Second, we present the structure of the experiment and study blocks. The structure of an experiment block is similar to that of a submodel block: static, initial and dynamic regions.

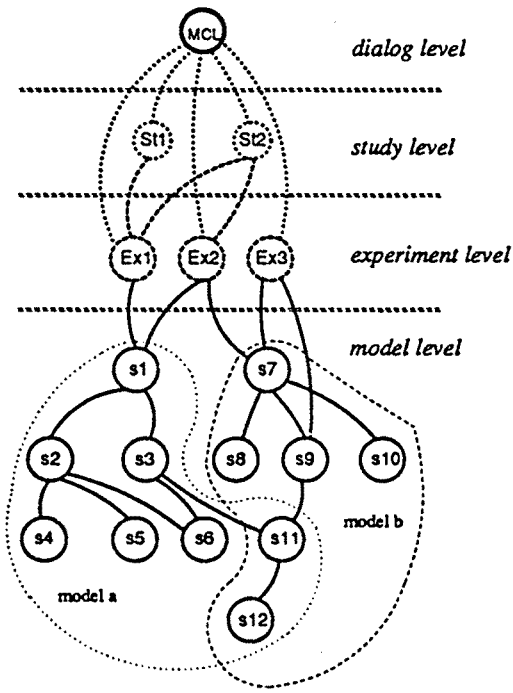

Figure 4.1: *User defined interactive simulation environments may have four hierarchical levels : dialog, study, experiment and model levels. From the dialog level experiment or study instances of any experiment or study present in the environment can be created and activated for execution.*

- Last, the *MCL* (*MUSS Command Language*) is described. An user defined simulation environment may include a large number of models, experiments, studies and data files. Thus, a good management of the environment is very important. To achieve that goal, the *MCL* —see Appendix C— has been designed in order to provide the users with a friendly interface with the *MUSS* environment. *MCL* can be seen as the monitor of the simulation environment.

Figure 4.1 represents models, experiments and studies in an user defined interactive simulation environment.

The models appear at the lowest level of the hierarchy. They are composed by a hierarchical set of submodel blocks. From the user perspective a model is a tree of submodels (i.e. submodel *s11* called from submodel *s3* does not model the same physical subsystem as submodel *s11* called from from submodel *s9*). From the implementation point of view, the submodel hierarchy can better be handled as a digraph.

In the next bottom-up level of the hierarchy, experiment blocks appear. Experiment blocks may call zero, one or more models. Its goal is to control a run.

The next-up level may include study blocks. Its objective is the control of model experimentation (i.e. optimization, identification, sensitivity analysis).

The *dialog level* is on top. In the dialog level the *MCL* language is used to communicate with the simulation environment. Through it, information about any lower level block can be got.

4.2 Model structure

Figure 4.2 represents a user defined simulation environment embracing *SMPR* (appendix C) and *non_linear_system* models. The continuous model SMPR (non linear system) is the combination of the SMPR (non linear system) submodel and connected lower level submodel blocks through directed paths. The model name is taken from that of the higher level submodel. In fact, model is a relative concept which depends on the users perspective. The strict use of the word *model* appears in an experiment context.

The above mentioned figure emphasizes that in this example each model has one model instance (copy of the model) in the environment. When a model instance is created, they have to be generated as many instances as existing directed paths from the higher level submodel to them. For example, *real_pole* submodel has two instances.

A continuous model can be seen as a *continuous process*. A continuous process instance may be created, executed and destroyed explicitly by the user[1]

[1]Although *MUSS* language is a piecewise continuous system simulation language, it has been designed keeping in mind, as has been explained in subsection 5.2 , a future extension to combined simulation languages. Thus, process instances will be created, destroyed, executed and suspended implicitly during each simulation run.

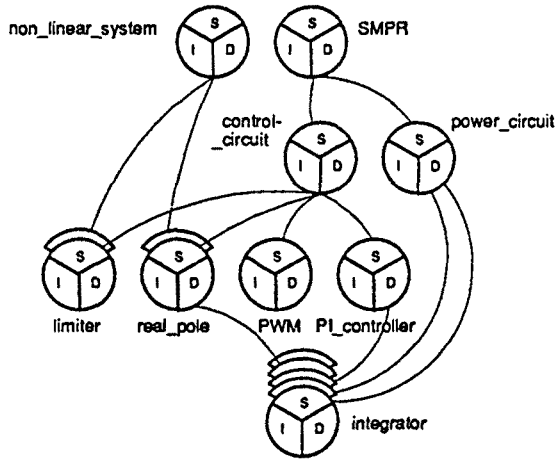

Figure 4.2: *User defined simulation environment embracing SMPR and non linear system models.* **S**, **I** *and* **D** *represent the static, initial and dynamic regions of each submodel. Here, the CSSL-like integrator submodel to update state and derivative variables has been used, but, in* MUSS *practice, state variables would be defined in any submodel.*

The structure of *MUSS* models has been designed to solve the main problems related to the hierarchical modelling approach and the separate compilation of submodels of the *MUSS* simulation system. These problems are:

- *Reentrance:* a private data storage area must be allocated and handled for each submodel instance. See in figure 4.2 that the *integrator* submodel has five independent storage areas, one for each submodel instance:

  1. `non_linear_system.real_pole.integrator`
  2. `SMPR.control_circuit.real_pole.integrator`
  3. `SMPR.control_circuit.PI_controller.integrator`
  4. `SMPR.power_circuit.integrator`
  5. `SMPR.power_circuit.integrator`

  Each submodel instance is identified by a unique directed path.

- *Symbolic access:* The present *MUSS* prototype allows symbolic access to all submodel variables and access to the submodel information stored in the model data base.

- *Dynamic memory management:* each piecewise continuous model can also be seen as a *generic model* but does not actually occupy data storage. An instance may be created, called and destroyed implicitly. To create and destroy a model instance implies allocation and deallocation of data storage private to each submodel instance.

- *Model initialization :* the static initialization is performed when instances of continuous processes are created and the dynamic initialization is performed before each each simulation run (experiment) when the initial segments are executed.

The approach in *MUSS* to solve the above problems is based on:

- A *submodel data structure:* A *submodel data structure* is defined from the submodel block at preprocessing time, which is stored for further use by the definition routines. This submodel data structure keeps the information needed for the symbolic access of the submodel variables. Furthermore, it stores the amount of memory needed for each submodel instance.

- A *definition digraph:* To create a model instance or to access information concerning submodels or its instances it is needed to access all the submodel data structures associated to the submodels included in the hierarchical model. The definition digraph keeps the links between the submodel data structures.

  Each node of the definition digraph is a submodel data structure and the structure of the definition digraph is equivalent to the structure of the model hierarchy.

  The definition digraph is used by the *MCL* executive to perform the dynamic allocation of the models.

- An *initialization sequence:* the initial segment of each submodel is preprocessed in such a way that it holds the calls to lower level submodel segments in the proper sequence for the dynamic initialization.

- A *dynamic sequence:* the state, derivative, algebraic and discontinuous segments of each submodel access lower level segments in the appropriate order.

4.2.1 Submodel data structure

The submodel data structure holds information about the submodel static block characteristics, i.e. its contents does not depend on the relative situation of the submodel in the model hierarchy and it is also independent on its state (activated or not).

```
struct submodel {
        char  *name;
        char  *date;
        char  *library;
        int   N_symbols;
        int   reentrance
        int   memory[N_TYPES];
        char  **symbols;
        int   **attributes;
        float **ranges;
        float **dimensions;
        int   (*init_address)();
        int   (*init_parameters)();
        struct lower_submodels *head;
};
```

Code 4.1 *Submodel data structure defined in C language.*

The submodel data structure, see code 4.1, has the following members which are filled by the preprocessor:

- *name* : Pointer to the submodel symbolic name.
- *date* : Pointer to the preprocessing date.
- *library* : Pointer to the library name in the model data base where the information (that one recorded in the source code and some other) about the submodel is stored.
- *N_symbols* : Total number of symbols in the submodel.
- *memory[N_TYPES]* : Stores the amount of memory area which is needed for each data type that must be allocated for each submodel instance. Six different data-type areas have been defined:

  0 : state variables.

  1 : derivative variables.

  2 : discontinuous functions that handle the discontinuities.

  3 : real variables.

  4 : integer variables.

  5 : logical variables.

- *symbols* : Pointer to the table of pointers to submodel symbols.
- *attributes* : Pointer to the table of attributes. The table of attributes has an 'element' for each variable. The fields in the 'element' are filled by integers and can be coded in a more compact way than the standard for integers.
- *ranges* : Points to the table where maximums and minimums of the screened variables will be stored. The range of the values of the submodel variables can be specified as part of a submodel in order to impose some run-time consistency checking.
- *dimensions* : Points to the table of dimensions. This table stores the number of dimensions of all the indexed variables and its range.

A second group of fields in the submodel data structure are set by the definition routine and will be explained in more detail in subsection 4.2.2:

- *init_parameters* : Pointer to the function that initializes the submodel parameters and constants (static initialization).
- *init_address* : Pointer to the function that assigns memory positions to submodel variables.
- *head* : Pointer to a linked list whose elements point to a lower level submodel data structure.

The last field is:

- *reentrance* : At run time, the number of active instances will be stored in this member of the submodel data structure.

In the following example the submodel data structure associated to *real_pole* submodel block (see figure 4.2) is shown.

Submodel code is:

```
Continuous submodel real_pole is
      Static region
            inputs  {real ic,tau,x;}
            outputs {real y;}
```

```
        End static region;
        Dynamic region
              y = integrator(ic,(x-y)/tau);
        End dynamic region;
    End submodel real_pole;
```

Code 4.2 *Real_pole submodel.*

The preprocessor object (C target) code related to the submodel data structure is:

```
* ------------------------------ real_pole       */
#define N_SYMBOLS 5
static readonly char *symbols[N_SYMBOLS] = {
        "x","y","x0","tau","dy"
};
static readonly int attributes[N_SYMBOLS*5] = {
        3,1,3,0,0, 3,2,4,0,0, 3,3,3,0,0, 3,4,3,0,0,
        3,5,0,0,0
};
static struct submodel real_pole = {
        "real_pole",          /* name       */
        "OCT27861205",        /* date       */
        "musssil",            /* library    */
        N_SYMBOLS,            /* N_symbols  */
        -1,                   /* reentrance */
        0,0,0,5,0,0,          /* memory[..] */
        symbols,              /* symbols    */
        attributes            /* attributes */
;
```

Code 4.3 *Real_pole data structure (C code) and associated declarative statements.*

Ranges and *dimensions* are not defined in this data structure because in this case all the variables are dimensionless and without limitations in the range of their values.

The relationship between the submodel data structure and the auxiliary data tables is emphasized in figure 4.3. For each submodel instance, the amount of private memory that must be allocated is defined in the *memory* member of the submodel data structure. Later in this section, it will be seen that each submodel instance has an associated set of base pointers to the beginning of its private storage areas.

4.2.2 Definition digraph

Simulation language users, in contrast to general purpose programming language users, use to work with more powerful environments. Features such as symbolic access to variables, interactive operation and model sorting have been provided by the simulation languages since the early days of digital computer simulation [Clancy65a]. In our opinion, new simulation languages should preserve and improve these features, even though it may increase the memory requirements and the time overhead.

One of the initial objectives to be attained with the *MUSS* structure is that isolated preprocessing of submodels as well as run time symbolic access must be supported [Huber86a]. Isolated preprocessing and run time symbolic access capabilities seem in conflict one to each other; indeed, some Fortran-based CSSL-like continuous system simulation languages solve symbolic access by means of a global common (ACSL [Mitchell81a], CSMP [James77a]), which is straightforward in monolithic models but not in hierarchical models. Other solutions based in Fortran are costly and quite cumbersome for hierarchical models.

Structured programming languages offer more powerful structures, which allow the design of neat structure to achieve the above objectives.

Several strategies have been analyzed to access all the submodel data structures. Finally, a structure has been chosen which is akin to the model structure. It will be called *definition digraph.*

Definition 4.1 *The definition digraph is a digraph whose topology is alike to the hierarchical structure of the model. Each node has the submodel data structure of its associated submodel and the edges correspond with the calls in the model structure.*

To create the definition digraph we need to define the edges between nodes. To create the edges:

- At preprocessing time a *definition routine* is generated for each submodel.
- In the environment initialization, the definition routines are executed in the model hierarchical order. When called, a definition routine creates a linked list. Upon return, each definition routine gives back the address of its data structure to the higher level calling one which in turn fills the *lower* members of the linked list with the address.

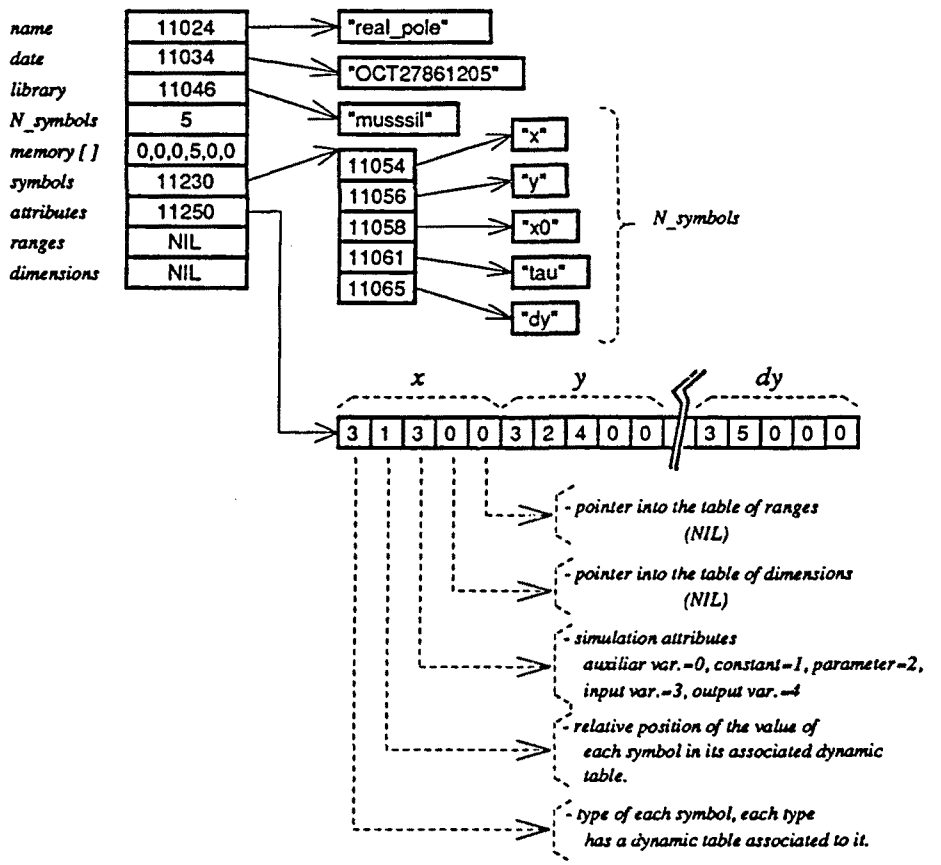

Figure 4.3: *Represents the relationship between* real_pole *submodel data structure and auxiliary static tables. In this figure,* reentrance, init_parameters, init_address *and* head *members of the data structure do not appear because they are not interesting within the subject of this subsection.*

Each element of the linked list is a variable whose structure — *LS data structure* standing for *lower_submodels* data structure— has been defined in C language as:

```
struct lower_submodels {
     double *m_state;
     int     m_derivative; /* relative position from a base address
                              given by LSODAR            */
     int     m_groot;      /* relative position from a base address
                              given by LSODAR            */
     double *m_real;
     int    *m_integer;
     int    *m_logical;
     struct submodel *lower;
     struct lower_submodels *next;
};
```

Code 4.4 *LS data structure.*

The *lower* member of the LS data structure will keep the address of a lower level submodel data structure and the *next* member, the address of the next element (LS structure) in the linked list. The other members of each LS data structure store the base addresses of the storage areas reserved for the corresponding submodel instances. These members will be explained in more detail later.

The definition routine code of the *real_pole* submodel (figure 4.2) is as follows:

```
static struct lower_submodels *low_1;
struct submodel *a0_definition()
{
        struct submodel *a1_definition();
        struct lower_submodels *alloc_lowsub();
        int address(),initpar();

        real_pole.init_address      = address;
        real_pole.init_parameters   = initpar;
        low_1           = alloc_lowsub();
        realp_pole.head = low_1;
        low_1->next     = NIL_LOW;
        low_1->lower    = a1_definition();
        return(&real_pole);
}
```

Code 4.5 *Real_pole definition routine.*

Figure 4.4 represents part of the job done by the real_pole definition routine. The number of elements in the linked list is equal to the number of lower level submodels. Remember that the real_pole submodel data structure has been defined in page 116.

The names of the definition routines — *a0_definition* and *a1_definition* — are chosen by the *MUSS* preprocessor which solves the problem of names uniqueness. The names given are cryptic because in C language only the first eight characters of variable names are significant in contrast to the *MUSS* language where the length is unlimited.

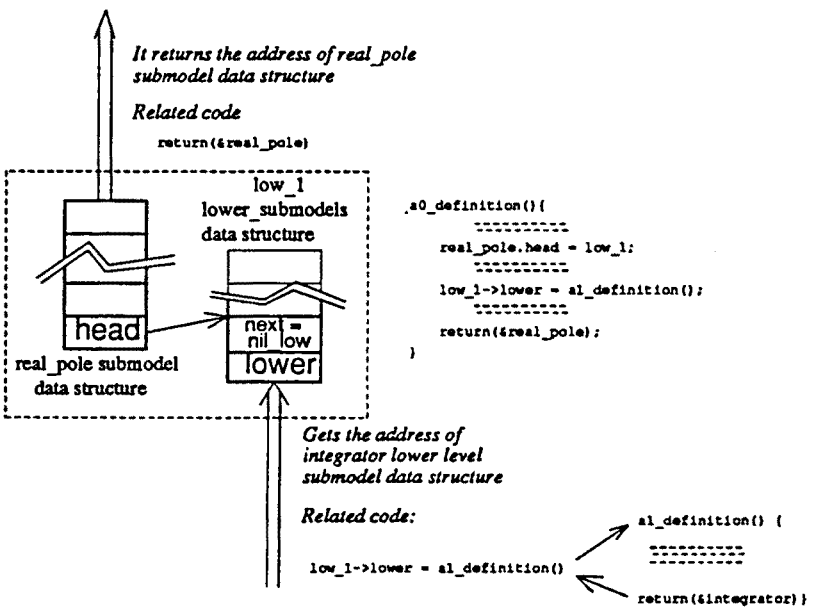

Figure 4.4: *Actions around the real_pole LS data structure achieved by the real_pole definition routine.*

Figure 4.5 shows the definition digraph of the models in figure 4.2. There is a remarkably simple recursive algorithm for visiting all the nodes of a definition digraph, in order to perform some predefined function once the target node/s has been reached.

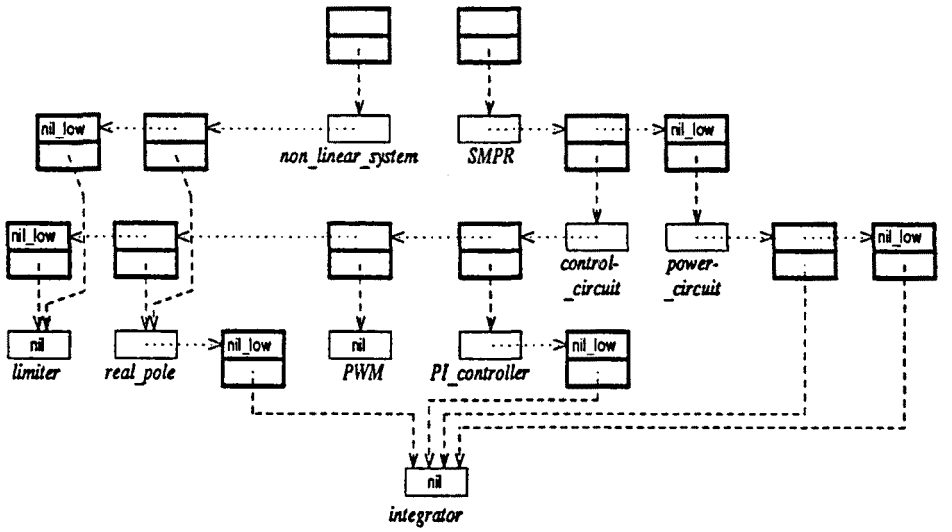

Figure 4.5: *Definition digraph of* non_linear_system *and* SMPR *models. Elements with one member represent the* head *member of submodel data structures and elements with two members represent* next *and* lower *members of LS data structures.*

```
/*================================================ visit
        Visits the definition digraph and executes
        an external defined function at each node.
*/
extern char **stack;
extern struct lower_submodels *NIL_LOW;

visit(low,function,rlevel)
int (*function)();
int rlevel;
struct lower_submodels *low;
{
        rlevel++;
        /*... stores into the stack the submodel name  */
        *(stack+rlevel-1) = low->lower->name;
        /*... call the external function               */
        if((*function)(low,rlevel)) {
                low = low->lower->head;
                while(low != NIL_LOW) {
                        visit(low,function,rlevel);
                        low = low->next;
```

```
            }
        }
        rlevel--;
}
```

Code 4.6 *Depth-first search modified algorithm to visit the nodes of the definition digraph.*

Simple additions to the skeleton in code 4.6 can be used to solve a variety of problems, like handling private memory areas of each submodel instance.

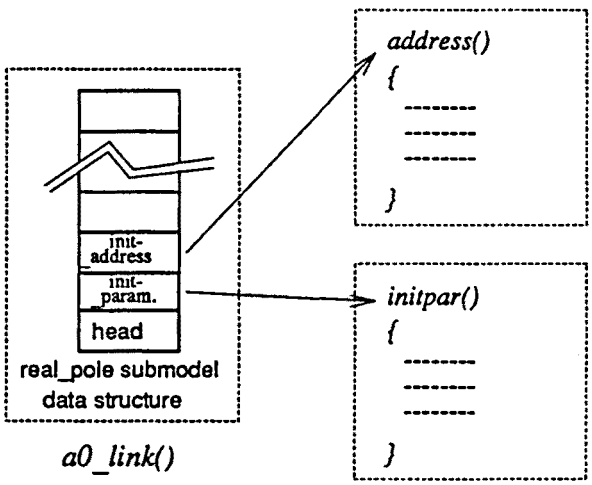

Figure 4.6: *Actions around the real_pole submodel data structure performed by the real_pole definition routine.*

Three members of the submodel data structure introduced in page 116 will now be explained in more detail: *init_address, init_parameters* and *reentrance.*

Figure 4.6 emphasizes the objective of init_address and init_parameters members, which are initialized by the submodel definition routine (see code 4.5 in page 119). The init_address member is a pointer to a submodel-specific function that initializes the address of the submodel variables. Init_parameters member is a pointer to a submodel-specific function that performs the static initialization of the submodel.

The execution of initpar() and address() functions is activated through an algorithm similar to that reported in code 4.6.

Given a LS data structure (see code 4.4 in page 119) which holds the base addresses of the private storage areas for a given submodel instance, the address() function initializes the address of each submodel variable.

The following code is the address() function —generated at preprocessing time— associated to the *control_circuit* submodel (page 197) which is represented in figure 4.2.

```
static double  *Vlic,*Tf,*V2ic,*Ti,*G,*lower_limit,*upper_limit,
               *Td,*period,*W,*error,*Vl,*transistor_on,*Vip;

static int address(low)
struct lower_submodels *low;
{
        static double *real;
        real    = low->m_real;

        Vlic    = real++;  Tf      = real++;  V2ic    = real++;
        Ti      = real++;  G       = real++;  lower_limit = real++;
        upper_limit = real++;                 Td      = real++;
        period  = real++;  W       = real++;  error   = real++;
        Vl      = real++;  transistor_on = real++;
        Vip     = real++;
}
```

Code 4.7 *Control_circuit address() routine.*

The initpar() function —also generated by the preprocessor— associated to the *control_circuit* submodel is written below,

```
static int initpar(low)
struct lower_submodels *low;
{
        *Vlic   = 0.0125;  *Tf     = 2.0e-5;  *V2ic    = 0.50;
        *Ti     = 4.5e-4;  *G      = 1.0;     *lower_limit = 0.05;
        *upper_limit = 0.95;                  *Td      = 0.0;
        *period = 1.25e-5;
}
```

Code 4.8 *Control_circuit initpar() object routine.*

It initializes the submodel parameters and constants.

Similarly to the static initialization, with a recursive algorithm, the initpar() function associated to each submodel can be accessed to perform a change in the parameters from the experiments, the studies or from the dialog level. A "reset parameters" command is provided to restore the original values of the parameters.

Concerning reentrance, each time a model is activated, the required number of submodel instances is created. This number is stored in the reentrance member by the *MCL* executive using a visiting algorithm.

4.2.3 Initialization sequence

At preprocessing time the initial segment digraph is created and thereafter used to generate the initial segment.

The objective of the initial segment is to perform the initialization of the model before the experiment run start (dynamic initialization).

To achieve the above goal the initial segment of a given submodel in the hierarchical structure:

- Has to call the lower level initial segments pertaining to the submodels explicitly invoked in the dynamic region.
- Has also to call the necessary code to initialize the input variables of the called lower level initial segments. This code in most cases is an assembly of: sentences of the dynamic region and calls to lower level state and algebraic segments.

The *control_circuit* submodel initial segment is written below. Each statement of the initial segment is a vertex in the initial segment digraph in the submodel digraph of figure C.19 in page 199.

```
c3_initial(low,re) /* initial segment */
int re;
struct lower_submodels *low;
{
        if(reentrance) {
                lev_memory(low);
                address(low);
        }
        a0_initial(low_1,V1ic);
        a0_f_s(low_1,V1);
        a6_initial(low_4,V2ic);
```

```
        a6_f_a(low_4,G,V1,Vip);
        a4_initial(low_3,lower_limit,upper_limit,Vip);
        a4_f_a(low_3,lower_limit,upper_limit,Vip,W);
        a7_initial(low_5,Td,W,period);
}
```

Code 4.9 *Control_circuit initial segment.*

Notice that besides the necessary code for the initialization of the submodels in the hierarchy, the preprocessor also generates a standard `if` statement.

Its purpose is to handle the working areas associated to the instances created at experiment activation time.

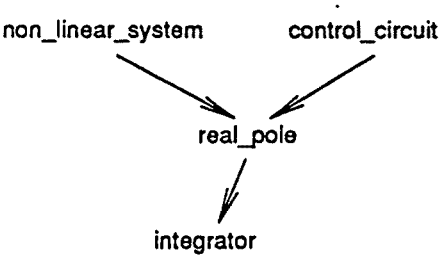

Figure 4.7: *Real_pole context if an experiment on non_linear_system and SMPR submodels has been defined.*

Figure 4.7 focuses the real_pole submodel context in the *SMPR* and *non_linear_system* hierarchical models; figure 4.8 is the definition digraph around real_pole.

Some comments of figure 4.8 are:

- Data structures included within a dot box belong to the same compiled unit.

  Notice that the number of *LS* data structures in it corresponds to the number of lower level submodel calls.

- A *LS* data structure at a given level holds the base addresses of the working areas of one of the called lower level submodels.

  This addresses are computed from the base addresses of the present submodel (instance) which in turn was sent by the calling higher level submodel (instance).

The propagation of this mechanism guarantees that all the submodel instances will work with the proper private area.

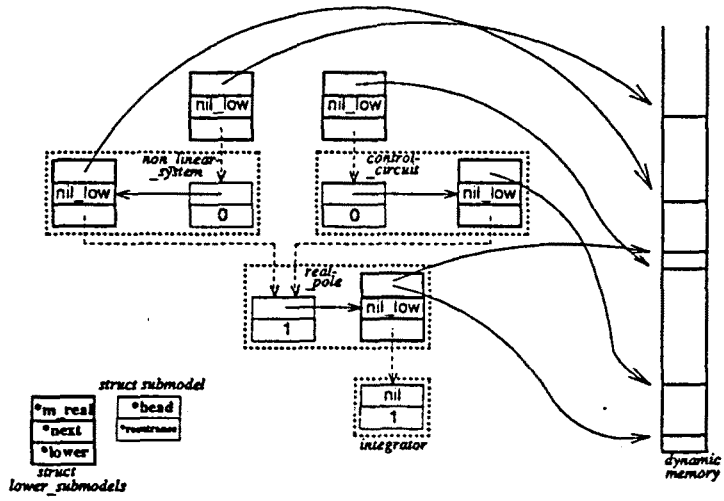

Figure 4.8: *Since the real_pole submodel is called from more than one submodel (non_linear_system and control_circuit submodels), each real_pole segment must set the base addresses of the lower level segments before calling them.*

At run time the above mechanism has to be used to compute the base addresses of the memory areas of the present instance in case of multiplicity of instances of the same submodel.

This is accomplished by the following code:

```
if(reentrance) {
        lev_memory(low);
}
```

Notice that in this example there are two instances of real_pole, hence the base addresses of the working areas of the integrator has to be computed each time real_pole submodel is called, before calling the integrator.

- The member *reentrance* is set, every time a model instance is activated, to the number of active submodel instances minus one. Thus, the *reentrance* member of the *integrator* submodel data structure has to have the value one because the integrator submodel has two active instances.

- When a submodel segment calls to a lower level submodel segment, besides the submodel input/output variables the *LS* data structure associated to the called submodel is passed.

- After the model activation, the addresses of all the model variables are set.

  When a submodel segment is executed the addresses of its variables must be preset if the number of active submodel instances is greater than one.

  The base addresses stored in the LS data structures are used to preset the addresses of the submodel variables.

  This is accomplished with the following code (the address() function has been explained before in page 123),

```
if(reentrance) {
        address(low);
}
```

  This is the case of *real_pole* and *integrator* submodel segments (see figure 4.8).

4.2.4 Dynamic sequence

When a submodel is preprocessed, besides the initial segment, the discontinuous, state, algebraic and derivative segments are created.

The calls to this segments from a higher level (submodel or experiment segment) are properly ordered by the preprocessor in basis of the segment-link digraph, in order to achieve the proper dynamic execution sequence.

State, derivative, and discontinuous segments of *control_submodel* are written below. Each statement is represented as a vertex in the corresponding submodel digraph of figure C.19 in page 199.

```
c3_f_s(low,re,traonout) /* state segment */
double *traonout;
int re;
struct lower_submodels *low;
{
        static double dic;
        if(reentrance) {
                lev_memory(low);
                address(low);
        }
```

```
        a7_f_s(low_5,transistor_on);
        *traonout = *transistor_on;
}
```

Code 4.10 *Control_circuit state segment.*

```
c3_f_d(low,re,ein) /* derivative segment */
double *ein;
int re;
struct lower_submodels *low;
{
        if(reentrance) {
                lev_memory(low);
                address(low);
        }
        *e = *ein;
        a0_f_s(low_1,V1);
        a0_f_d(low_1,Tf,error);
        a6_f_d(low_4,Ti,V1);
        a7_f_d(low_5,period)
}
```

Code 4.11 *Control_circuit derivative segment.*

```
c3_g(low,re) /* discontinuous segment */
int re;
struct lower_submodels *low;
{
        if(reentrance) {
                lev_memory(low);
                address(low);
        }
        a0_f_s(low_1,V1);
        a6_f_a(low_4,G,V1,Vip);
        a4_g(low_3,lower_limit,upper_limit,Vip);
        a4_f_a(low_3,lower_limit,upper_limit,Vip,W);
        a7_g(low_5,W,period);
}
```

Code 4.12 *Control_circuit discontinuous segment.*

4.3 Experiment and study structures

In Figure 4.9 the structure of the experiment block stated in section 2.2.3 in page 18 is reminded.

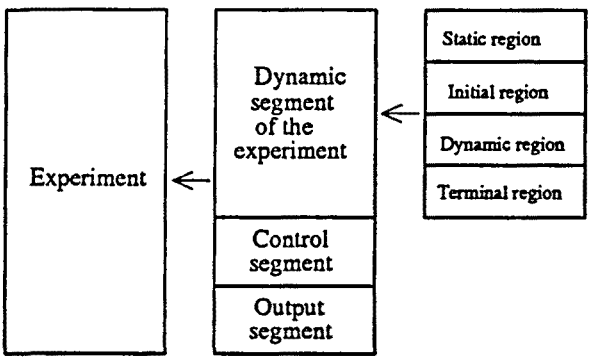

Figure 4.9: *Experiment block.*

The regions in the dynamic segment of the experiment are like those in the submodel dynamic segment block, only a terminal region is added. Therefore, the submodel digraph concept applies to analyze the experiment code.

The preprocessing of the dynamic segment of the experiment can be summarized as follows:

- Code included in the initial region of the experiment is assembled into the initial segment.
- Code included in the dynamic region is grouped in two segments: the ODE segment and the discontinuous segment. But, this time, the ODE segment has not to be splitted into the state, algebraic and derivative segments because "dynamic" code of the experiments are not called from a higher level block.
- Code included in the terminal region is grouped into the terminal segment.
- The initpar() function is created by the preprocessor (see page 123). This function will handle the static initialization of the experiment.

The control segment and the output segment of the experiment are not transformed by the preprocessor which stores it in the model data base.

The environment generator creates an interfacing function (see code 4.13) to handle the entry points to the selected experiments within the simulation environment. This function initializes an one dimensional table where each element is an *exp_table* data structure storing the information concerning the experiment.

```
#define N_EXPERIMENTS 3
struct experiment *exp_table[N_EXPERIMENTS];

/* Binds the muss environment with user defined experiments */
exp_interface()
{
        struct submodels
                *e1_definition(),   /* bouncing_ball         */
                *e2_definition(),   /* non_linear_experiment */
                *e3_definition();   /* real_pole_experiment  */
        int     e1_initial(),e1_ode(),e1_g(),
                e2_initial(),e2_ode(),e2_g(),
                e3_initial(),e3_ode(),e3_g();

        struct experiment *alloc_exp();   int i;

        /*... allocates memory for each experiment data structure */
        for(i=0;i<N_EXPERIMENTS;i++) {exp_table[i] = alloc_exp();}

        exp_table[0]->definition_function = e0_definition;
        exp_table[0]->initial_segment     = e0_initial;
        exp_table[0]->ode_segment         = e0_ode;
        exp_table[0]->disc_segment        = e0_g;

        exp_table[1]->definition_function = e1_definition;
        exp_table[1]->initial_segment     = e1_initial;
        exp_table[1]->ode_segment         = e1_ode;
        exp_table[1]->disc_segment        = e1_g;

        exp_table[2]->definition_function = e2_definition;
        exp_table[2]->initial_segment     = e2_initial;
        exp_table[2]->ode_segment         = e2_ode;
        exp_table[2]->disc_segment        = e2_g;
}
```

Code 4.13 *The interface() routine is created by the environment generator (see figure 1.2 in page 5). It binds the* MUSS *experiments with the simulation environment.*

The *exp_table* data structures store enough information to manage and execute selected experiments.

To handle the experiment instances (see page 26), the *exp_instance* data structure has been defined. Moreover, an *exp_instance* linked list allocates dynamically the experiment instances present in the simulation environment.

Experiments are not allowed to be executed in parallel. In fact, this is not a structural limitation because there is not a real necessity of executing two or more continuous experiments in parallel.

Nevertheless, notice that the *MUSS* system design ables the expansion to support combined simulation methodologies and the execution of processes (continuous or discrete) in parallel.

From the structural point of view, an experiment can be seen as the root of a submodel 'tree'. In fact, the integration and the root finder algorithms call the ODE segment and the discontinuous segment of the experiment to evaluate the derivative and discontinuous function values; this segments, in turn, call its homologous in the involved submodels to get the appropriate values.

Study blocks are placed in a higher level in the hierarchy of the simulation environment.

Being the code in the initial, dynamic and terminal regions of the study, procedural, the submodel digraph concept can not be applied to test the code robustness. Furthermore, code is not segmented.

The data structures involved in the management of studies are equivalent to those involved in the management of experiments.

Study blocks directly call experiment blocks. However, from the structural point of view, the study object code is not directly linked to the experiment object code. The study object code only informs the integration front-end routine about which experiment instance is to be executed.

4.4 MUSS command language (MCL)

The Muss Command Language (MCL) is the language through which simulation users communicate with the *MUSS* simulation environment. *MCL* contains an extensive friendly set of commands that allows users to do tasks such us:

- Get information about models, experiments and studies present in the simulation environment.
- Execute selected studies and/or experiments.
- Edit and execute *MCL* command files.
- Get run time statistics from instrumental variables in the system.

Appendix B describes the complete set of commands accepted by the *MCL* language and the *MCL* grammar specification as it is implemented.

In this section, some of the most representative commands are explained giving special emphasis to the data structures used when executing the command:

- *show:* displays information about the user defined simulation environment.
- *create:* creates active versions (instances) of studies and experiments present in the environment.
- *set block:* sets the default block (study, experiment or submodel block). The default block can be directly accessed.
- *type variable:* displays the values of study, experiment or submodel variables, parameters or constants.
- *do:* invokes for execution an active version of a study or an experiment.
- *remove:* delete an active version of a study or an experiment.

Lets suppose that the defined simulation environment is composed by the set of experiments and submodels shown in figure 4.10.

When entering into the simulation environment, the prompt

muss in >

indicates to the user that the system is ready for answering user's commands.

Now, he has the possibility to display the experiments present in the environment with the command,

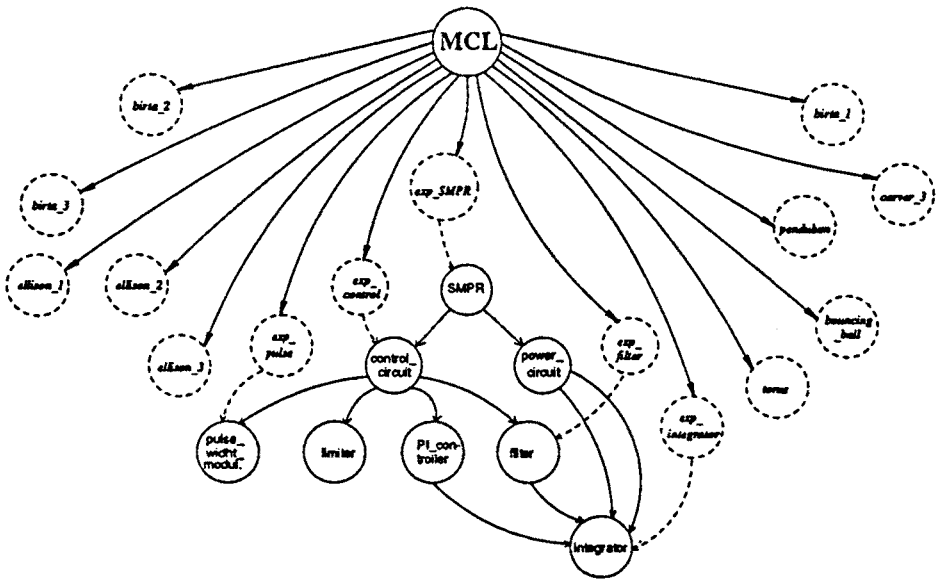

Figure 4.10: *A simulation environment.*

muss in > show experiment :*

Once the command has been issued, the *MCL* executive accesses the exp_table to find the symbolic names of the experiment to elaborate the answer:

```
:exp_integrator
:exp_filter
:exp_pulse
:exp_control
:exp_SMPR
:bouncing_ball
:torus
:simple_switch
:pendulum
:ellison_1
:ellison_2
:ellison_3
:carver_3
:birta_1
:birta_2
:birta_3
```

Each element of the exp_table is associated to an experiment. For a given experiment, for example the one associated to the third element of the table, its name can be found in the address pointed by (see figure 4.11),

```
exp_table[3]->low->lower->name
```

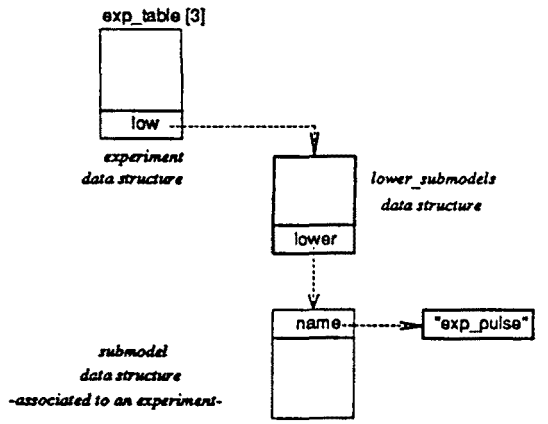

Figure 4.11: *MCL command: show experiment :**

Alternatively, the command

muss in > show model/submodel [*]

shows the models included in the simulation environment and the submodels associated to each model.

The answer to the command is:

```
[SMPR]
        [.control_circuit]
        [.control_circuit.filter]
        [.control_circuit.filter.integrator]
        [.control_circuit.limiter]
        [.control_circuit.PI_controller]
        [.control_circuit.PI_controller.integrator]
        [.control_circuit.pulse_width_modulator]
        [.power_circuit]
        [.power_circuit.integrator]
        [.power_circuit.integrator]
```

Since a model is relative concept which depends on the experiment being performed, a submodel and its lower level submodels will be considered a model if there is an experiment over the submodel.

The qualifier `\experiment` is used to get the experiments which call to each submodel.

By means of the definition digraph it is straightforward to find the models and associated submodels present in the environment as well as the experiments linked to each submodel.

The following command is used to create an instance of the *exp_SMPR* experiment,

muss in > create :exp_SMPR(instance1)

The environment must allocate data space for the experiment block and for the submodel blocks involved in the experiment. The total amount of data needed can easily be known traversing the definition digraph because each node stores the memory needed for its associated block.

The allocated memory area is thereafter distributed to the involved block ('instances').

It is also possible to set a default block with the command,

muss in > set block :exp_SMPR(instance1)[SMPR.control_circuit]

In this case, the command,

muss in > type variable/parameter *

displays the value of the parameters of the default block. Clearly, the block might have been completely specified or defined in relation with the default block.

```
:exp_SMPR(instance1)[SMPR.control_circuit]

        real    upper_limit = 9.500000e-01
        real    lower_limit = 5.000000e-02
        real    V1ic        = 1.250000e-02
        real    V2ic        = 5.000000e-01
```

```
real      Tf          = 2.000000e-05
real      Ti          = 4.500000e-04
real      Td          = 0.000000e+00
real      G           = 1.000000e+00
real      period      = 1.250000e-05
```

The next command activates the experiment instance *:exp_SMPR(instance1)* for execution,

muss in > do :exp_SMPR(instance1)

The mechanisms involved are:

- Search in the *exp_instance* linked list the data structure associated to the present experiment instance.
- Get the base addresses of the memory areas allocated for the instance.
- Pass these addresses as well as the pointers to the initial segment, ODE segment and discontinuous segment to the front-end ODE solver routine.
- Execute the experiment.

The command,

muss in > remove :exp_SMPR(instance1)

deallocates the memory areas related to the experiment instance and removes the associated *exp_instance* data structure from the linked list.

4.5 Summary

The simulation environment has been presented in this chapter. The description of it has been restricted to those aspects related to the *MUSS* language architecture: the model, experiment and study blocks plus the *MUSS Command Language*.

The simulation environment has been designed using Software Engineering techniques. The followed methodological approach can be subdivided in three parts which relay on the separation between data and code:

- Consider the simulation environment and record the understanding of it by defining structures for the data to be processed.
- Specify a program structure based on the data structures.
- Define the tasks to be performed in terms of the elementary operations available.

One important contribution has been the design of a digraph (*definition digraph*) whose structure is similar to the hierarchical structure of the models. Furthermore, each node is a data structure that stores the knowledge about the associated submodel.