AN ACSL INTERFACE FOR DYMOLA

by

Sunil Idnani

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 1

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Francois Cellier for his guidance and immeasurable support. I am especially grateful to him for his encouragement and constructive review of my work.

I would like to thank Dr. Fredrick Hill and Dr. Hal Tharp for taking the time to participate on my thesis committee.

In addition, I wish to give special thanks to Qingsu Wang for her help in offering valuable suggestions and general support.

Finally, I wish to express my deep appreciation to my wife for her constant encouragement, endurance and moral support to help me reach my goal.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

ABSTRACT

This MS Thesis proposes the use of DYMOLA, an object-oriented language for modeling hierarchically structured systems, to generate ACSL simulation programs for continuous system analysis. An ACSL model described in terms of time dependent non-linear differential equations or transfer functions can be generated from a hierarchical model description of the system using DYMOLA. The model description in DYMOLA can be an equation description or a non-linear hierarchical bond graph abstraction to describe the system under investigation.

The interface provides an automated method to generate ACSL simulation programs, hence eliminating the need for manual coding. The provision to specify an experiment description for run-time analysis and additional model statements is implemented. The implementation of the compiler's code generator includes parsing, error checking and system dependent file handling routines.

Implementation techniques, model and control file specifications, and validation with examples in several application areas are described.

CHAPTER 1

INTRODUCTION: Simulation Environment and Motivation

There has been significant development in systems modeling and computer simulation over the past several years. More sophisticated and powerful techniques are now available in continuous and discrete system simulation. Simulation languages play a major role in systems design, modeling analysis and development. They provide the basis for a good scientific methodology from specification to implementation and maintenance of systems. A system consisting of hardware and software components with a set of interfaces to manipulate information creates an environment. The simulation environment includes system representation, modeling effort and behavior analysis. The general goal of the environment is to enhance the development productivity and quality of any resulting system.

1.1 System and Model Specification

A system is generally classified by its definite and measurable attributes such that functional relations exist between them. A collection of entities meet a set of objectives through a process of interaction over time to form a system (Kheir, 1988).

A model is formed by simplification and approximation to represent the system. In particular, John Casti defines a model to be "an encapsulation of some slice of a real world within the confines of the relationships constituting a formal mathematical system." A relationship must exist between the model and the system characteristics such that model behavior can be used to predict system behavior (Bobillier, 1976). The representation of a system need not necessarily be unique.

The fundamental structure of a model can have the attributes classified as input, control and output variables. The input and control variables can arise from outside or from within the system resulting in a certain behavior of the model which can be observed and measured. The output variables are used to interpret the results of the model.

A model is used to validate some hypothesis on the functional relations between the different attributes of the system. This validation is accomplished when an experiment is performed on the system. Cellier places experimentation on a system in a simple yet elegant perspective. An experiment is a method by which system performance can be studied with the application of certain conditions on accessible input variables and analysis of accessible output variables (Cellier, 1991). The objectives are to use the model to predict future events in the system, or further properties of the system and to interpret and understand the fundamental structure of a complex physical system.

1.2 Modeling and Simulation

Modeling is the bridge between real system and a model (Zeigler, 1976). The functional description of a system can be developed and then studied by the process of modeling. Effective modeling of a complex system involves a detailed understanding of the mechanisms that control the operation of the system. The performance of the system can be optimized and changes in the system can be predicted with modeling.

The behavior of a continuous system is analyzed by describing the model in terms of differential and/or difference equations. Models that represent continuous-time systems have variables that undergo continuous change whereas those of discrete models take values in discrete time steps. State change in discrete systems takes place in discrete points in time separated by periods of inactivity. Continuous-time system examples include RLC circuits, mechanical systems involving particle motion, chemical systems and many more. Discrete-time system examples include events in a synchronous digital circuit where the rate of the clock pulse generator determines that changes occur only at discrete instants of time, models of telephone exchanges, etc.

The behavior of a model can be examined using computer simulation. The concept is well described by Granino Korn and John Wait, who state, "Simulation is experimentation with models." The practical approach to observe the behavior of a system, to predict changes and to optimize system performance is detailed simulation. Simulation studies avoid costly design errors and ensure safe designs.

Simulation is the connection to a computational device that can activate the mechanism within the model (Zeigler, 1976). Most simulation attempts are based on models that have a time dimension. This makes the models and in turn the simulation dynamic. Effective simulation study involves multiple runs combined with detailed analysis of results and validation with real data if available. Several simulation runs can be used to study model changes, effects of varying initial conditions and other parameters.

Time and state of variables are the two important coordinates in describing simulation models. The mechanisms include explicit definitions, cause-and-effect relations within the model and finally the termination conditions of simulation.

1.3 General Purpose, Simulation and Modeling Languages

General purpose high-level programming languages are used for simulation studies. In particular, FORTRAN is used extensively by numerous simulation processors and several simulation languages are FORTRAN-based. Pascal, BASIC, PL/I, SNOBOL, and Ada continue to be used, both for a wide spectrum of applications and for special purpose uses, such as simulation. C, a language that does not enforce strong typing, unlike Pascal or Ada, is extensively used for systems programming and general applications.

Simulation languages capture problem semantics efficiently and like general purpose high-level programming languages are compiler-based, but are specifically used for simulation. Continuous systems, hybrid systems in which discrete time and continuous variables coexist and discrete systems classify the different simulation languages.

Discrete system simulation programs are written in high-level programming languages like FORTRAN, Pascal, BASIC, and Ada, or in simulation languages such as SIMSCRIPT, GPSS (General-Purpose Simulation System), GASP (General Activity Simulation Program), SIMULA, SLAM (Simulation Language for Alternative Modeling), etc.

Continuous system simulation languages (CSSLs) described by Simulation Councils, Inc. (SCi) are used to implement a simulation model and an experiment. Some of the more prominent languages include ACSL (Advanced Continuous Simulation Language), DARE-P (Differential Analysis Replacement Evaluation), SIMNON and DESIRE.

Tools such as CTRL-C and MATLAB are more versatile than CSSL-type simulation languages and provide simulation as one of the several different analysis techniques (Cellier, 1991). A modeling language such as DYMOLA can be used as a front end to several different simulation languages to develop hierarchically structured model descriptions. Other modeling languages include ENPORT-4, TUTSIM and CAMP.

1.4 Motivation and Other Translators in DYMOLA

DYMOLA (Elmqvist, 1978) is a general purpose hierarchical modeling software for continuous-time systems. A compiler switch determines the target simulation language for which code is generated. The simulation languages before the enhancement included DESIRE (Korn, 1989) and SIMNON (Elmqvist, 1975). The general purpose high-level programming language for which the interface exists is FORTRAN.

A better simulation engine was needed for practical and wide-spread applications. It was essential for the simulation language to model and evaluate the performance of continuous systems described by time-dependent, non-linear differential and/or difference equations. The ability of the language to run and evaluate the model on-line, the provision to handle the problem of high data volume and a flexible explicit structure were all key characteristics that were desired in the simulation language. The sophisticated simulation language ACSL was chosen for these reasons. An ACSL program can be constructed from block diagram representation, standard FORTRAN statements or a combination of both (Mitchell and Gauthier, 1986).

1.5 Application Areas and Advantages of ACSL

ACSL is a language based on FORTRAN, and designed to model and analyze the behavior of continuous systems. It can solve time- dependent non-linear differential equations or transfer functions. Typical application areas of ACSL are missile and aircraft simulation, control systems design, chemical processes, heat transfer analysis and electronics. The diversified use of ACSL in less traditional areas such as in biological, agricultural, and the like is a more recent application of computer modeling techniques.

The language assists the user to analyze dynamic responses given a mathematical description of the system in the form of a model. It provides access to integration algorithms like Gear's Stiff, Runge-Kutta-Fehlberg and Adam's Moulton (Mitchell and Gauthier, 1986). Frequency response analysis, optimization studies, matrix integration, root locus and Jacobian evaluation can also be done. ACSL has been improved and expanded in several areas. In particular, improved methods of modeling discrete events at specific intervals of time have been introduced.

Currently, ACSL is implemented for numerous operating systems on different computers. The language can run on mainframes, minicomputers as well as personal computers. A future enhancement is to develop an ACSL translator for UNIX.

1.6 Organization of Thesis

The thesis begins with the introduction of general simulation environment concepts in Chapter 1. Chapter 1 includes the definition and classification of elements in a simulation environment, the general conceptual architecture of systems modeling, and required characteristics of simulation. Motivation behind the ACSL interface implementation in DYMOLA is discussed in this chapter.

Basic requirements and detailed specifications for the development of the ACSL interface in DYMOLA are described in Chapter 2. The development concept is discussed, and the basic structure of an ACSL model definition is presented. The desired ACSL model and control file specifications are described in this chapter. The detailed specifications are critical for the detailed design phase involving algorithm development and implementation.

Chapter 3 focuses on the detailed design of the software development. Partitioning of the functions, and the program flow outline with the integration of the interface modules by PARSER in DYMOLA's code are discussed. Variable name manipulation, development of the ACSL simulation program, and parsing and error checking the control file are described. Algorithms for implementation of modules making up the program hierarchy are described with flow diagrams in this chapter.

Chapter 4 describes several examples to validate the ACSL interface implementation. The specifications are verified with results from program execution. DYMOLA programs, control files, and resulting ACSL simulation programs are included. Simulation waveforms and results using ACSL and CTRLC are presented in this chapter.

The conclusion in Chapter 5 summarizes the software development concept and the ACSL interface implementation in DYMOLA. It concludes by discussing possible refinements and the remaining future work.

Appendix A provides related background information on DYMOLA, the general purpose hierarchical modeling software. DYMOLA as a modeling language for continuous systems is discussed with its macro handling and program generation capabilities. Language elements and DYMOLA commands are presented in this appendix.

Appendix B includes DYMOLA example programs, command sequences, resulting sets of model equations, ACSL simulation programs and waveform plots.

CHAPTER 2

DEVELOPMENT SPECIFICATIONS

A completely new language can be defined by the development of a compiler program that reads this new language. Higher level symbolic constructs such as data and control structures are made possible. The architectural design of this language may be comprised of an interpreter, a program written in a common high-level language that can decode and execute another syntax (or language). Interaction with other languages becomes crucial, and the language becomes more powerful.

2.1 Compiler vs. Interface

A compiler is a program designed to read, translate and identify errors in a source program. The compiler translates the source program in one language into an equivalent program in a target language (Aho, Sethi and Ullman, 1988). An interface is simply the translation program built into the compiler. The interface provides for error checking with the translation capability of the compiler. The extent of this error checking is determined by the parsing algorithm implemented in the interface.

2.2 Development Concept

Software development of the ACSL interface proceeds with preliminary design, detailed design, and implementation and validation. Software development begins by defining the user requirements that must be met by the software as a subsystem. Implied requirements for the software which are necessary to meet the the user reqirements are also derived. This initial design phase results in a preliminary program design. This work is performed prior to the detailed analysis that is required to fully design and implement the software system, which is an interface for a modeling language.

2.3 ACSL Model Definition

The first element that is taken into consideration towards the development phase is the general specification of an ACSL model definition. The ACSL simulation system consists of model definition and runtime analysis commands. Mathematical specification describing the dynamics of a continuous system are contained in the ACSL model definition. The general structure of an ACSL model definition is shown in Figure 2.1 (Mitchell and Gauthier, 1986).

```
PROGRAM
    INITIAL

        (Model Initialization prior to each simulation run)

    END
    DYNAMIC
        DERIVATIVE

            (Dynamic Model, differential equations in integral form)

        END
        DISCRETE

            (Statements executed in discrete time steps)

        END

        (Statements executed every communication interval)

    END
    TERMINAL

        (Statements executed when termination condition in TERMT becomes true)

    END
END
```

Figure 2.1 Outline of an ACSL Model Structure

A simulation model in ACSL may consist of only PROGRAM and END statements in which case the entire program specifies model dynamics. A more complete model definition includes INITIAL, DYNAMIC and TERMINAL sections.

The INITIAL section contains statements that are executed once prior to the simulation of the dynamic model, that is at the start of each simulation run. They are used to compute initial conditions for state variables which are outputs of integrators. Variables that do not change their values during a simulation run are placed in the INITIAL section. It is not required to place CONSTANT statements in the INITIAL section since they are not executable. The DYNAMIC section contains statements that are executed every communication interval for the duration of the simulation run. Any number of DERIVATIVE and DISCRETE sections can be nested within the DYNAMIC section. The DERIVATIVE section contains differential equations which describe the dynamics of the continuous system. This section of the simulation program is performed at the request of the integration routine to evaluate the state variable derivatives with respect to an independent variable, usually time. It is not required for the statements in this section to be ordered since they are sorted into the correct sequence by a sorting algorithm. Multiple DERIVATIVE sections are allowed with a different integration algorithm and step size for each. The ACSL integral operator is used to specify the differential equations in integral form. The DISCRETE section contains statements that describe the behavior of the continuous system at discrete events. The statements are executed at a discrete event or time point. Variables take values in discrete time steps and any state changes take place in discrete points in time separated by periods of inactivity. The TERMINAL section contains statements that are executed when the termination condition specified in a TERMT statement becomes true indicating the end of a simulation run. This section can be used to prevent repetitive calculations of variables every integration step or communication interval during a simulation run.

The model equations are converted into FORTRAN statements by the ACSL translator. This translation generates several intermediate variables and code segments in FORTRAN for efficient execution. The runtime commands are used to exercise the model with one or more simulation runs. Output variables can be listed and plotted at each communication interval. Initial conditions can be changed and new simulation runs can be performed iteratively for performance evaluation and optimization of the model characteristics. Thus, relatively complex phenomena can be represented with ACSL models, and a series of experiments at runtime can be used to analyze the behavior of the physical system. The runtime analysis commands are not discussed in further detail since they do not affect the development cycle of the ACSL interface.

2.4 Requirement Definitions

The basic requirement definitions lead to the formation of a basic algorithm for the interface development. This will permit the later detailed development of implementation and validation to work within structured bounds and constraints, thereby resulting in a realistic and more robust interface.

A systematically executed functional analysis must be performed for the desired ACSL model to satisfy the minimum requirements of the user. The most obvious criterion is the syntactical correctness of the model for an error free compilation in an ACSL environment. Several considerations such as variable name manipulation, constraining to limits on maximum number of characters per line, conformance to correct syntax for continuation, comments, etc. must be taken into account. The coherence of the different elements in an ACSL simulation program will contribute towards the realization of an algorithm for an initial program design. The solved system of equations generated by DYMOLA from the set of equations that define the behavior of a continuous system in a DYMOLA program need to establish the dynamics of an ACSL simulation program, and thus constitute the dynamic model. The set of differential equations must be expressed in integral form with ACSL's INTEG operator. Initial conditions specified for state variables in a DYMOLA program must be made accessible to the integration routine in ACSL for solving the model equations. Variables with initial values and constants in a DYMOLA program must be declared in an ACSL simulation program as part of model initialization. Detailed specifications for the desired ACSL model must be defined for implementation.

The imperative need to implement a control file becomes apparent with the different features that can possibly be accomodated in ACSL simulation program. The control file must support the minimum specifications for simulation comprised of an experiment description and/or additional constructs. The experiment description used to exercise the model consists of constraints on simulation, and possible declarations of input variables specifying types and values. The explicit constraints on simulation in the control file limit the duration of a simulation run and identify the data recording or communication interval in ACSL. The implicit condition that arises from these constraints is the maximum possible number of data points that can be used for output listing and plotting at the end of a simulation run. The calculation interval of the integration routine, also referred to as the integration step size, is also implicitly defined, where its value depends on the communication interval and number of integration steps in a communication interval.

The additional constructs of the control file must support the user in defining termination conditions for simulation, model equations that have not already been defined in the DYMOLA program, additional statements for model initialization, and multiple occurrences of necessary section(s). For instance, equations, variable assignments, and constant declarations must all be supported, and the user must be allowed to specify multiple instances of the DISCRETE section.

The provision for maximum flexibility in specification of sections and statements must be supported through implementation of the control file as part of the ACSL interface. The experiment description portion of the control file must be parsed for inclusion of relevant information into the ACSL program. The control file must also be parsed to identify the different sections that form the additional constructs. The statements within these sections must simply be copied into appropriate sections of the ACSL program, thus leaving the syntactical, and obviously logical, correctness of the statements up to the user. Some basic error checking determined by the parsing algorithm in the implementation of the control file must be performed. For instance, atleast one occurrence of a TERMT statement specifying the termination condition for simulation must be checked. Detailed specifications for the control file must be defined to facilitate the user in writing it, as well as for code implementation. The parsing and error checking will be done in accordance with these specifications.

2.5 Detailed Specifications

The general specification and outline of an ACSL model definition serve as the starting elements to help define the basic requirements for the ACSL interface development. The essential interface requirements are derived from the basic requirement definitions. The technical objectives are to completely specify the interface requirements and establish the algorithm for the interface implementation in accordance with these requirements. The detailed specifications for the desired ACSL model and control file provide a baseline for the detailed design phase.

2.5.1 ACSL Model Specifications

The target ACSL model must conform to all syntactical rules of ACSL for a successful compilation. Variable names must be less than or equal to six characters, and must start with a letter, followed by zero to five letters or digits. In several cases, the variables formed in DYMOLA exceed the maximum limit of six characters, and hence an algorithm to compress such variable names must be developed. The new variables thus formed may not be unique, and so a method of making non-unique names unique must be determined. In an ACSL program, the first seventy-two characters on a line are used for program information, whereas seventy-three to eighty are for identification purposes only. It is best to remain within this constraint not only for the model equations generated from a DYMOLA program where the restriction has already been implemented, but also for other sections, such as declarations with CONSTANT statements in the ACSL model's initialization section. These declarations consist of variables with initial values and constants that result from a DYMOLA program, and may also consist of variables with specified values and statements from the control file. A comma must be used as the delimiter between declarations. An ACSL statement must be continued onto the next line with an ellipsis, three consecutive periods, to remain within the limit of maximum characters per line. More than one statement should be placed on one line by separating the statements with a dollar sign ($). Comments in double quotes and indentation of ACSL statements must be used to meet clarity and good readability standards.

The desired ACSL model must begin with a PROGRAM statement followed by a model name from the DYMOLA program, and have a corresponding END statement as the last line. The model initialization section identified by the keyword INITIAL and a corresponding END statement are included. This block contains declarations from a DYMOLA program and/or control file. A DYNAMIC section with one DERIVATIVE section nested within, and each with its own END statement, is required. The differential equations from a DYMOLA program must be represented in integral form using ACSL's INTEG operator inside the DERIVATIVE section. The INTEG operator must be able to handle initial conditions for state variables that have been specified using submodel statements, or otherwise, in a DYMOLA program. A submodel reference must be made to identify the model element used to derive one or more equations in the form of a comment before the generated equation(s). The DERIVATIVE section must be implemented not only to include model equations from a DYMOLA program, but also equations from an equivalent section and those derived from an input statement in the control file. In other words, the ability to augment the DERIVATIVE section with the control file must be provided. The DYNAMIC section must be implemented to include multiple DISCRETE sections, following the existing DERIVATIVE section, from the control file. Identity of each DISCRETE section must be maintained in the ACSL model with a distinct name specified in the control file. This section must also be designed to include termination conditions in one or more TERMT statements if they appear outside of all sections in the control file. It must be ensured with the implementation of the control file that atleast one TERMT statement is included in the ACSL model, and the statement may not necessarily be a part of the DYNAMIC section. The provision to augment the DYNAMIC section, following the DERIVATIVE and any DISCRETE sections and/or TERMT statement(s), with ACSL statements from an equivalent section in the control file must be facilitated. The capability to include a TERMINAL section, following the DYNAMIC section, with ACSL statements from an equivalent section in the control file must be provided.

The desired ACSL model must contain a minimum of INITIAL, DYNAMIC and DERIVATIVE sections. Additionally, it can contain a TERMINAL and possibly multiple DISCRETE sections when specified in the control file.

The most simple ACSL model that can result from the interface conforms to the specifications shown in Figure 2.2.

```
PROGRAM model_name (Model name from DYMOLA program)
    INITIAL
        (Statements from DYMOLA program and control file)
    END $ "of INITIAL"
    DYNAMIC
        DERIVATIVE
            (Statements from DYMOLA program and control file)
        END $ "of DERIVATIVE"
        (Statements from control file, TERMT statement(s) if outside
         of all sections in control file)
    END $ "of DYNAMIC"
END $ "of PROGRAM"
```

Figure 2.2 Minimum ACSL Model

The most complete ACSL model that can result from the interface conforms to the specifications shown in Figure 2.3.

```
PROGRAM model_name (Model name from DYMOLA program)
    INITIAL
        (Statements from DYMOLA program and control file)
    END $ "of INITIAL"
    DYNAMIC
        DERIVATIVE
            (Statements from DYMOLA program and control file)
        END $ "of DERIVATIVE"
        DISCRETE name1
            (Statements from control file)
        END $ "of DISCRETE name1"
        .
        .   (Multiple DISCRETE sections allowed with names and
        .    statements specified in control file)
        .
        DISCRETE nameN
            (Statements from control file)
        END $ "of DISCRETE nameN"
        (Statements from control file, TERMT statement(s) if outside
         of all sections in control file)
    END $ "of DYNAMIC"
    TERMINAL
        (Statements from control file)
    END $ "of TERMINAL"
END $ "of PROGRAM"
```

Figure 2.3 Detailed ACSL Model Specifications

2.5.2 Control File Specifications

The user specified control file consists of an experiment description portion followed by sections that can be used to augment or enhance the minimum structure of an ACSL model. All the statements specified in the control file must be reflected in one form or another within the resulting ACSL simulation program. The specifications and subsequent implementation of the control file must provide maximum flexibility for the user. The general syntax and other details of the specifications are summarized in a template at the end of this section.

The control file must begin with a *cmodel* statement, and have a matching *end* statement to identify the end of relevant information in the file. Each line in the file must remain less than or equal to eighty characters. A maximum time limit for a simulation run may be specified with variable assignment in a *maxtime* statement. This statement is optional because the user may choose to specify the maximum simulation time with a CONSTANT statement inside an initial section, which may be present later, or the condition to terminate a simulation run in a TERMT statement may not depend on it. The general syntax is *maxtime* followed by a variable assignment with an "=" sign. Upon declaration in the control file, the variable and its value must appear in the INITIAL section of the resulting ACSL simulation model or program with a CONSTANT statement. The communication interval, the interval during which the DYNAMIC section is executed and the output variables have their values recorded, may be specified with variable assignment in a *cinterval* statement. This statement is optional because the user may choose to use the default name and value of this quantity (CINT = 0.1), or may opt to declare it inside a following initial section as desired. The general syntax is *cinterval* followed by a variable assignment with an "=" sign. If this statement is present in the control file, then the variable and its value must appear in the INITIAL section of the resulting ACSL program with a CINTERVAL statement. The explicit constraints on simulation thus specified imply the maximum number of data points for output printing or plotting since:

$$\text{maximum no. of data points} = \text{MAXTIME/CINTERVAL}$$

The other quantity that is implicitly defined is the integration step size, or the calculation interval, which depends on the communication interval and the number of integration steps in a communication interval. The default value of the latter quantity (NSTP = 10) may be reassigned in another section.

$$\text{integration step size} = \text{CINTERVAL/NSTP}$$

The condition to terminate a simulation run must be specified in the control file with a TERMT statement. A minimum of one TERMT statement must be placed in a location either outside of all existing sections and before a possible *input* statement, or within one of three sections from dynamic, derivative and discrete. Multiple TERMT statements are allowed in any of these locations i.e., any number of TERMT statements can be appear outside of all existing sections and before a possible *input* statement, as well as within dynamic, derivative and discrete sections. The user must specify according to ACSL, syntactically correct TERMT statements and the logical expressions contained within. The placement of this statement in the resulting ACSL program depends upon the location of this statement in the control file. One or more TERMT statements outside of all existing sections and before a possible *input* statement must be included as part of the DYNAMIC section, following the existing DERIVATIVE section and any DISCRETE sections from the control file. Any TERMT statement(s) in dynamic, derivative, and discrete sections are placed respectively in DYNAMIC, DERIVATIVE, and DISCRETE sections of the ACSL program, along with any other statements that may be present within each section. A TERMT statement placed in the DYNAMIC section of an ACSL program terminates the simulation run at a communication interval when the logical expression in the statement becomes true. When placed in a DERIVATIVE or a DISCRETE section the statement terminates the simulation run at the integration step, or the calculation interval, following the logical expression becoming true.

The input variables in a DYMOLA program, dependent or independent, must be declared with an *input* statement in the control file. This statement is required if there are any inputs in the model, and is not to be placed if none exist. The general syntax is *input* followed by the number of inputs and then input declarations. A comma must separate the number of inputs and the first input declaration, as well as separate each input declaration in the list. An input declaration must consist of the variable name with its classification as an independent variable and a value associated with it, or as a dependent variable and an expression associated with it. An independent variable identified by the keyword *independ* must be separated from its value by a comma as the delimiter. The keyword and the value must be enclosed in parenthesis, preceeded by the variable name. The same format applies for a dependent variable with the keyword *depend*, and instead of a value an expresssion is used. A syntactically correct expression must be specified by the user. Any combination of independent and dependent variables may form the declaration list. The declarations may be continued on to the next line with a complete input declaration on the previous line. The placement of these variables in the resulting ACSL program depends upon their classification. Independent variables and their values must be included in the INITIAL section with a CONSTANT statement, whereas dependent variables and associated expressions must be included as part of the DERIVATIVE section.

The statements thus far described constitute the experiment description portion of the control file. The *maxtime*, *cinterval*, and TERMT statement(s), if present, must appear before an *input* statement if there exist any inputs in the model. In the case where there are no inputs, and hence no *input* statement, they must directly preceed any sections or blocks in the control file. Multiple TERMT statements, but only single instances of *cinterval* and *maxtime* statements are allowed. None, all, or any combination of these statements can appear in any order. The *input* statement, if present, must appear before any sections in the control file.

The sections allowed to be placed in the control file are initial, dynamic, derivative, discrete, and terminal with an *end* statement for each. Multiple discrete sections, but only single instances of the rest are allowed. None, all, or any combination of these sections can appear in any order. All sections are optional, and none can be nested within another.

Compliance with ACSL rules, and syntactical correctness of the statements and equations within each section is left up to the user. These statements are copied as is into the respective sections of the resulting ACSL program. INITIAL, DYNAMIC, and DERIVATIVE sections present in the ACSL program are simply augmented, as determined by the control file. A TERMINAL section, and one or more DISCRETE sections are created inside the ACSL program only when equivalent sections appear in the control file, and statements are copied into each section accordingly. Each discrete section in the control file can optionally be identified by a name, and is created in the ACSL program with the same name.

All these sections constitute the so called additional constructs in the requirement definitions.

The most basic control file that can be specified by the user for an ACSL interface and a model with no inputs is shown in Figure 2.4.

```
cmodel

TERMT (<logical expression>)

end
```

Figure 2.4 Minimum Control File (model with no inputs)

where < ... > = specified by user.

The most basic control file that can be specified by the user for an ACSL interface and a model with inputs is shown in Figure 2.5.

```
cmodel

TERMT (<logical expression>)
input <N>, <var1>(independ,<value1>), <var2>(depend,<expr1>),
           <var3>(independ,<value2>), <var4>(depend,<expr4>),
           .....
           <varN>(depend,<exprN>)          /* or */
           <varN>(independ,<valueN>)
end
```

Figure 2.5 Minimum Control File (model with inputs)

where < ... > = specified by user,   /*...*/ = comment,
N = number of inputs,
var1 ... varN = input variable names,
value1 ... valueN = numerical values for independent variables, and
expr1 ... exprN = expressions for dependent variables.

The detailed specifications can be summarized in a template format. The most complete control file that can be specified by the user for an ACSL interface is shown in Figure 2.6.

```
cmodel /* required */

          /* maxtime, cinterval and TERMT can appear in any order */
          /* before a possible input statement and/or sections */

maxtime <variable> = <value> /* optional, one statement only, */
                             /* placed in INITIAL with CONSTANT */

cinterval <variable> = <value> /* optional, one statement only, */
                               /* placed in INITIAL with CINTERVAL */

TERMT (<logical expression>) /* required if not in dynamic, derivative or */
                             /* discrete; multiple TERMTs allowed, */
                             /* placed in DYNAMIC following */
                             /* DERIVATIVE and any DISCRETE */

input <N>, <var1>(independ,<value1>), <var2>(depend,<expr1>),
           <var3>(independ,<value2>), <var4>(depend,<expr4>),
           .....
           <varN>(depend,<exprN>)     /* or */
           <varN>(independ,<valueN>)
        /* required if number of inputs > 0, else should not be included; */
        /* must appear before a section when present; independ variables */
        /* placed in INITIAL with CONSTANT, depend variables */
        /* placed in DERIVATIVE as equations */

        /* sections can appear in any order, end required for each section, */
        /* and ACSL statements copied as is into respective sections of */
        /* an ACSL program */

initial /* optional, one section only, valid ACSL statements */
.... /* e.g., CONSTANT, CINTERVAL, MINTERVAL, MAXTERVAL, */
.... /* NSTEPS, ALGORITHM, equations etc., placed in INITIAL */
end

dynamic /* optional, one section only, valid ACSL statements */
.... /* e.g., TERMT, equations etc., placed in DYNAMIC following */
.... /* DERIVATIVE, any DISCRETE and/or any TERMT from above */
end
```

Figure 2.6 Detailed Control File Specifications

```
derivative /∗ optional, one section only, valid ACSL statements ∗/
.... /∗ e.g., TERMT, SCHEDULE, equations etc., placed in DERIVATIVE ∗/
.... /∗ following any equations for dependent input variables, and solved ∗/
.... /∗ model equations from DYMOLA ∗/
end

discrete <name1> /∗ optional, valid ACSL statements ∗/
.... /∗ e.g., TERMT, INTERVAL, SCHEDULE, equations etc., placed ∗/
.... /∗ in new DISCRETE section, identity maintained with <name1> ∗/
end
.
.   /∗ multiple discrete sections allowed, names optional ∗/
.
discrete <nameN>
....
....
end

terminal /∗ optional, one section only, valid ACSL statements ∗/
.... /∗ equations etc., placed in new TERMINAL section ∗/
end

end /∗ required for cmodel ∗/
```

Figure 2.6 Detailed Control File Specifications (contd.)

where < ... > = specified by user and /∗...∗/ = comment.

CHAPTER 3

ALGORITHMS AND IMPLEMENTATION

The detailed design of the software development involves analysis, documentation, detailed program design with algorithms and implementation with coding. The analysis activity involves the required grouping of functions into modules, modification approach for each of the components to be adapted from the existing software with data structure analysis, software design analysis, and general program flow. The detailed specifications provide the baseline for the detailed design and algorithm development for implementation. The most important part of the detailed design phase is detailed documentation. The documentation is the design, and if it is incomplete, then the design is incomplete. The algorithms to be implemented in the development phase are describe with flow charts which provide logic flows. The implementation process is the translation of documented algorithms and descriptions and into executable code.

3.1 Modularization

Preliminary software design involves organization of functions in the requirement definitions into functional related groupings, also referred to as modules, which are the first organizational level below the ACSL interface program. For each level of organization through module-level design, the components are identified by name and function. Descriptions of the modules making up the interface program hierarchy lead to the development of algorithms for implementation purposes.

The partitioning of functions begins with the assignment of variable name manipulation to the first module called ACSLNAME. Variable names that need to be compressed and made unique for ACSL are handled in this module. The formation of the ACSL simulation program into different sections determines the functionality of the second module ACSLPRINT. This formation involves incorporation of information from a DYMOLA program to meet the minimum specifications, or from a DYMOLA program and a control file depending upon user request for an ACSL model or an ACSL program. The control file is compiled with the module ACCOMPILE, which performs parsing and error checking to meet all requirements. Error reporting upon detection of a compilation error in the control file is performed by the module ACSLERR. The integration of all these modules, and the general program flow is defined by the module PARSER in DYMOLA's code. This module is modified to incorporate the ACSL interface into DYMOLA's implementation. The differential equations describing the model are manipulated to be expressed in integral form with ACSL's INTEG operator, and initial conditions for state variables are made accessible to the integration routine with this operator. Several DYMOLA modules are modified to implement this feature.

3.2 Program Flow

The primary thrust of the top-down design approach is to define how the program will be structured to perform the functions. A first cut at sequencing the executable elements and other aspects of module-level design is necessary to accomplish this task. The PARSER module in DYMOLA's code is modified to integrate the ACSL modules, and incorporate the interface code with the rest of the implementation.

The SCAN function in DYMOLA's code is used extensively by PARSER. The function is used to parse a user command, and one or more modules are invoked by the PARSER in accordance with the request. SCAN can be used to fetch an item from the screen or a file with each invocation. Each item fetched, referred to as the *nextitem*, can be an identifier, number or delimiter. The function skips blank lines and comments, and handles the continuation symbol (—>) as well as other special symbols. Routines that perform system dependent file handling are invoked by the function, and the item and its type are returned to the calling module.

The modules making up the interface program hierarchy are integrated with PARSER. The general outline of the program flow is described by a flow diagram in Figure 3.1.

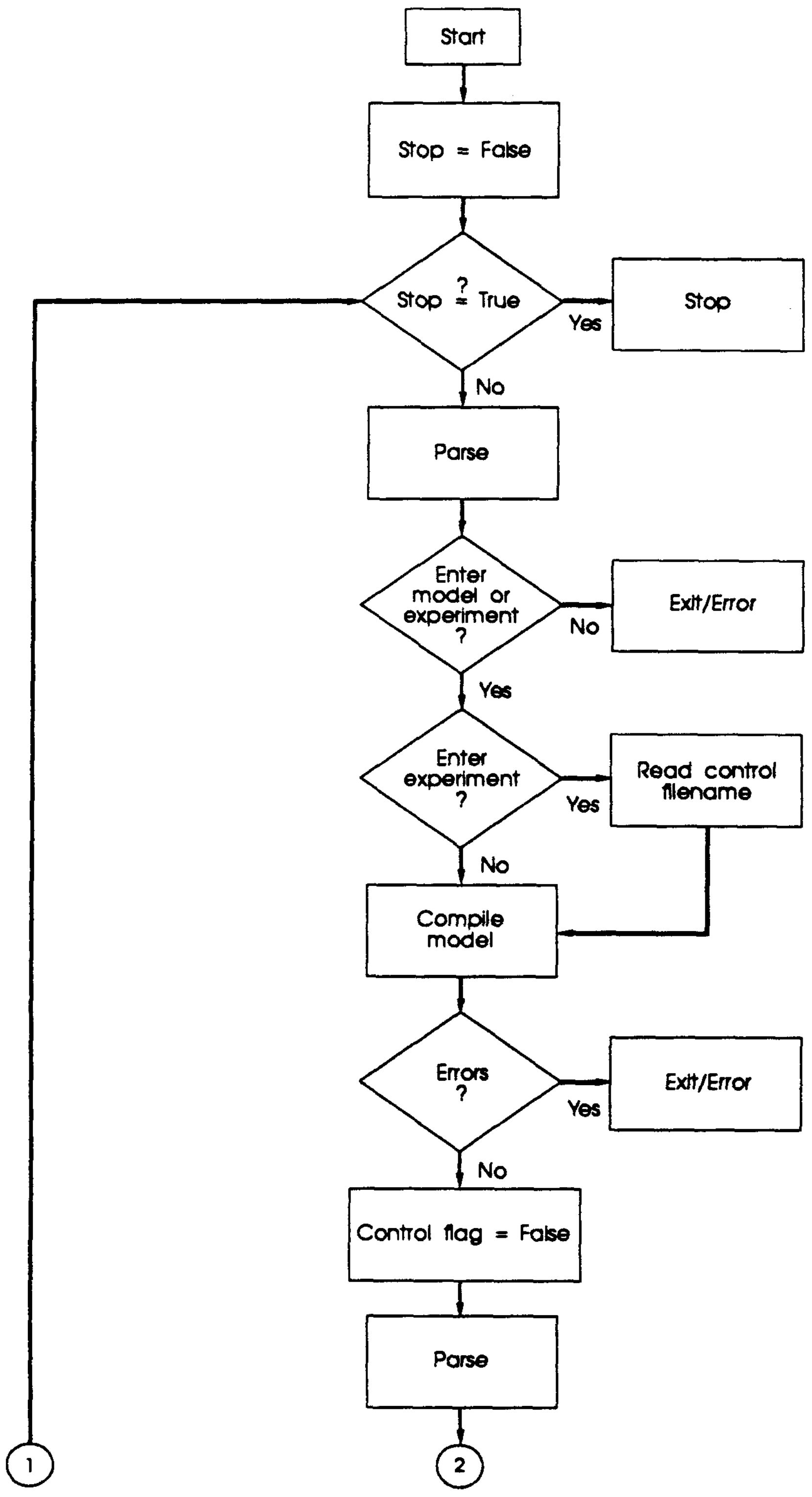

Figure 3.1 Program Flow Outline

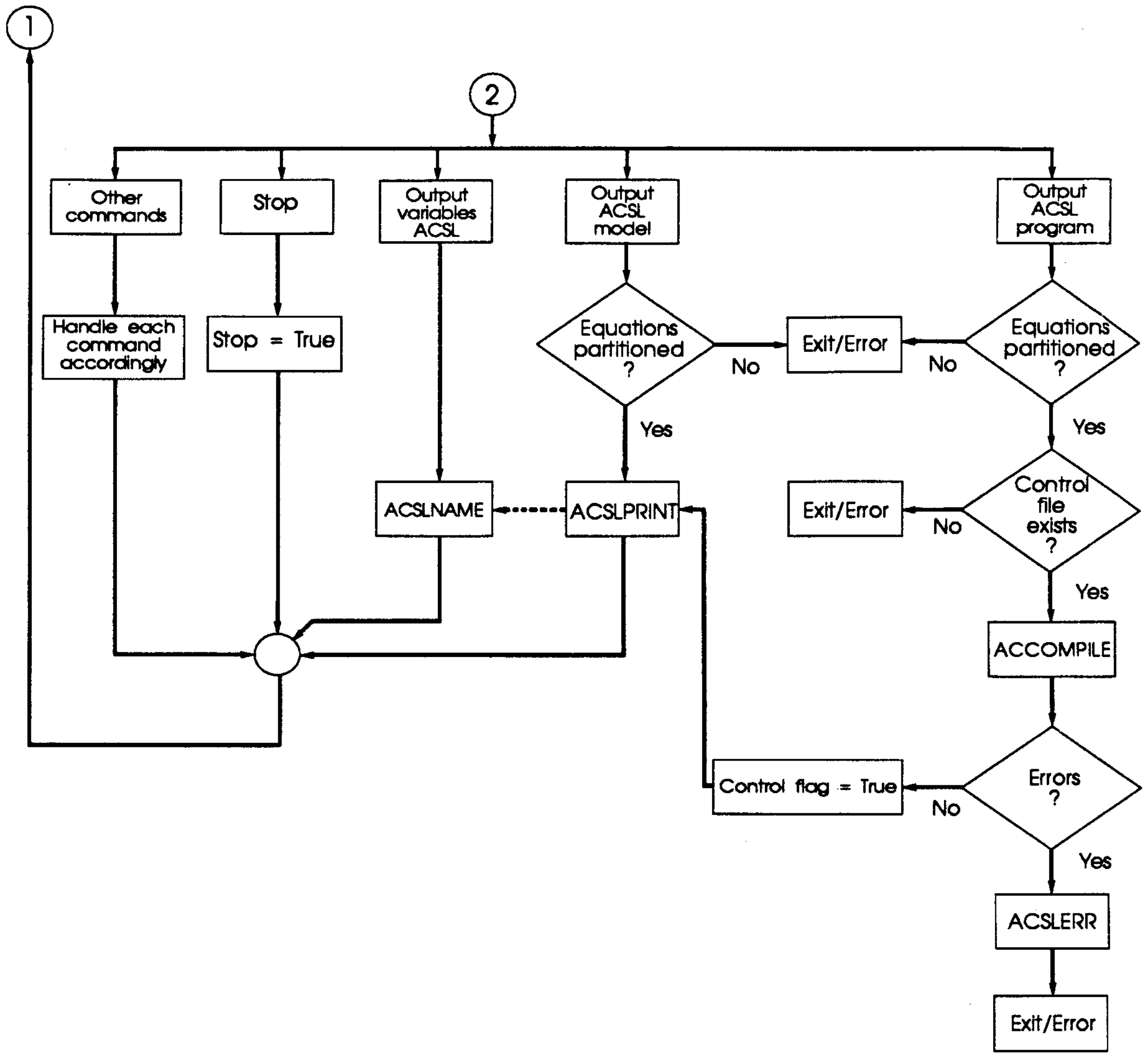

Figure 3.1 Program Flow Outline (cont.)

3.3 Variable Name Manipulation

The variable names formed by DYMOLA may, in several cases, exceed the maximum limit of six characters required by ACSL. This is a direct consequence of a concatenation process during which a variable name is appended (with delimiter ".") to the name of a respective model or a submodel it belongs to, and thus proper hierarchy is established to maintain distinction. Such variable names are compressed to the maximum limit, and then compared with all other variable names, that may or may not have, depending on their size, undergone an identical compression algorithm earlier. They are also compared with reserved names and other keywords for ACSL statements.

If a variable after compression is found to be not unique, as detected by name collision, then the algorithm to make it unique is applied. The model or submodel name is included in the variable name using the delimiter "X". An alphabet is chosen as the delimiter for names since any other special symbol is not acceptable by the ACSL compiler. First, the model or submodel name identifier is compressed to two or more characters depending upon the length of the non-unique variable name. Next, the non-unique variable name is compressed to three or less characters depending upon the length of the compressed model or submodel name. The length of the newly formed variable is restricted to six characters, with the compressed model or submodel name followed by the delimiter "X", and the compressed non-unique variable name. This new variable may still not be unique. It is then compared again with all other variable names. The other variable names may or may not have undergone the application of the same algorithm, depending on if they were unique or not after initial compression.

If the variable from the first algorithm is found to be not unique, then a second algorithm to make it unique is applied. It forces uniqueness by first compressing the model or submodel name to one or more characters depending upon the size of the *original* variable, which is the variable before any compression or algorithm has been applied. Next, the *original* variable name is compressed to three or less characters depending upon the length of the compressed model or submodel name. The length of the newly formed variable is again restricted to six characters, with the compressed model or submodel name followed by the delimiter "X", the compressed *original* variable name, and a number. This number forces uniqueness this time since it is incremented for each non-unique variable name from the first algorithm.

Finally, the derivatives of all state variables are formed by concatenating the state variable names to the character "d", and then compressing the resulting names to six characters when they exceed the maximum limit required by ACSL.

Compression during variable name manipulation takes place by shifting characters from right to left of a variable name, until the maximum limit is reached. The number of characters shifted is determined appropriately. This reduction technique is performed without losing complete originality of the variable name, especially when submodel names are included in the variable names.

The flow diagram describing the algorithm for variable name manipulation in ACSLNAME is shown in Figure 3.2.

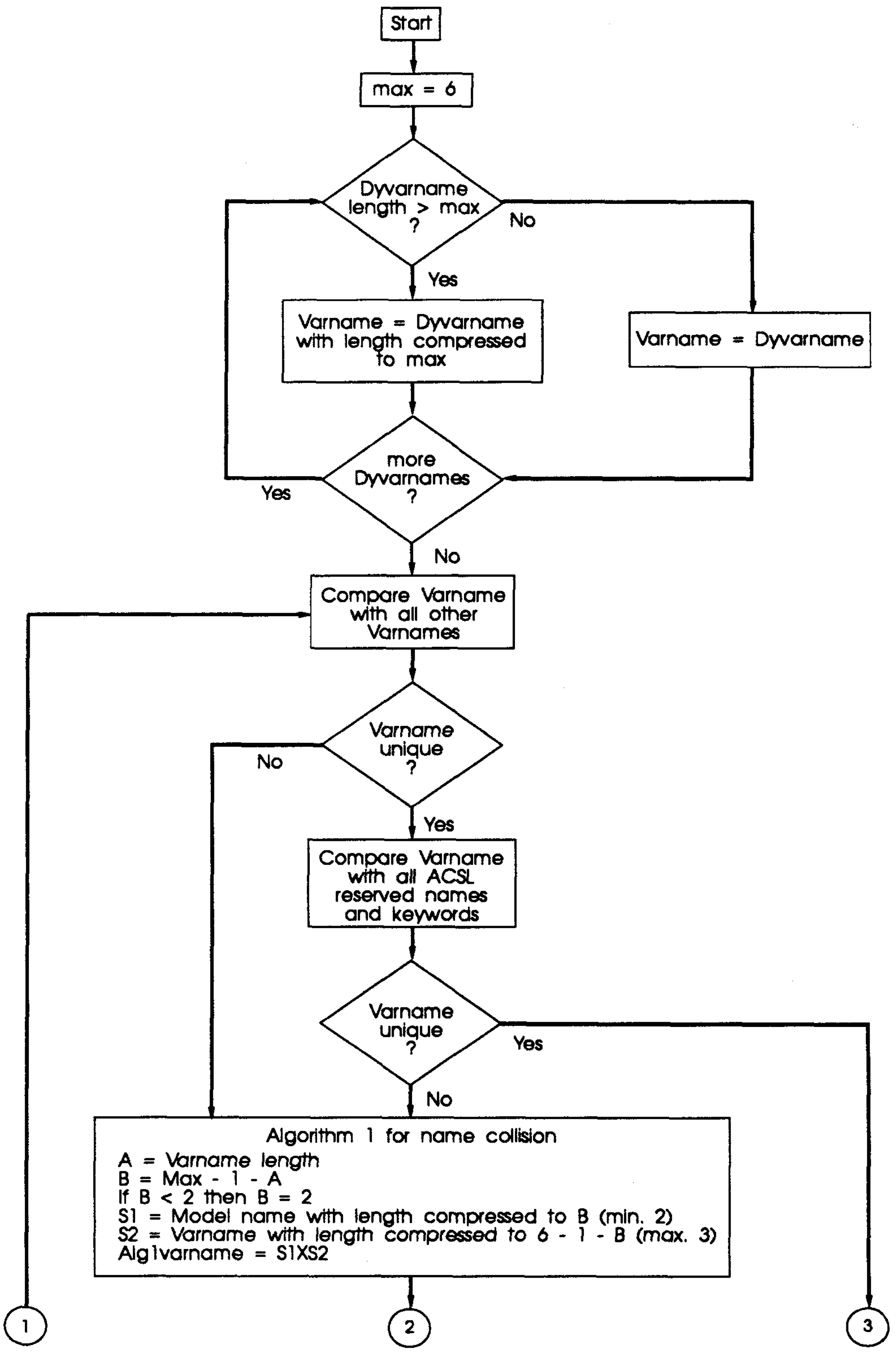
Start
max = 6
Dyvarname length > max ?
No
Yes
Varname = Dyvarname with length compressed to max
Varname = Dyvarname
more Dyvarnames ?
Yes
No
Compare Varname with all other Varnames
Varname unique ?
No
Yes
Compare Varname with all ACSL reserved names and keywords
Varname unique ?
Yes
No
Algorithm 1 for name collision
A = Varname length
B = Max - 1 - A
If B < 2 then B = 2
S1 = Model name with length compressed to B (min. 2)
S2 = Varname with length compressed to 6 - 1 - B (max. 3)
Alg1varname = S1XS2
1
2
3

Figure 3.2 Algorithm for Variable Name Manipulation

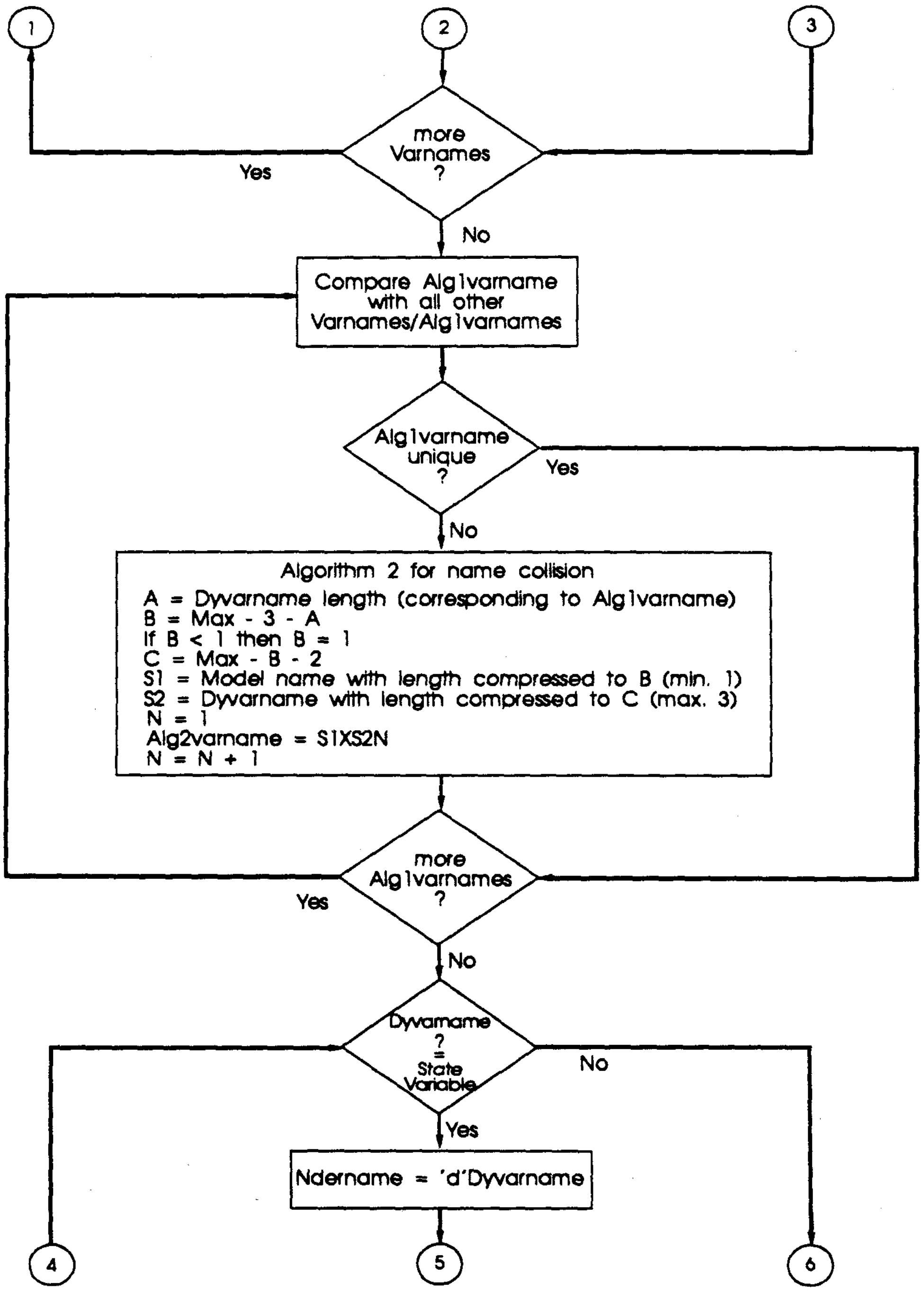
1
2
3
more Varnames ?
Yes
No
Compare Alg1varname with all other Varnames/Alg1varnames
Alg1varname unique ?
Yes
No
Algorithm 2 for name collision
A = Dyvarname length (corresponding to Alg1varname)
B = Max - 3 - A
If B < 1 then B = 1
C = Max - B - 2
S1 = Model name with length compressed to B (min. 1)
S2 = Dyvarname with length compressed to C (max. 3)
N = 1
Alg2varname = S1XS2N
N = N + 1
more Alg1varnames ?
Yes
No
Dyvarname ? = State Variable
No
Yes
Ndername = 'd'Dyvarname
4
5
6

Figure 3.2 Algorithm for Variable Name Manipulation (cont.)

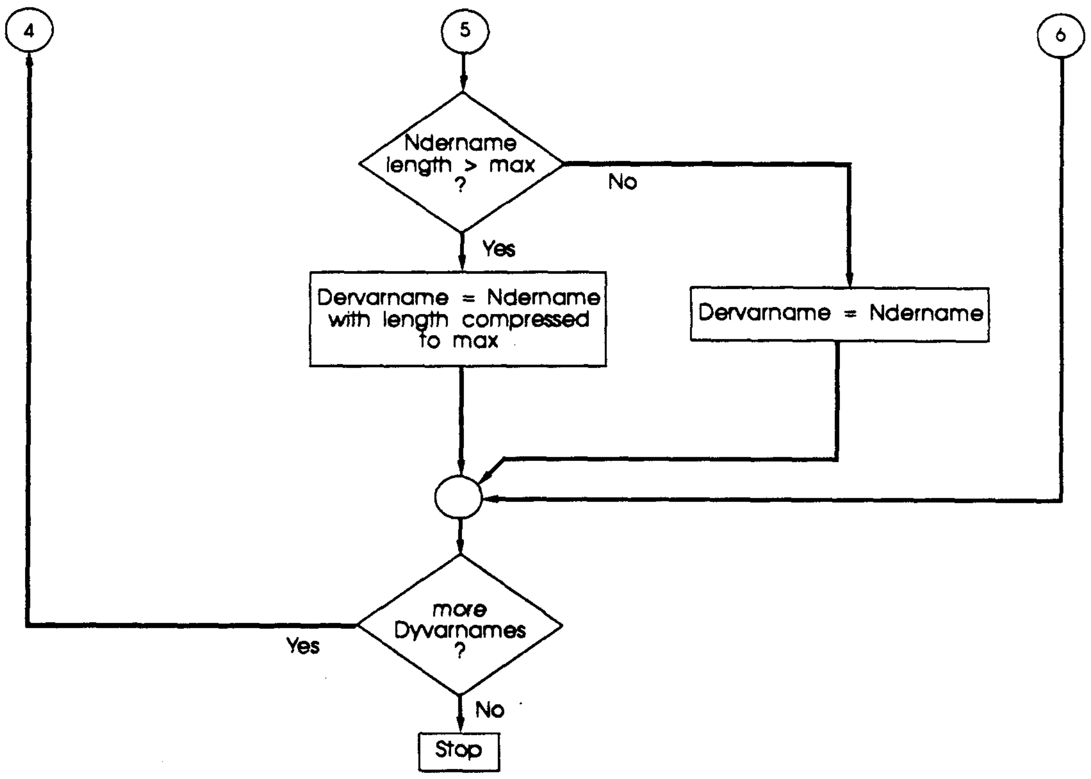

Figure 3.2 Algorithm for Variable Name Manipulation (cont.)

3.4 ACSL Simulation Program

The module ACSLPRINT is used to form the different sections of the target ACSL simulation program. It is invoked by two possible user commands in DYMOLA, *output acsl model* and *output acsl program*, provided the equations have been partitioned earlier. When the user requests for an ACSL model, the module is invoked directly, and the value of the control flag is false (Refer Figure 3.1). On the other hand, when the user requests for an ACSL program, the control file is first compiled by the module ACCOMPILE, and then if there are no errors, the control flag is set to a true value before module invocation. Based on the value of this control flag, the module incorporates appropriate information, with correct syntax, into the ACSL simulation program.

Several other existing DYMOLA modules invoked by ACSLPRINT, during the formation of the DERIVATIVE section, are modified. Every differential equation is detected from the system of solved equations in DYMOLA, and expressed in integral form with ACSL's INTEG operator. The initial condition for a state variable is passed directly to the operator. The implementation of these existing modules involves variable type detection, operand detection, several infix techniques and complex equation manipulations.

The flow diagram describing the algorithm for the basic formation of an ACSL simulation program in ACSLPRINT is shown in Figure 3.3.

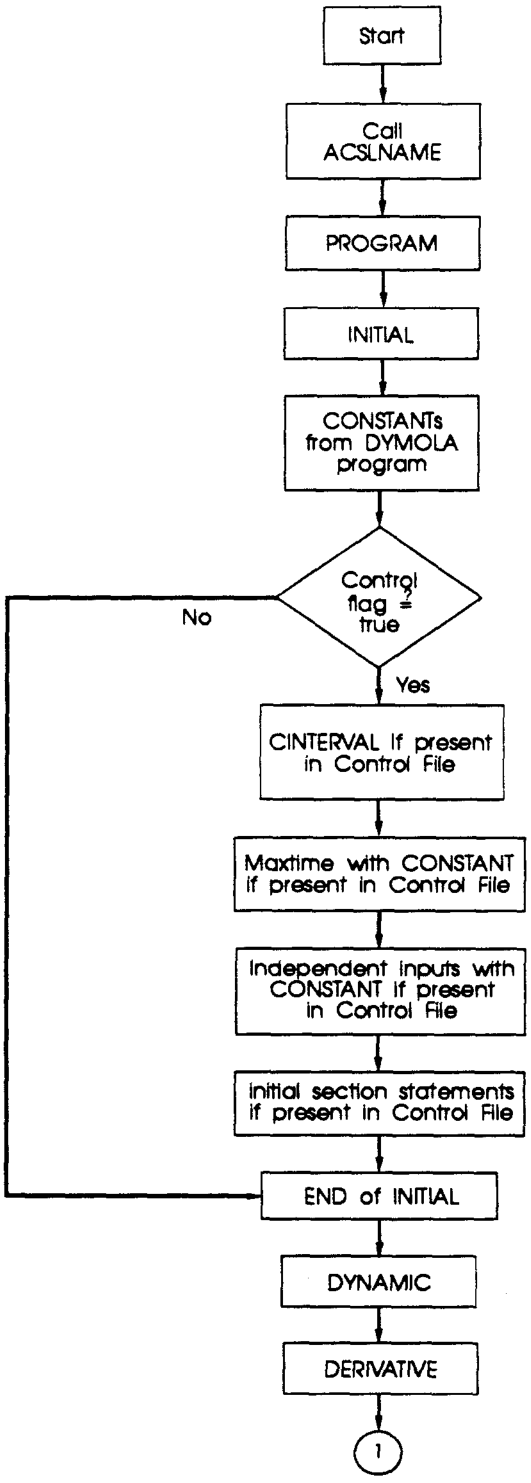
Start
Call ACSLNAME
PROGRAM
INITIAL
CONSTANTs from DYMOLA program
Control flag ≟ true
No
Yes
CINTERVAL if present in Control File
Maxtime with CONSTANT if present in Control File
Independent inputs with CONSTANT if present in Control File
Initial section statements if present in Control File
END of INITIAL
DYNAMIC
DERIVATIVE
1

Figure 3.3 Algorithm for ACSL Simulation Program

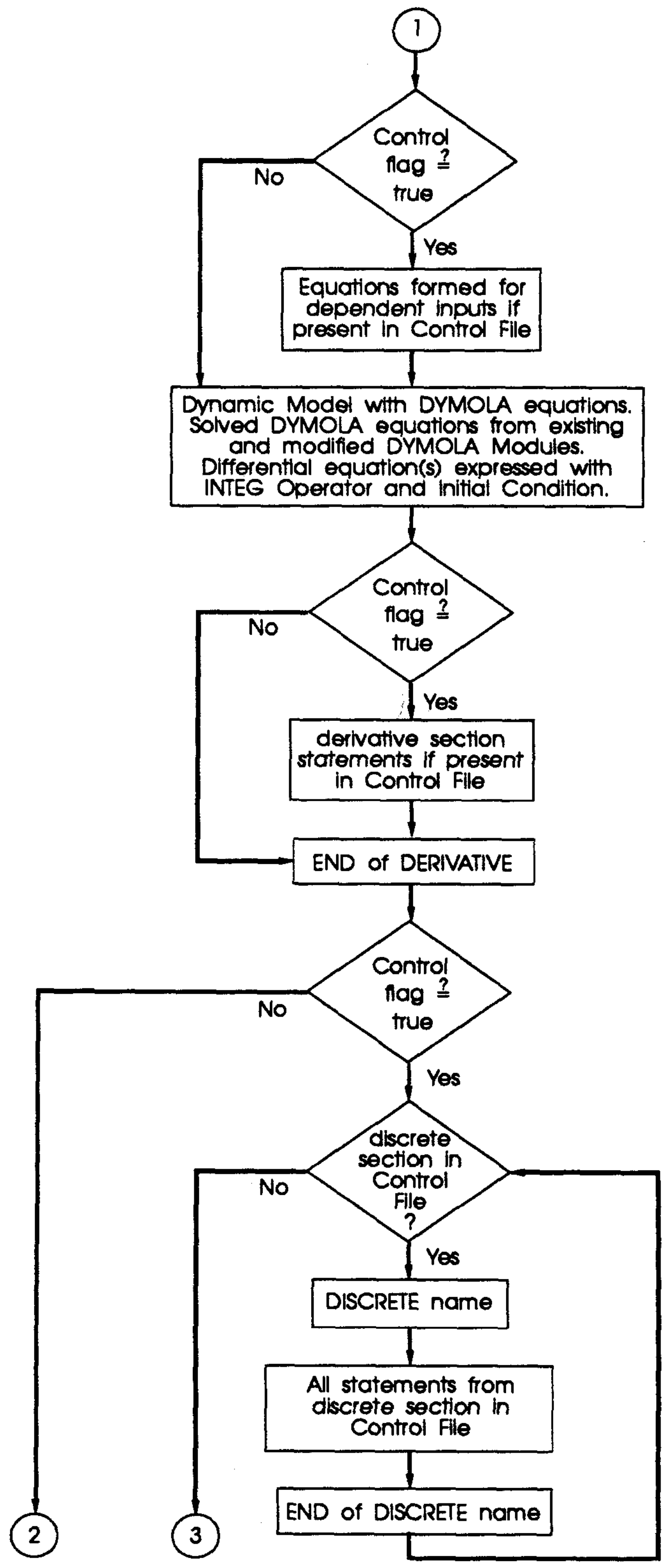
1
Control flag ≟ true
No
Yes
Equations formed for dependent inputs if present in Control File
Dynamic Model with DYMOLA equations. Solved DYMOLA equations from existing and modified DYMOLA Modules. Differential equation(s) expressed with INTEG Operator and Initial Condition.
Control flag ≟ true
No
Yes
derivative section statements if present in Control File
END of DERIVATIVE
Control flag ≟ true
No
Yes
discrete section in Control File ?
No
Yes
DISCRETE name
All statements from discrete section in Control File
END of DISCRETE name
2
3

Figure 3.3 Algorithm for ACSL Simulation Program (cont.)

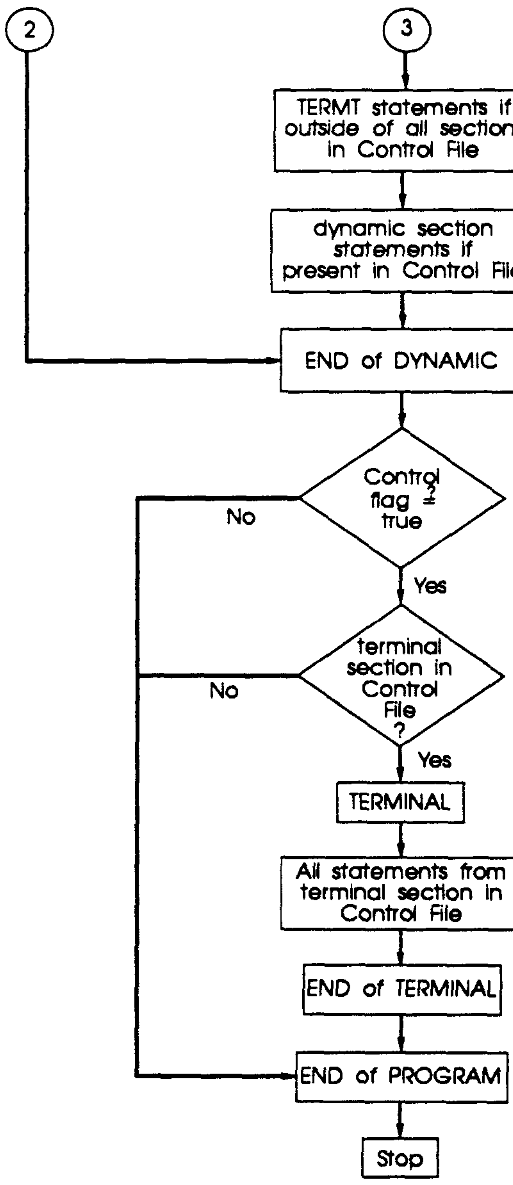
2
3
TERMT statements if outside of all sections in Control File
dynamic section statements if present in Control File
END of DYNAMIC
Control flag ≟ true
No
Yes
terminal section in Control File ?
No
Yes
TERMINAL
All statements from terminal section in Control File
END of TERMINAL
END of PROGRAM
Stop

Figure 3.3 Algorithm for ACSL Simulation Program (cont.)

3.5 Parsing and Error Checking

The different parts of an ACSL simulation program result from a DYMOLA program, and a control file. The system of solved equations from DYMOLA that form the dynamics of simulation, and the precise inclusion by printing of relevant information into an ACSL program are completely determined by program design and implementation of DYMOLA's code, and embedded interface code. Hence, error checking for results produced by these tasks is not necessary. On the other hand, the control file written by the user is prone to errors, and is parsed to detect errors before inclusion of any information into an ACSL program. Even with the most flexible structures, strict guidelines exist and the user must conform to them. The parsing algorithm is performed with system dependent file handling routines. The control file is scanned to identify statements and syntax, and error checking and reporting are performed in accordance with detailed specifications. All the parsing and error checking are done by the module ACCOMPILE. All the error reporting is done by the module ACSLERR, which contains a complete set of print statements for meaningful error messages.

The SCAN function in DYMOLA's implementation is used extensively to parse the control file. The starting line of the file and the experiment description portion is parsed to check for name identifiers such as *cmodel*, *maxtime*, *cinterval* and TERMT. All delimiters including parentheses, commas, and carriage returns are checked, as dictated by the specifications, and an error is reported if missing from a statement. A flag is set upon detection of a TERMT statement, whether it exists in this portion of the file, or within any section from dynamic, derivative or discrete. If none exists, an error message is reported. Single instances of optional *maxtime* and *cinterval* statements, and their placements with any TERMT statements *before* an *input* statement (if it exists), or *before* the first section (if it exists and the *input* statement does not) are checked. If the number of inputs specified in an *input* statement differs from the number of model inputs in a DYMOLA program, or if an *input* statement is not included for a model with inputs, or also if an *input* statement does not preceed any existing section, then an appropriate error is reported for each case.

The rest of the control file is parsed to identify the presence of optional sections, and appropriate flags are set for incorporation of statements contained within each section into an ACSL program. Single instances of all sections that are present, except discrete, are checked and an error is reported for any violation. An error also results if an *end* statement is missing for a particular section. Checks for a TERMT statement within all dynamic, derivative, and discrete sections are performed, and an error results if not found in any of these sections as well as above and outside the first section in a control file. It is checked to make sure that a section is not nested within another, that is if the name of another section appears before the *end* statement for a particular section.

The end of all relevant information is detected with an *end* statement for *cmodel*, and an error occurs if it is not present. All statements that might exist after the *end* for *cmodel* are ignored, a message stating that is printed out, and execution of the program continues. The presence of any blank lines at the end of the control file, following the *end* for *cmodel*, is taken into account during implementation. The SCAN routine in DYMOLA's code skips blank lines (a useful feature for all other cases), but as a result, it cannot be used *after* an end of file is encountered with its previous invocation (just like *readln* in Pascal). This limitation causes blank lines at the end of a control file to create a problem, especially when the function is used in a *while* loop to parse the file. It is worth noting that any blank lines elsewhere in the control file do not create such a problem. The situation is handled with the removal of all blank lines in the file before parsing. End of file is then encountered when it reads the last non-blank statement in the file, and the loop terminates as desired.

The flow diagram describing the algorithm for parsing and error checking the control file in ACCOMPILE is shown in Figure 3.4.

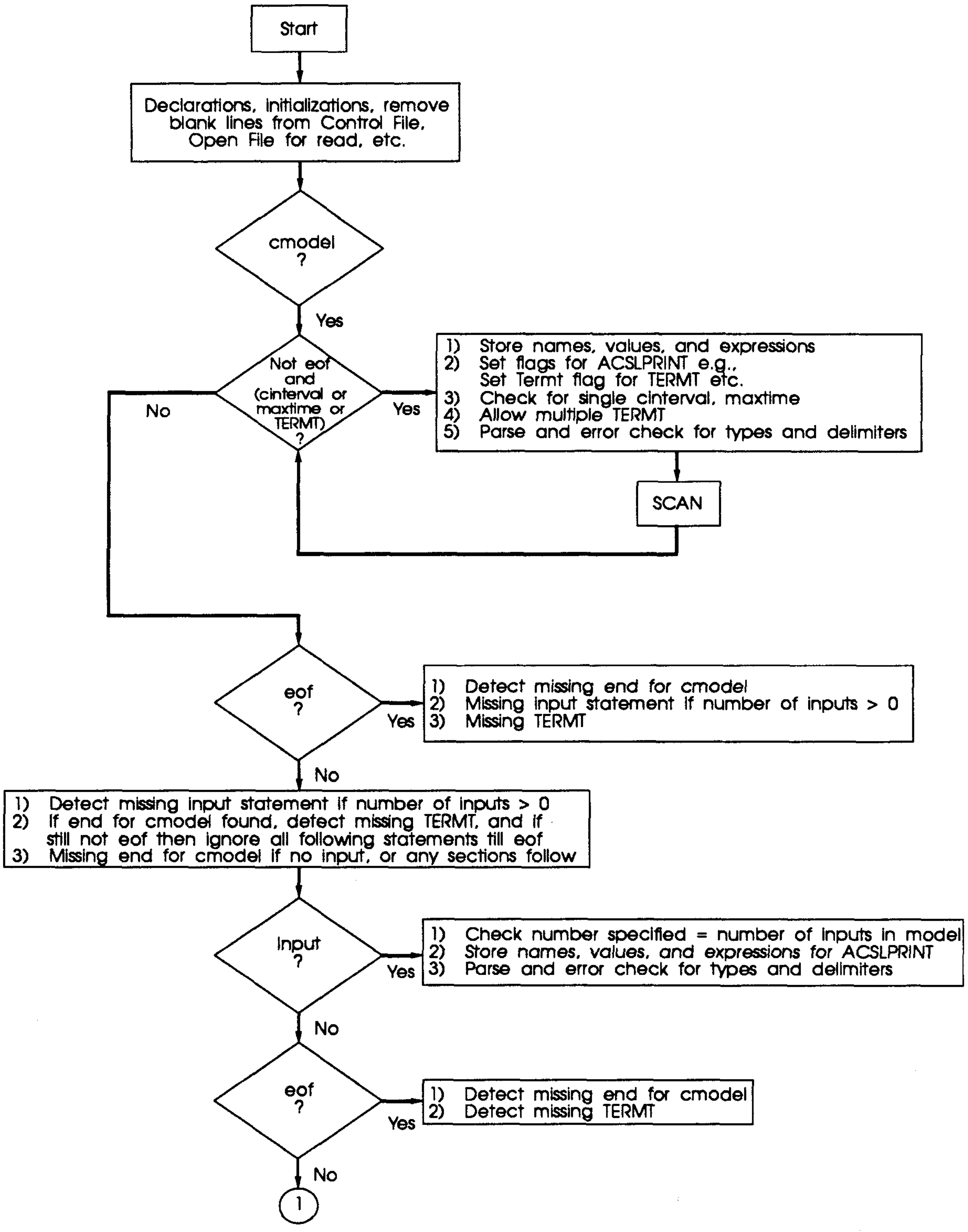
Start
Declarations, initializations, remove blank lines from Control File, Open File for read, etc.
cmodel ?
Yes
Not eof and (cinterval or maxtime or TERMT) ?
No
Yes
1) Store names, values, and expressions
2) Set flags for ACSLPRINT e.g., Set Termt flag for TERMT etc.
3) Check for single cinterval, maxtime
4) Allow multiple TERMT
5) Parse and error check for types and delimiters
SCAN
eof ?
Yes
1) Detect missing end for cmodel
2) Missing input statement if number of inputs > 0
3) Missing TERMT
No
1) Detect missing input statement if number of inputs > 0
2) If end for cmodel found, detect missing TERMT, and if still not eof then ignore all following statements till eof
3) Missing end for cmodel if no input, or any sections follow
Input ?
Yes
1) Check number specified = number of inputs in model
2) Store names, values, and expressions for ACSLPRINT
3) Parse and error check for types and delimiters
No
eof ?
Yes
1) Detect missing end for cmodel
2) Detect missing TERMT
No
1

Figure 3.4 Algorithm for Parsing and Error Checking Control File

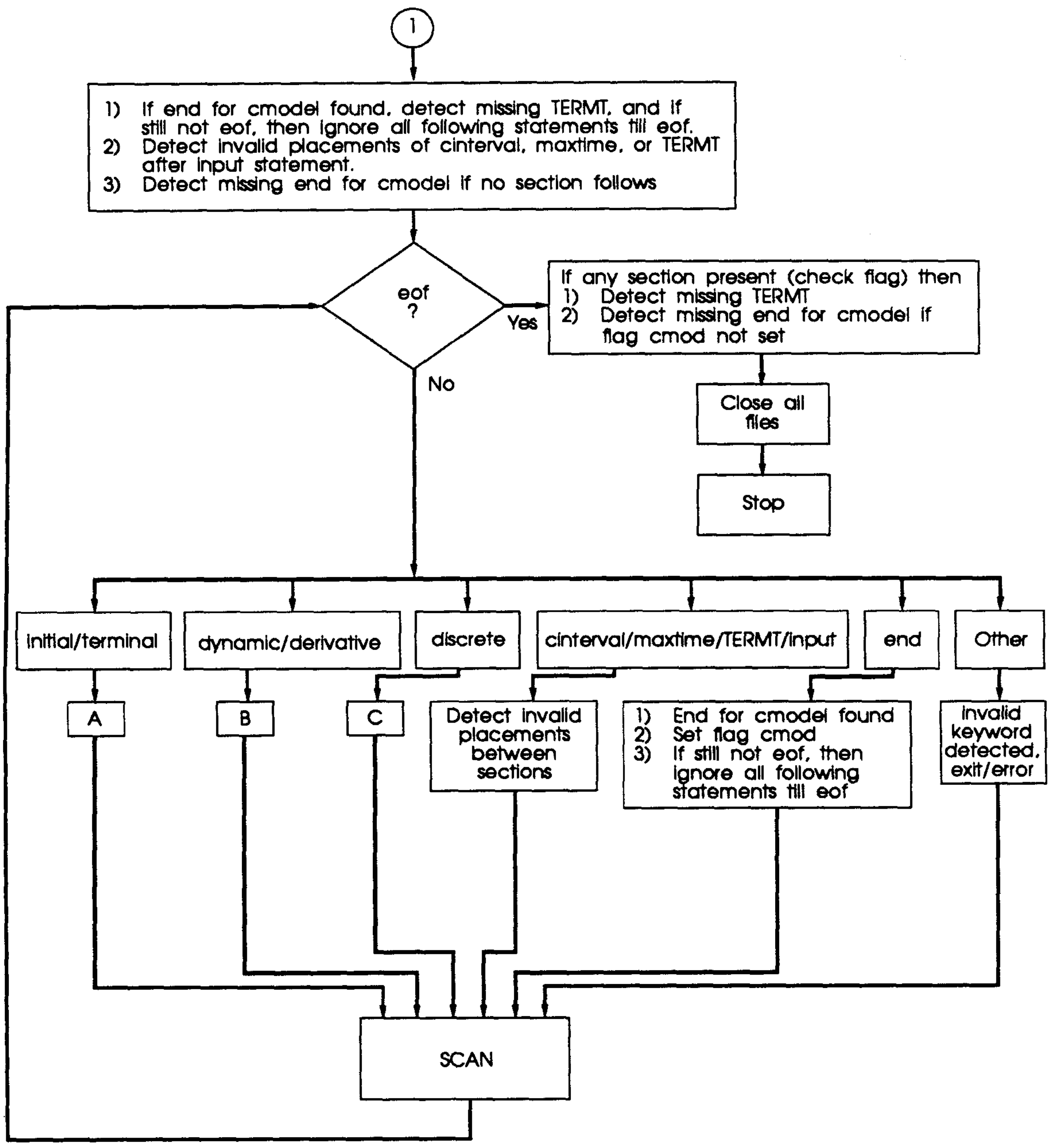
1
1) If end for cmodel found, detect missing TERMT, and if still not eof, then ignore all following statements till eof.
2) Detect invalid placements of cinterval, maxtime, or TERMT after input statement.
3) Detect missing end for cmodel if no section follows
eof ?
Yes
No
If any section present (check flag) then 1) Detect missing TERMT 2) Detect missing end for cmodel if flag cmod not set
Close all files
Stop
initial/terminal
dynamic/derivative
discrete
cinterval/maxtime/TERMT/input
end
Other
A
B
C
Detect invalid placements between sections
1) End for cmodel found 2) Set flag cmod 3) If still not eof, then ignore all following statements till eof
Invalid keyword detected, exit/error
SCAN

Figure 3.4 Algorithm for Parsing and Error Checking Control File (cont.)

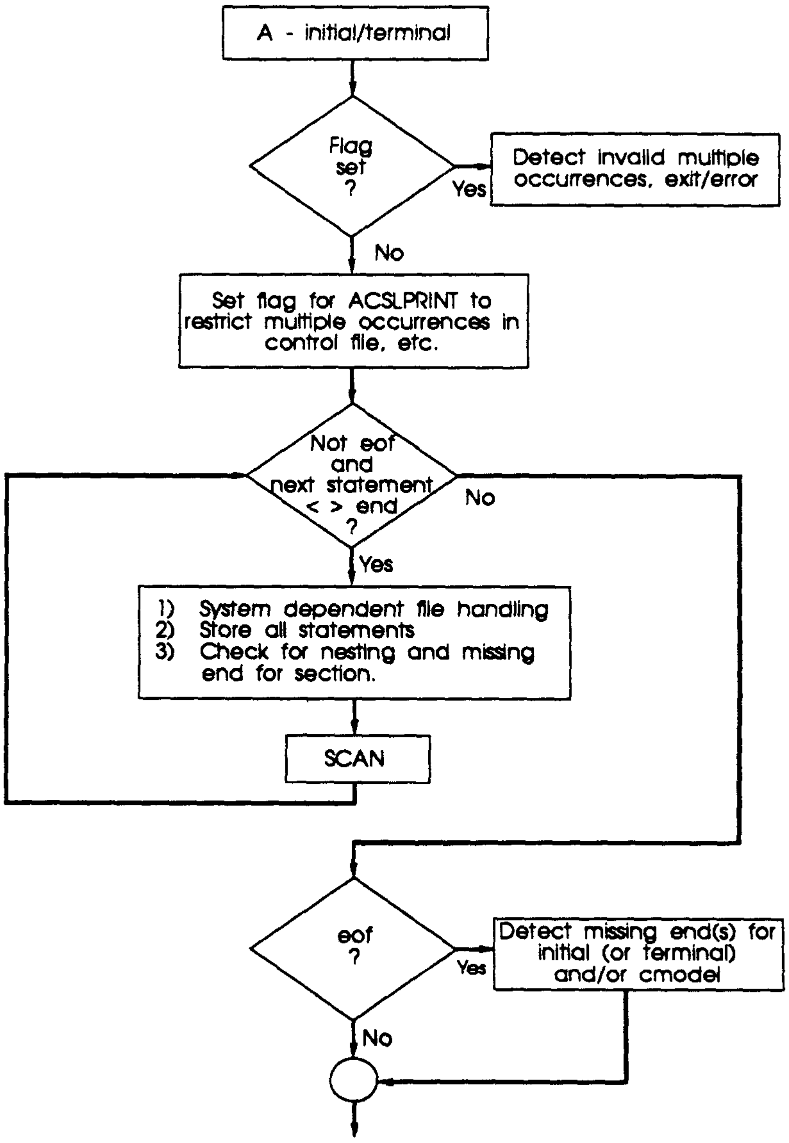
A - initial/terminal
Flag set ?
Yes
Detect invalid multiple occurrences, exit/error
No
Set flag for ACSLPRINT to restrict multiple occurrences in control file, etc.
Not eof and next statement < > end ?
No
Yes
1) System dependent file handling
2) Store all statements
3) Check for nesting and missing end for section.
SCAN
eof ?
Yes
Detect missing end(s) for initial (or terminal) and/or cmodel
No

Figure 3.4 Algorithm for Parsing and Error Checking Control File (cont.)

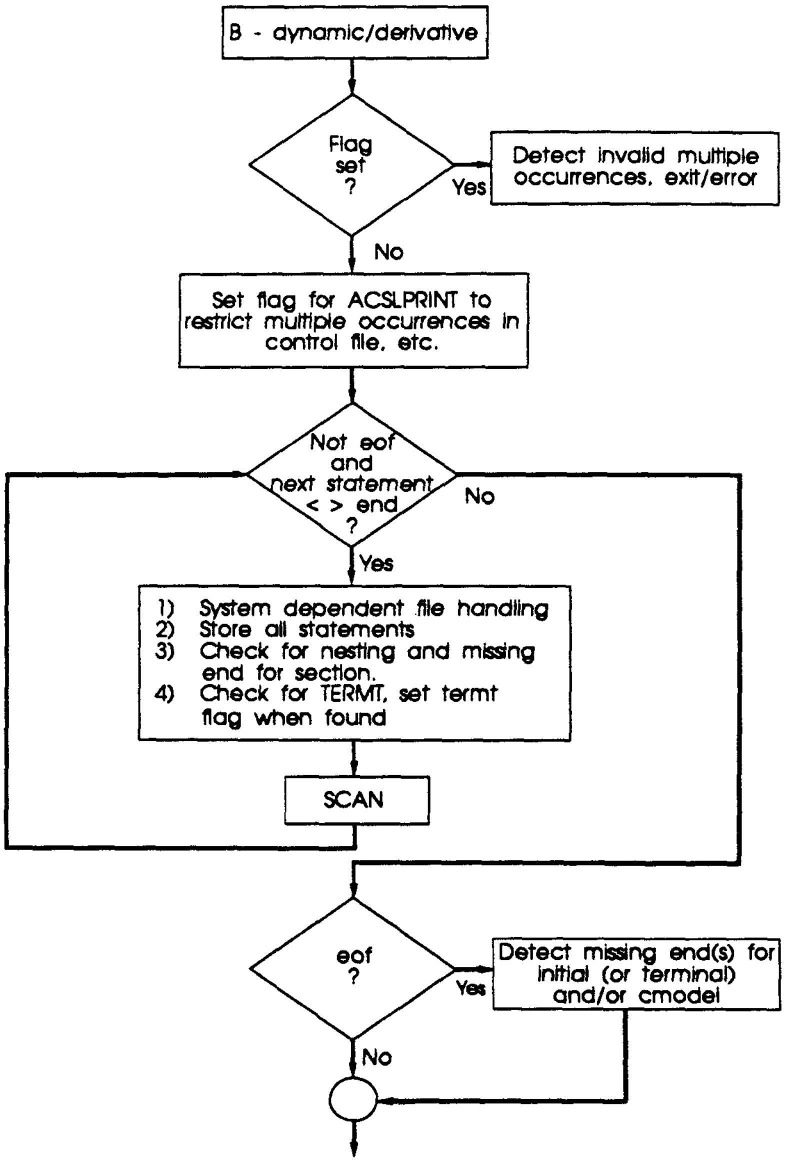
B - dynamic/derivative
Flag set ?
Yes
Detect invalid multiple occurrences, exit/error
No
Set flag for ACSLPRINT to restrict multiple occurrences in control file, etc.
Not eof and next statement < > end ?
No
Yes
1) System dependent file handling
2) Store all statements
3) Check for nesting and missing end for section.
4) Check for TERMT, set termt flag when found
SCAN
eof ?
Yes
Detect missing end(s) for initial (or terminal) and/or cmodel
No

Figure 3.4 Algorithm for Parsing and Error Checking Control File (cont.)

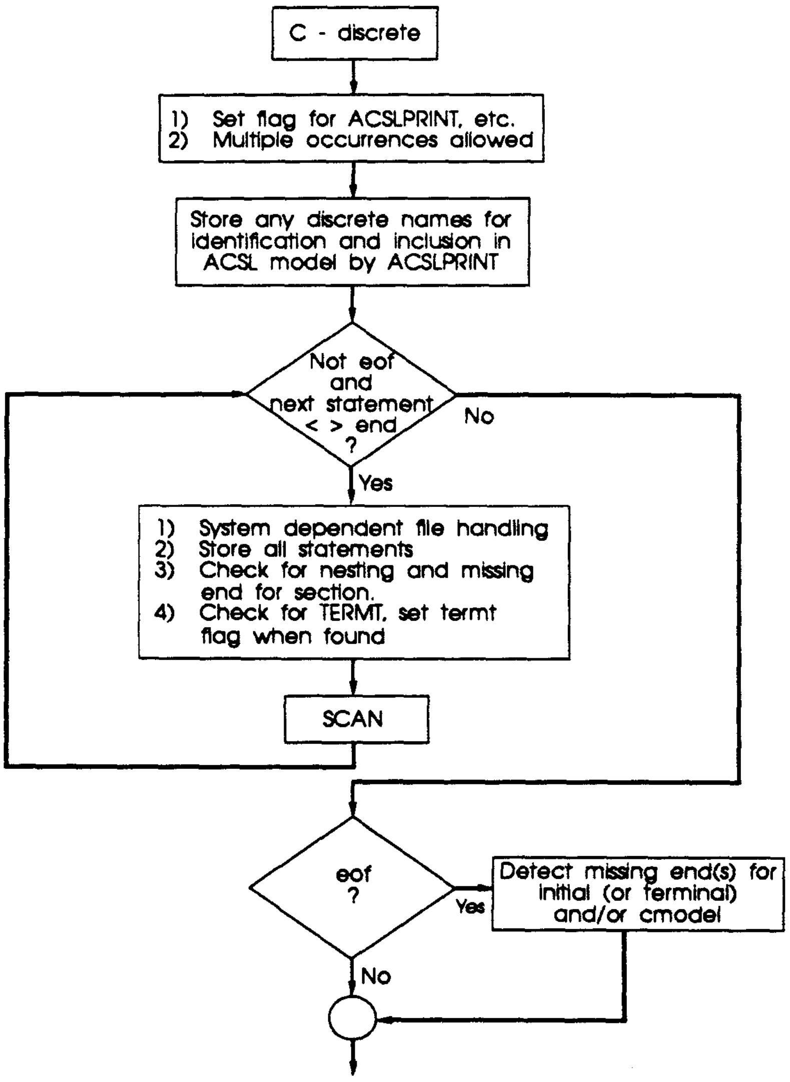
C - discrete
1) Set flag for ACSLPRINT, etc.
2) Multiple occurrences allowed
Store any discrete names for identification and inclusion in ACSL model by ACSLPRINT
Not eof and next statement < > end ?
No
Yes
1) System dependent file handling
2) Store all statements
3) Check for nesting and missing end for section.
4) Check for TERMT, set termt flag when found
SCAN
eof ?
Yes
Detect missing end(s) for initial (or terminal) and/or cmodel
No

Figure 3.4 Algorithm for Parsing and Error Checking Control File (cont.)

CHAPTER 4

VALIDATION WITH EXAMPLES

The validation of the ACSL interface implementation, and formal verification that results satisfy the documented specifications proceeds with a testing process. The basic objective is to validate the resulting software using an iterative process of coding and testing activities as errors are uncovered during testing. The testing process identifies not only errors in the code, but also functions that may not have been specified initially, and as a result not implemented. These functions may be capabilities that facilitate the user, or are absolutely essential to the user, and hence are highly desirable. This process involves sofware modifications and/or further development, and consequently further validation. The ACSL interface code validation is performed with selected examples in a variety of application areas. The chosen examples serve as good test suites.

Several examples use bond graphs as modeling elements to formulate model descriptions of the systems. The @ operator in DYMOLA programs includes each element definition stored in a separate file. The DYMOLA examples described using the bond graph approach uses only a subset of all the basic bond graphs available as modeling elements. The complete set of basic bond graphs is shown in Figure 4.1.

```
model type bond
   cut A(x/y), B(y/−x)
   main cut C[A,B]
   main path P<A−B>
end
model type SE
   main cut A(e/.)
   terminal E0
   E0 = e
end
model type SF
   main cut A(./−f)
   terminal F0
   F0 = f
end
model type R
   main cut A(e/f)
   parameter R=1.0
   R∗f = e
end
model type C
   main cut A(e/f)
   parameter C=1.0
   C∗der(e) = f
end
model type I
   main cut A(e/f)
   parameter I=1.0
   I∗der(f) = e
end
model type TF
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter m=1.0
   e1 = m∗e2
   f2 = m∗f1
end
model type GY
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter r=1.0
   e1 = r∗f2
   e2 = r∗f1
end
```

Figure 4.1 DYMOLA Bond Graphs

```
model type G
   main cut A(e/f)
   parameter G=1.0
   G*e = f
end
model type mG
   main cut A(e/f)
   terminal G
   f = G*sqrt(AMAX1(e,0.0))
end
```

Figure 4.1 DYMOLA Bond Graphs (cont.)

The reader is encouraged to refer to *Continuous-System Modeling* (Cellier, 1991) for concepts behind Bond Graph Modeling, techniques and terminology.

4.1 Electrical Network I

A simple RLC circuit is modeled using the bond graph approach. The electrical circuit diagram is shown in Figure 4.2.

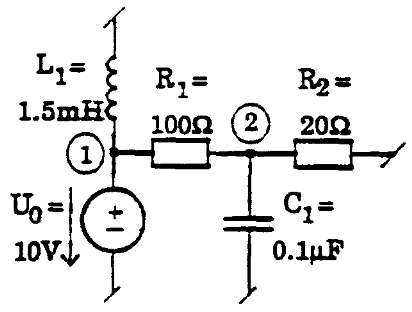

Figure 4.2 Simple RLC Circuit

Basic bond graphs are used as modeling elements for this example. The DYMOLA expanded bond graph with all elements attached to 0-junctions only is shown in Figure 4.3.

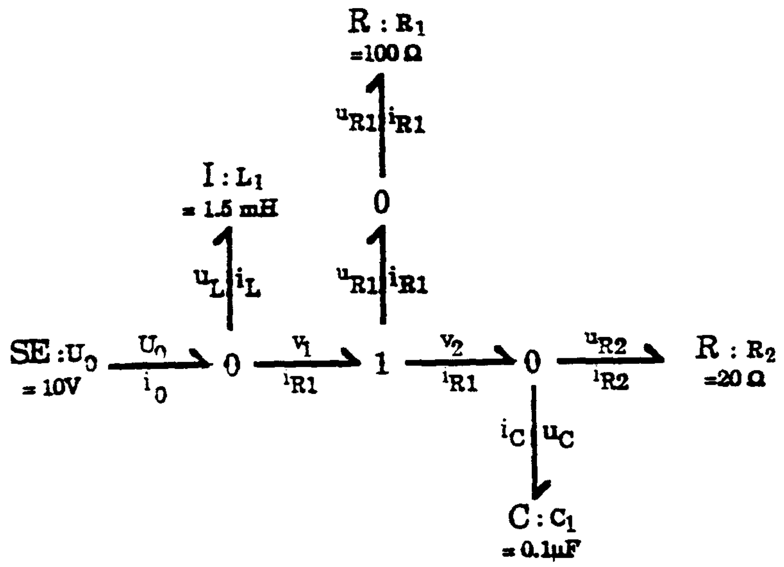

Figure 4.3 DYMOLA Bond Graph of simple RLC Circuit

The DYMOLA program used to code the bond graph is shown in Figure 4.4.

```
@bond.bnd
@se.bnd
@r.bnd
@c.bnd
@i.bnd

model RLC

  submodel (SE)   U0
  submodel (R)    R1(R=100.0), R2(R=20.0)
  submodel (I)    L1(I=1.5E-3) (ic f = 0.5)
  submodel (C)    C1(C=0.1E-6)
  submodel (bond) B1, B2, B3

  node   v1, ir1, vr1, v2
  input  u
  output y1, y2

  connect U0 at v1
  connect L1 at v1
```

Figure 4.4 DYMOLA Program for simple RLC Circuit

```
  connect R1 at vr1
  connect R2 at v2
  connect C1 at v2
  connect B1 from v1 to ir1
  connect B2 from ir1 to v2
  connect B3 from ir1 to vr1

  U0.E0  = u
  y1     = C1.e
  y2     = R2.f

end
```

Figure 4.4 DYMOLA Program for simple RLC Circuit (cont.)

The simple control file used for this example is shown in Figure 4.5.

```
cmodel

maxtime tmax = 2E-5
TERMT (t .GE. tmax)
cinterval cint = 2E-7
input 1, u(independ,10.0)

end
```

Figure 4.5 Control File for simple RLC Circuit

The ACSL program generated by DYMOLA's ACSL interface is shown in Figure 4.6.

```
"--------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"--------------------------------------------------------------"

PROGRAM RLC

  INITIAL

    CONSTANT ...
      R1XR=100.0, C=0.1E-6, I=1.5E-3, ...
      R2XR=20.0
    CINTERVAL cint = 2E-7
    CONSTANT tmax = 2E-5
    CONSTANT ...
      u = 10.0

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"---------------------------------------------Submodel: R2"
      R2Xf = C1Xe/R2XR
"---------------------------------------------Submodel: RLC"
      B3Xy = u - C1Xe
"---------------------------------------------Submodel: R1"
      B3Xx = B3Xy/R1XR
"---------------------------------------------Submodel: RLC"
      C1Xf = B3Xx - R2Xf
"---------------------------------------------Submodel: C1"
      C1Xe = INTEG(C1Xf/C, 0)
"---------------------------------------------Submodel: L1"
      L1Xf = INTEG(u/I, 0.5)
"---------------------------------------------Submodel: RLC"
      y1 = C1Xe
      y2 = R2Xf

    END $ "of DERIVATIVE"

    TERMT (t .GE. tmax)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure 4.6 ACSL Program for simple RLC Circuit

The resulting ACSL program is compiled to produce results for output variables which can be viewed using graphs. The graphs are produced after the simulation run is completed by use of CTRL-C's graphic routines and are shown in Figure 4.7.

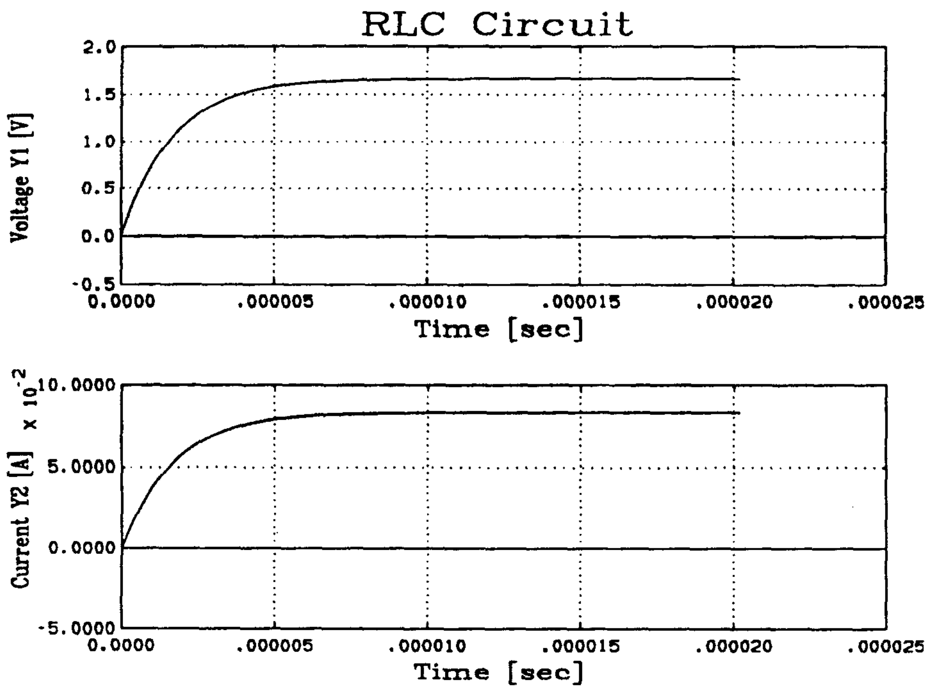

Figure 4.7 Simulation Waveforms for simple RLC Circuit

4.2 Electrical Network II

The electrical circuit diagram for the RLC circuit used for this example is shown in Figure 4.8.

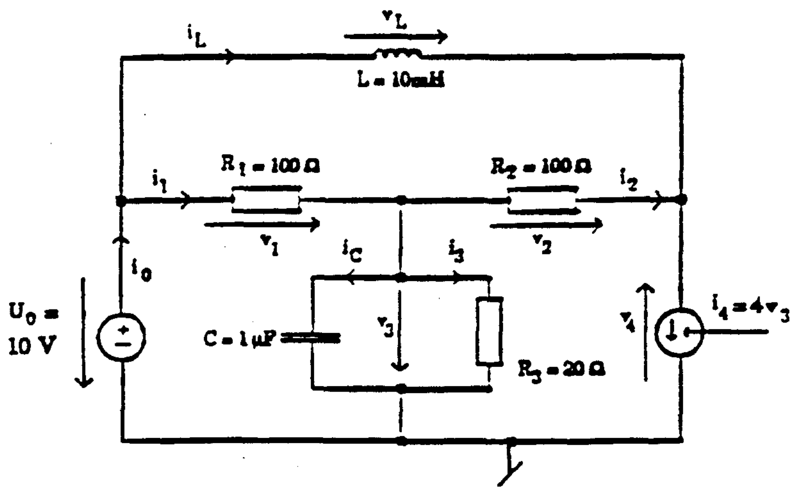

Figure 4.8 RLC Network

The DYMOLA program used to model the electrical network with the bond graph approach is shown in Figure 4.9.

```
{ Bond Graph model of an Electrical Network }

@r.bnd
@c.bnd
@i.bnd
@se.bnd
@sf.bnd
@bond.bnd

model network

  submodel (R)    R1(R=100.0), R2(R=100.0), R3(R=20.0)
  submodel (C)    C(C=1.0E-6) (ic e=0.02)
  submodel (I)    L(I=10.0E-3)
```

Figure 4.9 DYMOLA Program for RLC Network

```
  submodel (SE)   U0
  submodel (SF)   mI4
  submodel (bond) B1, B2, B3, B4, B5, B6, B7, B8, B9

  input  u
  output y1, y2
  node   v0, i1, dR1, v3, dR2, mv4, iL, dL, i2

  connect U0  at   v0,          ->
          B1  from v0  to i1,   ->
          B2  from i1  to dR1,  ->
          R1  at   dR1,         ->
          B3  from i1  to v3,   ->
          C   at   v3,          ->
          R3  at   v3,          ->
          B4  from v3  to i2,   ->
          B5  from i2  to dR2,  ->
          R2  at   dR2,         ->
          B6  from i2  to mv4,  ->
          B7  from v0  to iL,   ->
          B8  from iL  to dL,   ->
          L   at   dL,          ->
          B9  from iL  to mv4,  ->
          mI4 at   mv4

  mI4.F0 = -4.0*C.e
  U0.E0  = u
  y1     = 10.0*R3.e
  y2     = 10.0*C.f

end
```

Figure 4.9 DYMOLA Program for RLC Network (cont.)

The control file used for this example is shown in Figure 4.10.

```
cmodel
maxtime tmax = 50.0E-6
cinterval cint = 50.0E-9
TERMT (t .GE. tmax)
input 1, u(independ,10.0)
end
```

Figure 4.10 Control File for RLC Network

The resulting ACSL program is shown in Figure 4.11.

```
"--------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"--------------------------------------------------------------"

PROGRAM network

  INITIAL

    CONSTANT ...
      R1XR=100.0, C=1.0E-6, I=10.0E-3, ...
      R2XR=100.0, R3XR=20.0
    CINTERVAL cint = 50.0E-9
    CONSTANT tmax = 50.0E-6
    CONSTANT ...
      u = 10.0

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"----------------------------------------------Submodel: network"
      R1Xe = u - CXe
      mI4Xf = -4.0*CXe
      B6Xx = (mI4Xf + LXf)/(-1)
"----------------------------------------------Submodel: R3"
      R3Xf = CXe/R3XR
"----------------------------------------------Submodel: R1"
      B3Xx = R1Xe/R1XR
"----------------------------------------------Submodel: network"
      CXf = B3Xx - (B6Xx + R3Xf)
"----------------------------------------------Submodel: C"
      CXe = INTEG(CXf/C, 0.02)
"----------------------------------------------Submodel: R2"
      R2Xe = R2XR*B6Xx
"----------------------------------------------Submodel: network"
      B9Xy = CXe - R2Xe
      LXe = u - B9Xy
"----------------------------------------------Submodel: L"
      LXf = INTEG(LXe/I, 0)
```

Figure 4.11 ACSL Program for RLC Network

```
"---------------------------------------------Submodel: network"
      y1 = 10.0*CXe
      y2 = 10.0*CXf

    END $ "of DERIVATIVE"

    TERMT (t .GE. tmax)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure 4.11 ACSL Program for RLC Network (cont.)

The ACSL program is compiled, and the resulting waveforms for the output variables produced by CTRL-C are shown in Figure 4.12.

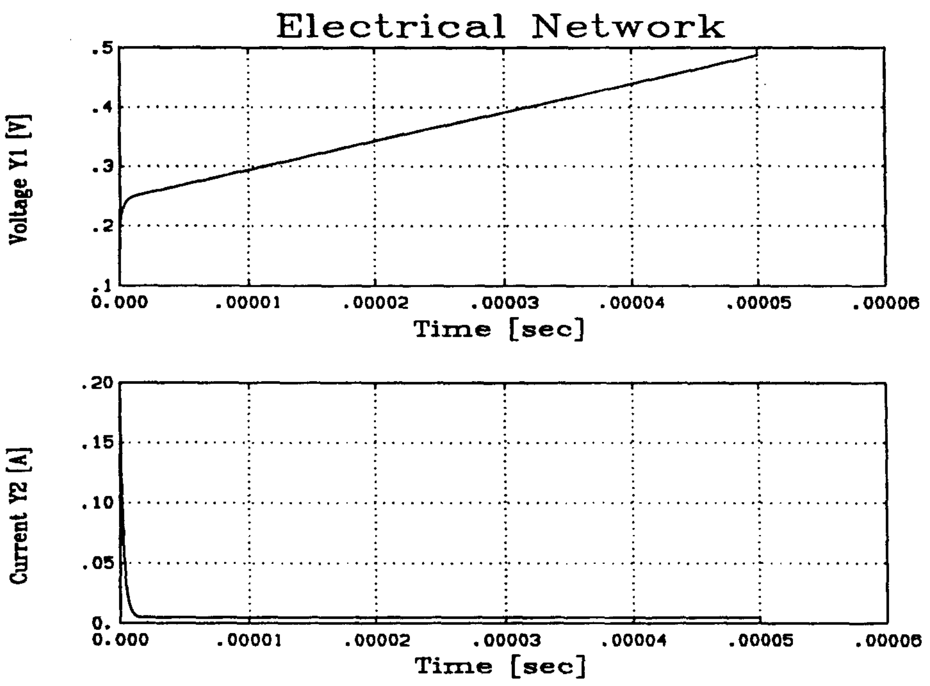

Figure 4.12 Simulation Waveforms for RLC Network

Detailed descriptions for the following three examples are found in *Continuous-System Modeling* (Cellier, 1991). The purpose here is to demonstrate the results of DYMOLA's ACSL interface using example programs and control files.

4.3 Hydraulic System

The system consists of a hydraulic motor with a four way servo valve. The DYMOLA program used to model the hydraulic system with the bond graph approach is shown in Figure 4.13.

```
{ Bond Graph model of a Hydraulic System }

@h7_7.r
@h7_7.g
@h7_7.c
@h7_7.i
@h7_7.se
@h7_7.tf
@h7_7.mg
@h7_7.bnd

model type limit
  cut A(e), B(u)
  main cut C[A B]
  main path P <A - B>
  u = BOUND(-1.0,1.0,e)
end

model type contr
  cut A(u), B(x)
  main cut C [A B]
  main path P <A - B>
  parameter tausv=0.005
  x + tausv*der(x) = u
end

model type hydro

  submodel (R)    rhom(R=1.5)
  submodel (G)    Gi(G=0.737E-13), Ge1(G=0.737E-12), Ge2(G=0.737E-12)
```

Figure 4.13 DYMOLA Program for Hydraulic System

```
submodel (mG)   mG1, mG2, mG3, mG4
submodel (I)    Jm(I=0.08)
submodel (C)    C1(C=0.1707E-13), C2(C=0.1707E-13)  { C=1/c1 }
submodel (SE)   PS, P0
submodel (TF)   tf(m=1.7391E+5)  { m=1/psi }
submodel (bond) B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, ->
                B12, B13, B14, B15, B16, B17, B18, B19, B20,  ->
                B21, B22, B23, B24, B25

cut A(x), B(thetam)
main cut C[A B]
main path P <A - B>

parameter x0=0.05, k=0.248E-6
local     xp, xn
node nPS, nP0, p1, p2, q1, q2, q3, q4, dmG1, dmG2, dmG3, dmG4, ->
     qL1, qL2, np1, np2, qi, dGi, qind, dTF1, dTF2, thmd, drho, dJm

connect PS   at   nPS,           ->
        P0   at   nP0,           ->
        B1   from nPS  to q4,    ->
        B2   from q4   to p2,    ->
        B3   from p2   to q3,    ->
        B4   from q3   to nP0,   ->
        B5   from nPS  to q1,    ->
        B6   from q1   to p1,    ->
        B7   from p1   to q2,    ->
        B8   from q2   to nP0,   ->
        B9   from q1   to dmG1,  ->
        mG1  at   dmG1,          ->
        B10  from q2   to dmG2,  ->
        mG2  at   dmG2,          ->
        B11  from q3   to dmG3,  ->
        mG3  at   dmG3,          ->
        B12  from q4   to dmG4,  ->
        mG4  at   dmG4,          ->
        B13  from p1   to qL1,   ->
        B14  from qL1  to np1,   ->
        Ge1  at   np1,           ->
        C1   at   np1,           ->
        B15  from qL2  to p2,    ->
        B16  from np2  to qL2,   ->
        Ge2  at   np2,           ->
        C2   at   np2,           ->
        B17  from np1  to qi,    ->
        B18  from qi   to np2,   ->
        B19  from qi   to dGi,   ->
```

Figure 4.13 DYMOLA Program for Hydraulic System (cont.)

```
          Gi   at   dGi,          ->
          B20  from np1  to qind, ->
          B21  from qind to np2,  ->
          B22  from qind to dTF1, ->
          tf   from dTF1 to dTF2, ->
          B23  from dTF2 to thmd, ->
          B24  from thmd to drho, ->
          rhom at   drho,         ->
          B25  from thmd to dJm,  ->
          Jm   at   dJm

  PS.E0 = 0.137E+8
  P0.E0 = 0.0
  xp    = k*AMAX1(x0+x,0.0)
  xn    = k*AMAX1(x0-x,0.0)
  mG1.G = xp
  mG2.G = xn
  mG3.G = xp
  mG4.G = xn
  Jm.f  = der(thetam)

end

model hydraulic

  submodel limit
  submodel contr
  submodel hydro

  input  thset
  output thetam
  local  e

  connect limit - contr - hydro

  e = thset - thetam
  limit.e = e
  hydro.thetam = thetam

end
```

Figure 4.13 DYMOLA Program for Hydraulic System (cont.)

The control file used for the model's experiment description is shown in Figure 4.14.

```
cmodel

maxtime tmx = 0.2
cinterval cint = 0.002
TERMT (t.ge.tmx)
input 1, thset(independ,1.0)

end
```

Figure 4.14 Control File for Hydraulic System

The ACSL program generated from the DYMOLA program and the control file is shown in Figure 4.15.

```
"-------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"-------------------------------------------------------------"

PROGRAM hydraulic

  INITIAL

    CONSTANT ...
      R=1.5, GiXG=0.737E-13, C1XC=0.1707E-13, ...
      I=0.08, m=1.7391E+5, tausv=0.005, ...
      Ge1XG=0.737E-12, Ge2XG=0.737E-12, C2XC=0.1707E-13, ...
      x0=0.05, k=0.248E-6
    CINTERVAL cint = 0.002
    CONSTANT tmx = 0.2
    CONSTANT ...
      thset = 1.0

  END $ "of INITIAL"
```

Figure 4.15 ACSL Program for Hydraulic System

```
 DYNAMIC

   DERIVATIVE

"---------------------------------------------Submodel: hydro::rhom"
      rhomXe = R*JmXf
"---------------------------------------------Submodel: hydro"
      GiXe = C1Xe - C2Xe
"---------------------------------------------Submodel: hydro::tf"
      B22Xx = JmXf/m
"---------------------------------------------Submodel: hydro::Gi"
      B19Xx = GiXG*GiXe
"---------------------------------------------Submodel: hydro::Ge1"
      Ge1Xf = Ge1XG*C1Xe
"---------------------------------------------Submodel: hydro"
      xn = k*AMAX1(x0 - cotrXx,0.0)
      B8Xy = 0.0
      mG2Xe = C1Xe - B8Xy
"---------------------------------------------Submodel: hydro::mG2"
      B10Xx = xn*sqrt(AMAX1(mG2Xe,0.0))
"---------------------------------------------Submodel: hydro"
      xp = k*AMAX1(x0 + cotrXx,0.0)
      B5Xx = 0.137E+8
      mG1Xe = B5Xx - C1Xe
"---------------------------------------------Submodel: hydro::mG1"
      B9Xx = xp*sqrt(AMAX1(mG1Xe,0.0))
"---------------------------------------------Submodel: hydro"
      B14Xx = B9Xx - B10Xx
      C1Xf = B14Xx - (B22Xx + B19Xx + Ge1Xf)
"---------------------------------------------Submodel: hydro::C1"
      C1Xe = INTEG(C1Xf/C1XC, 0)
"---------------------------------------------Submodel: hydro"
      e1 = C1Xe - C2Xe
"---------------------------------------------Submodel: hydro::tf"
      B23Xx = e1/m
"---------------------------------------------Submodel: hydro"
      JmXe = B23Xx - rhomXe
"---------------------------------------------Submodel: hydro::Jm"
      JmXf = INTEG(JmXe/I, 0)
"---------------------------------------------Submodel: hydraulic"
      thetam = hoXthm
      hyicXe = thset - thetam
"---------------------------------------------Submodel: limit"
      liitXu = BOUND(-1.0,1.0,hyicXe)
"---------------------------------------------Submodel: contr"
      cotrXx = INTEG((liitXu - cotrXx)/tausv, 0)
```

Figure 4.15 ACSL Program for Hydraulic System (cont.)

```
"---------------------------------------------Submodel: hydro::Ge2"
      Ge2Xf = Ge2XG*C2Xe
"---------------------------------------------Submodel: hydro"
      mG3Xe = C2Xe - B8Xy
      mG4Xe = B5Xx - C2Xe
"---------------------------------------------Submodel: hydro::mG3"
      B11Xx = xp*sqrt(AMAX1(mG3Xe,0.0))
"---------------------------------------------Submodel: hydro::mG4"
      B12Xx = xn*sqrt(AMAX1(mG4Xe,0.0))
"---------------------------------------------Submodel: hydro"
      B16Xy = B11Xx - B12Xx
      C2Xf = B22Xx + B19Xx - (Ge2Xf + B16Xy)
"---------------------------------------------Submodel: hydro::C2"
      C2Xe = INTEG(C2Xf/C2XC, 0)
"---------------------------------------------Submodel: hydro"
      hoXthm = INTEG(JmXf, 0)

    END $ "of DERIVATIVE"

    TERMT (t.ge.tmx)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure 4.15 ACSL Program for Hydraulic System (cont.)

The ACSL program is compiled, and the resulting waveform for *thetam* is shown in Figure 4.16.

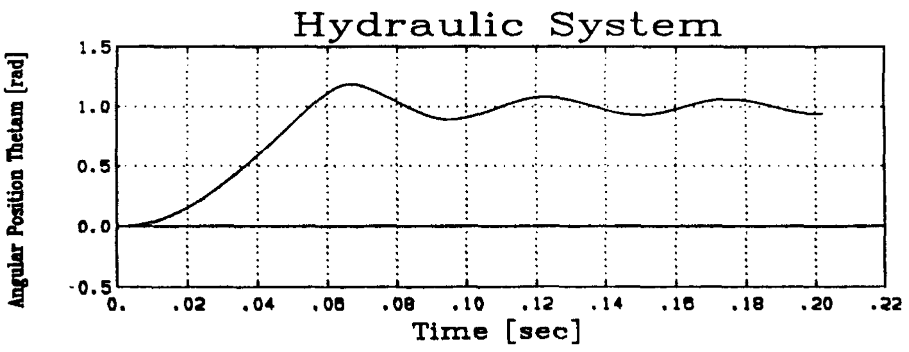

Figure 4.16 Simulation Waveform for Hydraulic System

4.4 Cable Reel

The system is comprised of a cable reel, a large DC motor that unrolls cable from the reel, a speedometer that detects the speed of the cable as it comes off the reel, and a simple proportional and integral (PI) controller used to keep the cable speed v at its preset value $Vset$. The DYMOLA program that describes the behavior of the cable reel system is shown in Figure 4.17.

```
model type CableReel
  cut vport(v), fport(F)
  cut mech(omega,tauL,JL)
  local R
  parameter Rempty, W, D, rho, BL, J0
  constant pi = 3.14159
    der(R) = -((D*D)/(2.0*pi*W))*omega
    v      = R*omega
    JL     = 0.5*pi*W*rho*(R**4 - Rempty**4) + J0
    tauL   = BL*omega - F*R
end

model type DCMotor
  cut uport(ua)
  cut mech(omega,tauL,JL)
  local ia, if, ui, psi, taum, Twist, theta, uf
  parameter Ra, Rf, kmot, Jm, Bm = 0.0
    uf         = 25.0
    0.0        = uf - Rf*if
    0.0        = ua - ui - Ra*ia
    psi        = kmot*if
    taum       = psi*ia
    ui         = psi*omega
    der(Twist) = taum - tauL - Bm*omega
    Twist      = (Jm+JL)*omega
    der(theta) = omega
end

model type Comparator
  cut setport(Vset), measport(Vmeas), errport(error)
    error = Vset - Vmeas
end
```

Figure 4.17 DYMOLA Program for Cable Reel System

```
model type Cable
  cut finport(Fin), errport(error), fport(F)
  parameter kship = 10.0
    F = AMAX1(kship*error - Fin,0.0) + Fin
end

model type PIController
  cut signal(error), command(u)
  local err
  parameter kint, kprop
    der(err) = error
    u        = kprop*error + kint*err
end

model type Speedometer
  cut vport(v), measport(Vmeas)
  local x
  parameter k = 3.0
    der(x) = -k*x + v
    Vmeas  = k*x
end

model Cabsys

  submodel Comparator
  submodel PIController(kint=0.2, kprop=6.0)
  submodel DCMotor(Ra=0.25, Rf=1.0, kmot=1.5, ->
                  Jm=5.0, Bm=0.2)
  submodel CableReel(Rempty=0.6, W=1.5, D=0.0127, ->
                    rho=1350.0, BL=6.5, J0=150.0) ->
                   (ic R=1.2)
  submodel Speedometer
  submodel Cable

  input Vdes, Fext
  output radius, veloc, omega

  connect Comparator:errport at PIController:signal
  connect PIController:command at DCMotor:uport
  connect DCMotor:mech at CableReel:mech
  connect CableReel:vport at Speedometer:vport
  connect Speedometer:measport at Comparator:measport
  connect Comparator:errport at Cable:errport
  connect Cable:fport at CableReel:fport
```

Figure 4.17 DYMOLA Program for Cable Reel System (cont.)

```
  Cable.Fin = Fext
  Comparator.Vset = Vdes
  radius = CableReel.R
  veloc = CableReel.v
  omega = CableReel.omega

end
```

Figure 4.17 DYMOLA Program for Cable Reel System (cont.)

The control file used for this example is shown in Figure 4.18.

```
cmodel

maxtime tmx = 3500
cinterval cint = 1.0
TERMT (t.ge.tmx .or. R.lt.Rempty)
input 2, Vdes(independ,15.0), Fext(independ,100.0)

initial
NSTEPS NSTP = 1000
end

end
```

Figure 4.18 Control File for Cable Reel System

The ACSL program generated from the DYMOLA program and the control file is shown in Figure 4.19.

```
"--------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"--------------------------------------------------------------"

PROGRAM Cabsys

  INITIAL

    CONSTANT ...
      Rempty=0.6, W=1.5, D=0.0127, ...
      rho=1350.0, BL=6.5, J0=150.0, ...
      pi=3.14159, Ra=0.25, Rf=1.0, ...
      kmot=1.5, Jm=5.0, Bm=0.2, ...
      kship=10.0, kint=0.2, kprop=6.0, ...
      k=3.0
    CINTERVAL cint = 1.0
    CONSTANT tmx = 3500
    CONSTANT ...
      Vdes = 15.0, Fext = 100.0
    NSTEPS NSTP = 1000

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"---------------------------------------------Submodel: CableReel"
      DCrXJL = 0.5*pi*W*rho*(R**4 - Rempty**4) + J0
"---------------------------------------------Submodel: DCMotor"
      DrXoma = Twist/(Jm + DCrXJL)
"---------------------------------------------Submodel: CableReel"
      CaelXv = R*DrXoma
      R = INTEG(-D*D/(2*pi*W)*DrXoma, 1.2)
"---------------------------------------------Submodel: Speedometer"
      SrXVms = k*x
"---------------------------------------------Submodel: Comparator"
      CrXerr = Vdes - SrXVms
"---------------------------------------------Submodel: Cable"
      CaleXF = AMAX1(kship*CrXerr - Fext,0.0) + Fext
"---------------------------------------------Submodel: CableReel"
      DrXtaL = BL*DrXoma - CaleXF*R
"---------------------------------------------Submodel: PIController"
      u = kprop*CrXerr + kint*err
"---------------------------------------------Submodel: DCMotor"
      uf = 25.0
      DCrXif = uf/Rf
```

Figure 4.19 ACSL Program for Cable Reel System

```
      psi = kmot*DCrXif
      ui = psi*DrXoma
      ia = (u - ui)/Ra
      taum = psi*ia
      Twist = INTEG(taum - DrXtaL - Bm*DrXoma, 0)
"----------------------------------------------Submodel: PIController"
      err = INTEG(CrXerr, 0)
"----------------------------------------------Submodel: Speedometer"
      x = INTEG(CaelXv - k*x, 0)
"----------------------------------------------Submodel: Cabsys"
      radius = R
      veloc = CaelXv
      omega = DrXoma

    END $ "of DERIVATIVE"

    TERMT (t.ge.tmx .or. R.lt.Rempty)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure 4.19 ACSL Program for Cable Reel System (cont.)

This ACSL program is compiled, and results of the simulation study are shown with the trajectory behavior in Figure 4.20.

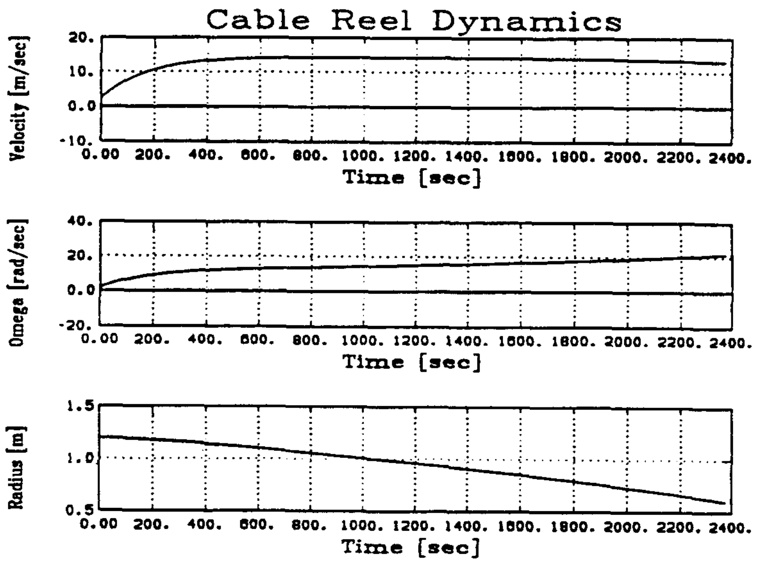

Figure 4.20 Simulation Waveforms for Cable Reel System

4.5 Lunar Lander

The lunar landing example describes the vertical motion of a rocket that is just about to perform a soft landing on the surface of the moon. The DYMOLA program describing the the dynamics of the system is shown in Figure 4.21.

```
model Lunar

constant r = 1738.0E3, c2 = 4.925E12
input  f1, f2
output h = 59404.0, v = -2003.0
parameter ff, cc
local  m = 1038.358, thrust, c1, a, mdot, g

thrust = ff
c1     = cc
der(h) = v
der(v) = a
a      = (1.0/m)*(thrust - m*g)
der(m) = mdot
mdot   = -c1*ABS(thrust)
g      = c2/(h + r)**2

end
```

Figure 4.21 DYMOLA Program for Lunar Lander

The control file shown in Figure 4.22 consists of an experiment description portion and also *initial*, *derivative*, and multiple *discrete* sections.

```
cmodel

maxtime tmx = 230.0
cinterval cint = 0.2
TERMT (t.GE.tmx .OR. h.LE.0.0 .OR. v.GT.0.0)
input 2, f1(independ,36350.0), f2(independ,1308.0)

initial
CONSTANT ...
c11 = 0.000277,  c12 = 0.000277, ...
tdec = 43.2,  tend = 210.0
ff = f1
cc = c11
end

derivative
SCHEDULE shutlg .XP. 9934.0-h
SCHEDULE shutsm .XP. 15.0-h
end

discrete shutlg
ff = f2
cc = c12
end

discrete shutsm
ff = 0.0
cc = 0.0
end

end
```

Figure 4.22 Control File for Lunar Lander

The ACSL program generated for this example is shown in Figure 4.23.

```
"--------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"--------------------------------------------------------------"

PROGRAM Lunar

  INITIAL

    CONSTANT ...
      r=1738.0E3, c2=4.925E12, ff=0, ...
      cc=0
    CINTERVAL cint = 0.2
    CONSTANT tmx = 230.0
    CONSTANT ...
      f1 = 36350.0, f2 = 1308.0
    CONSTANT ...
    c11 = 0.000277,  c12 = 0.000277, ...
    tdec = 43.2,  tend = 210.0
    ff = f1
    cc = c11

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"--------------------------------------------Submodel: Lunar"
      thrust = ff
      c1 = cc
      g = c2/(h + r)**2
      a = 1/m*(thrust - m*g)
      mdot = -c1*ABS(thrust)
      h = INTEG(v, 59404.0)
      v = INTEG(a, -2003.0)
      m = INTEG(mdot, 1038.358)

      SCHEDULE shutlg .XP. 9934.0-h
      SCHEDULE shutsm .XP. 15.0-h

    END $ "of DERIVATIVE"
```

Figure 4.23 ACSL Program for Lunar Lander

```
    DISCRETE shutlg

      ff = f2
      cc = c12

    END $ "of DISCRETE shutlg"

    DISCRETE shutsm

      ff = 0.0
      cc = 0.0

    END $ "of DISCRETE shutsm"

    TERMT (t.GE.tmx .OR. h.LE.0.0 .OR. v.GT.0.0)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure 4.23 ACSL Program for Lunar Lander (cont.)

This ACSL program is compiled, and results of the simulation study are presented using waveforms from CTRL-C. Figure 4.24 shows the time trajectories of five of the simulation variables.

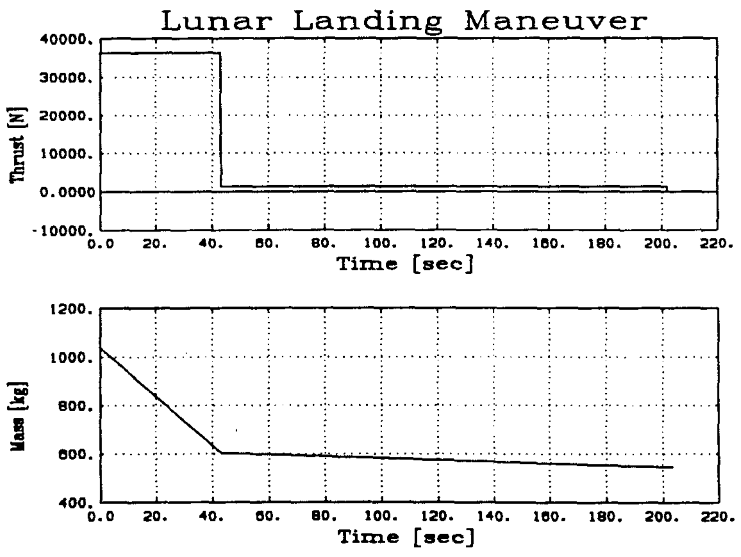

Figure 4.24 Simulation Waveforms for Lunar Landing Maneuver

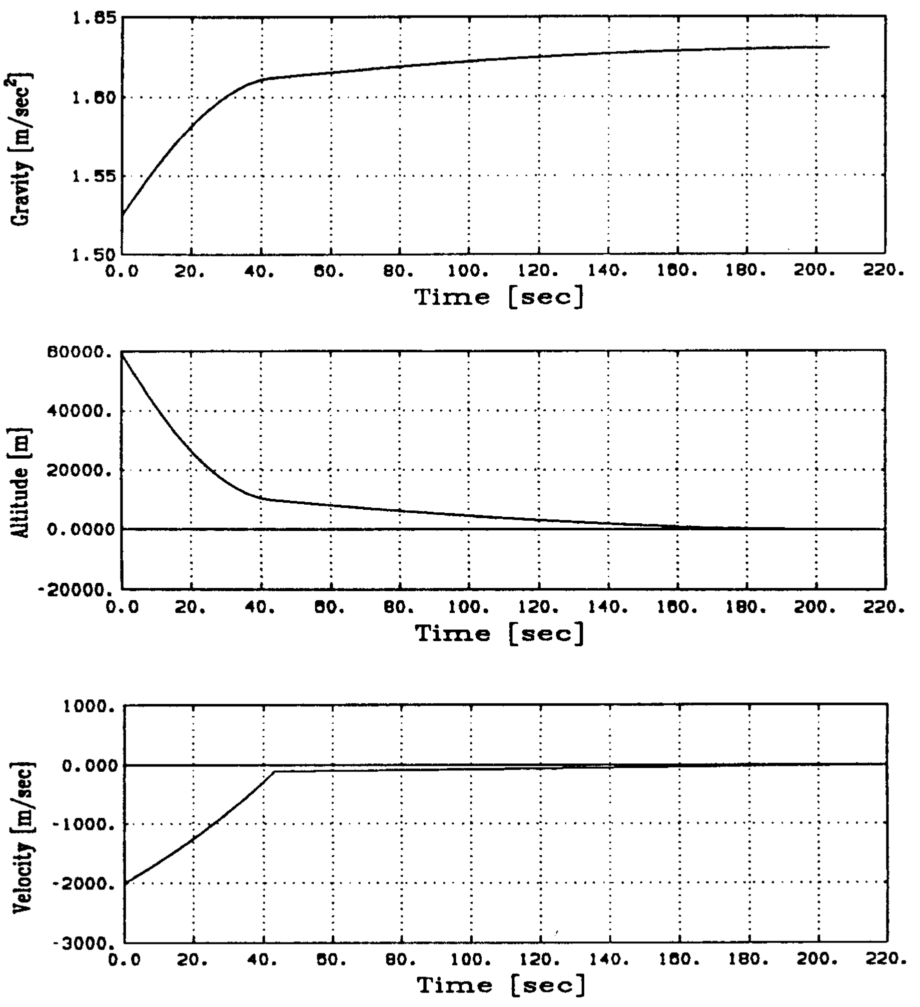

Figure 4.24 Simulation Waveforms for Lunar Landing Maneuver (cont.)

CHAPTER 5

CONCLUSIONS AND REFINEMENTS

The software development of an interface for a compiler designed for an object-oriented language involves design, implementation and validation. An interface to generate ACSL simulation programs for continuous system analysis is implemented for DYMOLA, a compiler for a modeling language. The interface eliminates the need for manual coding in ACSL. The design and implementation of the compiler's code generator proceeds through several stages during the development cycle.

The basic requirements and the detailed specifications lead to the detailed design phase involving algorithm development and implementation. An ACSL model described in terms of time dependent non-linear differential equations or transfer functions can be generated using DYMOLA. An experiment description for run-time analysis and additional model statements can be specified using an auxiliary file, referred to as the *control* file. Modularization is performed with variable name manipulation, formation of an ACSL simulation program, control file compilation and integration of program hierarchy. The implemenation uses variable type detection, operand detection, several infix techniques and complex equation manipulations. The parsing and error checking performed in accordance with the detailed specifications include complex procedures for scanning, system dependent file handling and extensive string manipulations.

The ACSL interface validation is performed with selected examples in several application areas. This is an exhaustive iterative process involving possible software modifications and/or further developments, followed by further validations. This process is based upon comparisons between the requirement definitions and the code generator's results, as well as efforts to make the interface as user-friendly as possible. Several examples use bond graphs as modeling elements to formulate model descriptions of systems. The resulting ACSL programs are compiled and the results of the simulation studies are presented with time trajectories of the simulation variables.

5.1 Refinements

The DYMOLA compiler already sorts all the equations, and thus ACSL's equation sorting can be eliminated altogether by the implementation of a PROCEDURAL block within the DERIVATIVE section of an ACSL program. The code within a PROCEDURAL block is not sorted by an ACSL translator and is executed in the existing sequence. Compilation of an ACSL simulation program is thus made faster. This feature has been implemented in DYMOLA's ACSL interface.

E.g., the DYMOLA statement:

```
L*der(f) = e
```

was translated by the ACSL interface into:

```
f = INTEG(e/L, f0)
```

Now, the two statements formed are:

```
df = e/L
f = INTEG(df, f0)
```

and, since DYMOLA already sorts all the equations, the resulting ACSL code is:

```
DERIVATIVE
  PROCEDURAL
    ...
    df = e/L
    ...
  END $ "of PROCEDURAL"
  ...
  f = INTEG(df, f0)
  ...
END $ "of DERIVATIVE"
```

All the INTEG statements are placed outside the PROCEDURAL section. Any SCHEDULE statements from the *derivative* section in a control file are also placed outside the PROCEDURAL section.

The TERMT statement within a control file is made optional if there already exists a *maxtime* statement. A TERMT statement is automatically generated within the DYNAMIC section of an ACSL program from the *maxtime* statement. This feature has been implemented in the ACSL interface.

E.g., the *maxtime* statement in the control file:

```
maxtime tmax = 2E-5
```

results in a TERMT statement within an ACSL program:

```
     . . .
     TERMT (t .GE. tmax)
     . . .
END $ "of DYNAMIC"
```

The incorporation of already present TERMT statements in the control file into an ACSL program is not affected, and is in accordance with the detailed specifications as described earlier.

APPENDIX A

DYMOLA, THE MODELING LANGUAGE

A.1 General Principles of Macro Facility

The macro facility is used to generate subsystem descriptions in most CSSLs. At first glance, the macro facility in simulation programs appears to resemble the subprogram approach in general purpose high-level programming languages. Macro expansion performed by the built-in macro handler in a simulation language generates equations which are sorted into an executable sequence. The macro facility for a language such as DESIRE, which does not provide for an equation sorter, is ineffective. Equation sorting involves a complete rearrangement of equations from the various macros indicating that macro replacement must be performed before the sorting process. As a result, macro compilation is not independent and macros must be included as part of the model source code. This is a drawback of the macro concept. Thus, the macro facility is distinctly different from subprograms in high-level programming languages.

In practice, macros provide a mechanism to successfully partition model structures, but do not allow hierarchical decomposition of the data structures. The formal parameter list in macro headers must be complete, including even the constants among the input parameters of the macro, to simulate with different values for these parameters and thus increase the macro's usability. Macro calls may become unmanageable with parameter values passed between each hierarchical level. This is yet another disadvantage of the macro concept.

Depending on the environment in which a model is used, several macro representations of the same element may be needed. Most CSSL-type modeling languages can sort equations, but obviously this is not sufficient. A modeling software must be able to solve equations for any variable. It is not essential for a simulation language to provide for a macro handler of its own. The shortcomings of the macro facility introduces an alternative utility that replaces the macro handler altogether.

A.2 DYMOLA

DYMOLA is a general purpose hierarchical modular modeling language for continuous-time systems. The language has been developed for the purpose of modeling hierarchically structured systems. The tool can also be used to model complex physical systems through hierarchical bond graphs, which can include arbitrary non-linearities. The language can be used to implement these hierarchical non-linear bond graphs.

The DYMOLA program is intended to support the user in coding more readable and better modularized hierarchically structural model descriptions. The user can write an equation description of the problem using block diagrams for instance, or utilize the non-linear hierarchical bond graph abstraction to describe the system under investigation. The bond graph modeling technique is extensively used in several application areas.

DYMOLA is a modeling software that can solve equations for arbitrary variables. It is a totally independent general purpose and powerful macro handler that can be called as a separate program before a simulation engine is used. It replaces the macro facility and its drawbacks. It is a program generator that can be used as a front end to several different simulation languages. DYMOLA currently supports the simulation languages DESIRE, SIMNON and ACSL. It also supports the general purpose high-level programming language FORTRAN. A compiler switch decides for what target language code is to be generated.

There are two different versions of DYMOLA that exist at the present time. The Simula version runs on UNIVAC computers. The Pascal version runs on VAX/VMS and also on PC/MS-DOS (Turbo Pascal Version 4.0 or higher). A UNIX version was generated by using the public domain p2c compiler that translates any PASCAL program into an equivalent C program.

A.3 Language Elements of DYMOLA

A.3.1 Variables

DYMOLA variables are classified into two different types, *terminal* and *local*. Variables that may change their values during a simulation run are either locals or terminals. In DYMOLA, all variables must be declared.

A.3.2 Terminals

Variables that are to be assigned using the dot notation or connected to another variable outside the model must be declared as terminal variables. Terminal variables are declared using *terminal*. Terminals can have default values. They are assigned these values using a *default* statement. In this way, they don't need to be externally connected, but they can be. A comma must separate each variable in a terminal statement; e.g.,

```
terminal ua, uf

default ua = 25.0, uf = 25.0
```

Terminal variables can be either inputs or outputs. The user can explicitly declare them as *input* or *output* rather than simply as *terminal*. Inputs and outputs are special types of terminals. Default values (or initial conditions) can be assigned to inputs and outputs. Negative default values are allowed; e.g.,

```
input ua = 25.0, uf = 25.0
output omega = −2.0
```

Variables declared as terminals (or in cuts) are undirected variables. e.g., the statement V = $V_b - V_a$ can be rearranged by the compiler into $V_b = V + V_a$ or $V_a = V_b - V$ if needed. If V had been declared as an output, the compiler would be prevented from rearranging this equation.

A.3.3 Locals

Variables that are totally connected inside the model are declared as local variables. Local variables are declared using *local*. Locals can have default values. They are assigned these values using a *default* statement. A comma must separate each variable in a local statement; e.g.,

```
local ia, Twist

default ia = 5.0, Twist = 5.0
```

A.3.4 Constants

Constants are variables that obtain a constant value upon declaration. They are declared with the keyword *constant*. Constants are not reassigned after the declaration assignment. Negative constant values are allowed; e.g.,

```
constant Temp = 300.0
```

A.3.5 Parameters

Parameters are similar to constants. Their values do not change during a simulation run. They can be reassigned, but only between simulation runs. This enables the compiler to extract all parameter computations from the dynamic loop. Parameters are one mechanism for data exchange between models. In this respect, they are similar to formal read-only arguments of a subprogram call in a traditional programming language.

DYMOLA constants can be declared to be of type *parameter*. Values are assigned to parameters from outside the model. Parameters can have default values in which case it is not necessary to assign a value to them from outside the model. The default parameter values can be overridden from outside the model.

Parameters cannot be reassigned within the model in which they are declared as parameters, only within the calling program.

A comma must separate the different parameters. Negative default values are allowed; e.g.,

```
parameter Ra, Rf
parameter La = 0.0, Lf = 0.0
```

Ra, Rf are assigned values in the call to the model in which these parameters are declared.

A.3.6 Externals and Internals

Externals are similar to parameters, but they provide for an implicit rather than an explicit data exchange mechanism. In this respect, they are similar to COMMON variables in a FORTRAN program. Externals are used to simplify the utilization of global constants or global parameters.

External variables declared in a called model are used in equations within this model. The calling program must acknowledge the existence of these globals by specifying them as *internal*. However, *internal* is not a declaration but only a provision for the redundancy, that is all internal variables must be declared as something else also, for example, as locals which must get defined in equations within the calling program.

A.3.7 Models and Submodels

The model description must begin with *model* followed by a name identifier. This name indentifies the main model; e.g.,

```
model RLC
```

A submodel description must begin with *model type* followed by a name identifier. This name identifies a model element; e.g.,

```
model type R
```

A submodel statement in the main model is used to access a model element. The submodel declaration must be followed by the model type in parenthesis. Multiple instance calls to a model element can be made in one submodel statement with distinct submodel names, which can be referenced later in the model description. Parameters can be assigned values in each call. A comma must separate each call; e.g.,

```
submodel (R) R1(R=100.0), R2(R=20.0)
```

A DYMOLA program may contain definitions for (or inclusions of) an arbitrary numbers of model types followed by exactly one model which invokes the declared model types as submodels. Initial values can be assigned to variables in a model type using an *ic* declaration in a submodel statement; e.g.,

```
submodel (I) L1(I=1.5E-3) (ic f=0.5)
```

where f is a state variable in model type I.

E.g.,

```
model type DCMOT
    terminal theta, omega, ua = 25.0, uf = 25.0, tauL, JL
    local ia, if, ui, psi, taum, Twist
    parameter Ra, Rf, Kmot, Jm
    parameter La = 0.0, Lf = 0.0, Bm = 0.0
        Lf*der(if) = uf − Rf * if
        La*der(ia) = ua − ui − Ra * ia
        psi = kmot * if
        taum = psi * ia
        ui = psi * omega
        der(Twist) = taum − tauL − Bm * omega
        Twist = (Jm + JL) * omega
        der(theta) = omega
end
```

The above model can then be called in the following way:

```
submodel (DCMOT) dcm1(Ra = 2.0, Rf = 5.5, kmot = 1.0, Jm = 15.0)
```

which can then be connected to the outside world using Pascal's dot notation (which has nothing to do with CSSL's dot notation):

```
dcm1.ua = kalph * err
dcm1.uf = 12.0
dcm1.JL = crl1.JL
dcm1.tauL = crl1.tauL
dcm1.omega = crl1.omega
```

where crl1 is a cable reel of model type CABREL with terminals JL, tauL and omega in common with DCMOT.

A.3.8 End

Each model description must terminate with an *end* statement. That is, each *model* and *model type* statement must have a corresponding *end* statement.

A.3.9 Include Operator

The @ operator instructs the DYMOLA preprocessor to include an external file at this place. It corresponds to the include statement of most programming languages. It can, for instance, be used to include the element definitions which were stored on separate files; e.g.,

```
@resistor.elec
```

A.3.10 Connect, Cuts and Nodes

Cuts are hierarchical data structures (similar to Pascal records). Wires are frequently grouped into cables or buses. DYMOLA provides for an equivalent mechanism by so-called CUTs. A cut is like a plug or socket. It defines an interface to the outside world. Variable common between model types can be placed in a cut definition. The syntax is, *cut* followed by a cut_name with the variable list in parenthesis. A comma must separate the different cuts; e.g.,

```
cut A(v1, v2), B(v3, v4)
```

Cut declarations in model types that have variables in common with other model types must be identical. The *connect* statement is used to connect cuts. The DYMOLA statement:

```
connect x:A at y:B
```

plugs the cut A of model x into the socket B of model y.

E.g., Declare cut *mech* in model types DCMOT and CABREL with three variables omega, tauL and JL in common:

```
cut mech(omega, tauL, JL)
```

and invoke in the main program a DC-motor *dcm1* of type DCMOT and a cable reel *crl1* of type CABREL. Next, connect the cut *mech* of *dcm1* at the cut *mech* of *crl1*. This is coded as follows:

```
submodel (DCMOT) dcm1(Ra =...)
submodel (CABREL) crl1(Bl = ...)
connect dcm1:mech at crl1:mech
```

The connect statement automatically generates the three model equations:

```
dcm1.omega = crl1.omega
dcm1.tauL = crl1.tauL
dcm1.JL = crl1.JL
```

Cuts can be hierarchically structured; e.g.,

```
cut mech(omega, tauL, JL)
cut elect(ua, uf)
cut both[mech, elect]
```

in which case cut *mech* and cut *elect* can be connected separately, or connected both together. During expansion of the connect statement, DYMOLA checks that the connected cuts are structurally compatible with each other.

One cut can be declared as *main* cut. The main cut is the default cut in a connection. The model name is specified to connect the main cut of a submodel. *Nodes* are a convenient means to organize connections. The node declaration is used for connections inside a model instead of across model boundaries. Nodes are named, and cuts can be connected to nodes. Nodes are hierarchically structured in the same manner as cuts are.

E.g.,

```
model M
    cut A(v1,v2), B(v3,v4)
    main cut D[A,B,C]
    . . .
end
. . .
node N
connect M at (N,N,N)
```

The connect statement is equivalent to:

```
connect M:A at N, M:B at N, M:C at N
```

which is identical to:

```
connect M:A at M:B at M:C
```

which results in the following set of equations:

```
M.v1 = M.v3
M.v3 = M.v5
```

```
M.v2 = M.v4
M.v4 = M.v6
```

DYMOLA provides for a second type of cut and connect mechanism, which is used extensively for bond graph modeling. The generalized form of this DYMOLA cut looks as follows:

```
cut cut_name(across_variables/through_variables)
```

DYMOLA supports the concept of *across* and *through* variables. Across variables around a node assume the same value whereas through variables into a node add up to zero. DYMOLA's nodes can be used as 0-junctions in a bond graph model. In bond graph terminology, across variables are called *effort* variables, while through variables are called *flow* variables. In a 1-junction of a bond graph, all flow variables are equal while all effort variables add up to zero. There is no DYMOLA equivalent for 1-junctions, but 1-junctions are the same as 0-junctions with the effort and flow variables interchanged.

In an electrical circuit, the potentials around a node must be equal, whereas the currents into a node add up to zero. Variables of type potential are called across variables, while variables of type current are called through variables. If three models m1, m2 and m3 have each a cut of type A declared as:

```
cut A(V/I)
```

and the connect statement used is:

```
connect m1:A at m2:A at m3:A
```

then, the following model equations are generated:

```
m1.v = m2.v
m2.v = m3.v
m1.i + m2.i + m3.i = 0.0
```

Thus, all the across variables to the left of the slash operator are set equal, and all the through variables to the right of the slash operator are summed up to zero. The DYMOLA preprocessor automatically generates the necessary coupling equations. Note that currents at cuts are normalized to point into the subsystem. If a current is directed in the opposite way, it must take a minus on the cut definition.

In a mechanical system, all forces and torques are across variables, whereas all positions, velocities and accelerations are flow variables. In a hydraulic system, water levels are across variables, while water flow is a through variable. In a thermic system, temperature and pressure are across variables, while heat flow is a through variable, etc.

The DYMOLA model library describing the basic bond graphs that can be used as modeling elements to formulate a bond graph description of a system is shown in Figure A.1.

```
model type bond
   cut A(x/y), B(y/−x)
   main cut C[A,B]
   main path P<A−B>
end
model type SE
   main cut A(e/.)
   terminal E0
   E0 = e
end
model type SF
   main cut A(./−f)
   terminal F0
   F0 = f
end
model type R
   main cut A(e/f)
   parameter R=1.0
   R*f = e
end
model type C
   main cut A(e/f)
   parameter C=1.0
   C*der(e) = f
end
model type I
   main cut A(e/f)
   parameter I=1.0
   I*der(f) = e
end
model type TF
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter m=1.0
   e1 = m*e2
   f2 = m*f1
end
model type GY
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter r=1.0
   e1 = r*f2
   e2 = r*f1
end
```

Figure A.1 DYMOLA Bond Graphs

The model type *bond* simply exchanges the effort and flow variables. 0-junctions and 1-junctions always toggle in any bond graph model. Neighboring junctions can both be described by regular DYMOLA nodes if they are connected with a bond. In order to avoid maintaining different types of R, C, L, TF and GY elements, all elements (except for the bonds) in DYMOLA must be attached to 0-junctions only. If they need to be connected to 1-junction, a bond must be placed inbetween.

A.3.11 Paths

A *path* is used to connect a variable through from a source to a destination. DYMOLA allows the user to declare a directed path from an input cut to an output cut.

E.g., A model used to describe a pump which is declared as follows:

```
model pump
   cut inwater(w1), outwater(w2)
   path water <inwater – outwater>
   . . .
end
```

Two more models describing a pipe and a tank with compatibly declared cuts and paths exist. Connect the water flow from the pump through the pipe to the tank with the statement:

```
connect (water) pump to pipe to tank
```

One path can always be declared as the main path. If the main path is to be connected, the path name can be omitted from the connect statement; e.g.,

```
model pump
   cut inwater(w1), outwater(w2)
   main path water <inwater – outwater>
   . . .
```

```
end
```

The connect statement can now be:

```
connect pump to pipe to tank
```

A.3.12 Connect Operators

Connection mechanisms in DYMOLA are possible with operators. The *at* operator is used to connect common variables in model types defined in a cut definition. It can be abbreviated with the "=" symbol. The *to* operator is used to connect paths and it denotes a series connection. It can be abbreviated with the "–" symbol. The *reversed* operator is used to connect a path in the opposite direction. It can be abbreviated with the "\" symbol. The *par* operator is used to parallel connect two paths. It can be abbreviated with the "//" symbol. The *loop* operator is used to connect paths in a loop.

A.3.13 Continuation

The "—>" symbol denotes continuation lines in DYMOLA; e.g.,

```
connect —>
        R1 from N1 to N2, —>
        R2 from N2 to N0, —>
        . . .
```

A.3.14 Comment

Comments may be placed in a DYMOLA program by enclosing them within left and right curly braces; e.g.,

```
{ Bond graph model of a Hydraulic System }
```

A.3.15 Derivative Operators

DYMOLA uses the dot notation. Derivatives are either expressed using the der (.) operator or a prime ($'$). It is also allowed to use a der2 (.) operator or a double prime ($''$) to denote a second derivative. Even higher derivatives are admissible; e.g.,

$$\mathrm{Lf} * \mathrm{der}(\mathrm{if}) = \mathrm{uf} - \mathrm{Rf} * \mathrm{if}$$

Contrary to most CSSLs, DYMOLA allows these operators to be used anywhere in the equation, both to the left and to the right of the equal sign. Consequently, it is not possible to set initial conditions for the integrators inside a model, which is clearly a disadvantage of DYMOLA.

A.3.16 IF Statement

In order to sort all equations properly, DYMOLA provides an *if* statement of the form:

<var> = if <cond> then <expr> else <expr>

E.g., tz = if IRB = 0.0 then 0.0 else tan(z)

A.3.17 Equation Syntax and Manipulation

DYMOLA uses the syntax *expression* = *expression*. It can solve equations for any variable which appears linearly in the equation. This doest not mean that the equation as a whole must be linear.

E.g., DYMOLA is able to handle the equation:

$$7 * \mathrm{x} + \mathrm{y} * \mathrm{y} - 3 * \mathrm{x} * \mathrm{y} = 25$$

if the variable it wants to solve this equation for is x.

In this case, DYMOLA will transform the above equation into:

$$x = (25 - y * y)/(7 - 3 * y)$$

However, it cannot solve the original equation for the variable y.

During the process of model expansion, equations are solved for the appropriate variable. For this reason, the Single Assignment Rule (SAL) no longer applies. It is perfectly acceptable to have der(Twist) on the left hand side of one equation, and Twist on the left hand side of another; e.g.,

```
der(Twist) = taum − tauL − Bm ∗ omega
Twist = (Jm + JL) ∗ omega
```

Terms which are multiplied by a zero parameter are automatically eliminated during the model expansion. E.g., in the model equation:

```
La ∗ der(ia) = ua − ui − Ra ∗ ia
```

if La = 0.0, the equation is first replaced with the model equation:

```
0.0 = ua − ui − Ra ∗ ia
```

which then results in one of three simulation equations:

```
(1) ua = ui + Ra ∗ ia
(2) ui = ua − Ra ∗ ia
(3) ia = (ua − ui)/Ra
```

depending on the environment in which the model is used. However, if La is not equal to 0.0, the model equation is transformed into the simulation equation:

```
der(ia) = (ua − ui − Ra∗ia)/La
```

This is a very elegant way to solve the “variant” macro problem of ACSL. Thus, parameters with value 0.0 are treated in a completely different manner. Parameters that are not set equal to zero are preserved in the generated simulation code, and can be interactively altered through the simulation program directly without a need to return to DYMOLA. Parameters with value 0.0 are optimized away by the DYMOLA compiler, and are not represented in the simulation code. However, the advantages of this decision are overwhelming, since this does away with an entire class of structural singularities.

Code Optimization – DYMOLA provides for a feature to eliminate trivial equations of the type a = b, by eliminating one of the two variables, and replacing other occurrences of this variable in the program by the retained variable. This can significantly speed up the execution of the simulation program.

Linear Algebraic Loops – DYMOLA recognizes from an initial set of equations those equations that contain a der(.) operator which must be solved for the derivatives. It moves those variables from the list of unknowns to the list of knowns. It further recognizes any variables that appear only in one of the remaining equations each and moves those variables from the list of unknowns to the list of knowns. Next, trivial equations of the type a=b are eliminated by discarding one of the two variables, and replacing other occurrences of this variable in the program by the retained variable. A set of solved equations is obtained along with a set of remaining equations. If each remaining equation contains at least two unknowns, and each of the unknowns appears in at least two equations, an algebraic loop results. If all the unknowns appear linearly in all the remaining equations, DYMOLA can rewrite the system of equations by finding the determinant of the linear equation system, and explicitly expressing the solution using formula manipulation.

A.3.18 Structural Singularities

Usually, each component of a system that can store energy is represented by one or more differential equations. Capacitors and inductances of electrical networks can store energy. Each capacitor and each inductor normally gives cause to one first order differential equation. Mechanical masses can store two forms of energy, potential and kinetic energy. Each separately movable mass in a mechanical system usually gives rise to one second order differential equation which is equivalent to two first order differential equations. However sometimes, this is not so. If we take two capacitors and connect them in parallel, the resulting system order is still one. This is due to the fact that there exists a linear relationship between the two voltages over the two capacitors (they are the same), and thus, they both do not qualify for state variables.

Such situations are called *system degeneracies* or *structural singularities*. DYMOLA is able to handle structural singularities with the *differentiate* command. Usually, subsystems will be designed such that no such singularities occur. The two parallel capacitors are simply represented in the model by one equivalent capacitor with the value:

$$\mathrm{Ceq} = \mathrm{C1} + \mathrm{C2}$$

However, when subsystems are connected together, structural singularity is a direct result of the coupling of the two subsystems. Let us assume the two subsystem orders of subsystems S_1 and S_2 are n_1 and n_2. If the coupled system S_c has a system order n_c which is smaller than the sum of n_1 and n_2, a structural singularity exists which is a result of the subsystem coupling.

A.4 DYMOLA Commands

At the operating system prompt ($), the DYMOLA preprocessor can be entered in an interactive mode with the following command:

```
$ dymola
>
```

DYMOLA responds with its own prompt (>) requesting the user to enter further commands. The set of commands available in DYMOLA and some of the mechanisms involved behind command execution are described.

A.4.1 Enter Model

The command to specify a model to be compiled at the DYMOLA prompt is:

```
> enter model
  − @model_file.dym
>
```

The *enter model* statement instructs DYMOLA to read in a model. DYMOLA responds with the next level prompt (−). The user is requested to enter a complete model specification. Equations can be entered here, but it is more practical to invoke them indirectly (@). DYMOLA returns to its first level interactive prompt (>) after it finds the model definition. The model filename and its extension are user chosen.

At this point, DYMOLA expands the set of equations by the coupling equations. The resulting equations can be observed, for example, with the *output equations* command.

A.4.2 Enter Experiment

The command to specify an experiment description for the model is:

```
> enter experiment
  – @control_file.ctl
>
```

The *enter experiment* statement instructs DYMOLA to read in an experiment description. DYMOLA responds with the next level prompt (–). The user is requested to enter a complete experiment description indirectly (@). The experiment filename and its extension are user chosen.

The experiment description for the model is specific for each of the target languages. DYMOLA's code generator portion can be used to generate a simulation model for DESIRE, SIMNON, ACSL or FORTRAN.

A.4.3 Partition

The command to determine what variable to compute from each equation is:

```
> partition
```

The partition algorithm assigns the causalities and sorts the equations into an executable sequence. The command invokes the partition algorithm, but does not display the results to the user. The resulting equations can be observed, for example, with the *output sorted equations* command.

A.4.4 Partition Eliminate

The command to partition the equations and automatically perform all types of elimination algorithms repetitively until the equations no longer change is:

> partition eliminate

The sorted equations contain variables enclosed in "[ ]" which must be solved for each equation. This set of equations contains many trivial equations of the type a = b. DYMOLA invokes all types of elimination algorithms to result in a much reduced set of equivalent equations. The resulting equations can be observed, for example, with the *output sorted equations* command.

A.4.5 Eliminate Equations

The command to eliminate trivial equations of type a = b, and replace all occurrences of the variable a in all other equations by the variable b is:

> eliminate equations

The set of equations observed by the *output sorted equations* command contains many aliases, i.e., the same physical quantity is stored several times under different variable names. This slows down the execution of the simulation program. This command will rid equations of such type. There is one exception to the rule: the eliminate operation will never eliminate a variable that was declared as either input or output. If, in an equation of the type a = b, a is an output variable, it will throw the equation away as well, but in this case, all occurrences of b are replaced by a. If both a and b are declared as output variables, the equation will not be eliminated at all.

This algorithm can be applied either to the original equations or to the partitioned equations, and it will work equally well in both cases. In reality, this algorithm reduces all equations of the types:

$$\pm a = \pm b$$

and:

$$\pm a \pm b = 0$$

which are variants of the previously discussed case. This algorithm works also if either a or b is a constant.

A.4.6 Eliminate Parameters

The command to apply an elimination algorithm with respect to parameters is:

```
> eliminate parameters
```

The elimination algorithm that is now executed will perform the following tasks:

1. All parameters with a numerical value of 0.0 or 1.0 are eliminated from the model, and the numerical value is replaced directly into the equations.
2. A numerical value of 1.0 that multiplies a term is eliminated from that term.
3. A term that is multiplied by a numnerical value of 0.0 is replaced as a whole by 0.0.
4. Additive terms of 0.0 are eliminated as a whole.
5. If, in an equation, an expression consists of parameters and constants only, a new equation is generated that will evaluate this expression (assigned to a new generic variable), and the occurrence of the expression in the equation is replaced by the new generic variable.
6. If an equation contains only one variable, it must be solved for that variable. This variable is then automatically redeclared as a parameter, and the equation is marked as a parameter equation which can be moved from the DYNAMIC portion of the simulation program into the INITIAL portion of the simulation program.

A model with parameters whose values are not 1.0 or 0.0 is not affected by this algorithm. This algorithm may have undesired effects. It is often desired to start with a simple model (by setting some parameters equal to 0.0) and then successively make the model more realistic by assigning in the simulation program, true values to the previously defaulted parameters. In this case, *eliminate parameters* must not be used, since this algorithm will cause these parameters to no longer appear in the simulation program.

A.4.7 Eliminate Variables

The command to affect algebraic loops is:

```
> eliminate variables
```

This command must be used only after the equations have been partitioned. It affects only algebraic loops and it affects each algebraic loop separately. DYMOLA counts the times each loop is referenced in an algebraic loop. Obviously, each loop variable must occur atleast twice, otherwise, it would not be a loop variable.

DYMOLA will investigate all loop variables that occur exactly twice in a loop. If it finds such a variable, and if this variable appears linearly in atleast one of the two equations, DYMOLA will solve the equation for that variable, and replace the other occurrence of the variable by the evaluated expression, thereby eliminating this variable altogether from the loop. If the eliminated variable is referenced anywhere after the loop, the equation defining this variable is not thrown away, but taken out of the loop and placed immediately after the loop. The same is true if the eliminated variable has been declared as an output variable. Although the same algorithm could be applied to variables that occur more than twice, this is not done since it tends to explode the code (the same, possibly long, expressions would have to be duplicated several times).

A.4.8 Eliminate Outputs

The command to take variables that appear only once in the set of equations out of the state-space model is:

```
> eliminate variables
```

DYMOLA checks for all variables in the DYNAMIC section of the code that appear only once in the set of equations. Obviously, the equations containing these variables must be used to evaluate them. Since these equations will not otherwise influence the behavior of the dynamic model, they can be marked as output equations, and can be taken out of the state-space model.

A.4.9 Differentiate

The command to handle structural singularities in DYMOLA is:

```
> differentiate
```

When subsystems are connected together, structural singularity is a direct result of the coupling of the two subsystems. This command takes care of this situation.

A.4.10 Output Equations

The command to observe the set of equations after DYMOLA has compiled the model and expanded the set of equations by the coupling equations is:

```
> output equations
```

Model compilation, as a result of the *enter model* command, replaces all submodel references by their model definitions. It generates the additional equations that are a result of the submodel couplings i.e., DYMOLA replaces the connect statements by the coupling equations. The results of this text replacement are observed by issuing the *output equations* command.

A.4.11 Output Sorted Equations

The command to observe the sorted and marked (but not yet solved) equations as a result of the partition algorithm, after it has determined which equation needs to be solved for what variable, is:

```
> output sorted equations
```

The variables enclosed in "[ ]" are the variables for which each equation must be solved. The resulting set of equations contains many trivial equations of the type a = b, which can be eliminated with other commands.

A.4.12 Output Solved Equations

The command to perform the actual symbolic manipulation on the equations after partitioning, eliminating and sorting is:

```
> output solved equations
```

The command produces a listing of solved variables with references to the corresponding model and its submodels. The experiment description can be added to the model after this command.

A.4.13 Output Variables <target_language>

The command to list a complete set of variables and related information on each variable in the model description is:

```
> output variables <target_language>
```

The <target_language> can be DESIRE, SIMNON, ACSL or FORTRAN. The resulting variable list includes the type (terminal, parameter, input, output, state, derivative, constant, etc.) of each variable and the value of each known variable. The list also includes a cross-reference between old and new names for the specified simulation language, i.e., before and after variable name changes. The original DYMOLA variable names in the model and target languuage specific variable names after conversion to conform with required variable name length, etc., are specified.

A.4.14 Output <target_language> Model

The command to generate a simulation *model* for a target language is:

```
> output <target_language> model
```

DYMOLA supports DESIRE, SIMNON, ACSL and FORTRAN as possible target languages for a simulation *model*. An experiment description is not needed to generate a model for any of these languages. A compiler switch determines code to be generated for the specified simulation or high-level programming language; e.g.,

```
> output acsl model
```

generates an ACSL simulation model.

A.4.15 Output <target_language> Program

The command to generate a simulation *program* for a target language is:

```
> output <target_language> program
```

DYMOLA supports DESIRE and ACSL as possible target languages for a simulation *program*. An experiment description in a control file is needed to generate a program for DESIRE or ACSL, and it is compiled with this command; e.g.,

```
> output acsl program
```

compiles the control file and generates an ACSL simulation program.

A.4.16 Outfile <filename.ext>

The command to redirect output to a file (e.g. for printout) is:

```
> outfile <filename.ext>
```

This command must be issued prior to the command whose output is to be redirected to a file. The output filename and its extension are user chosen; e.g.,

```
> outfile rlc.eq
> output equations
```

where the *outfile* statement specifies the output file name and the *output equations* statement writes the generated equations to the output file.

A.4.17 Stop

The command to exit from DYMOLA is:

```
> stop
$
```

This command returns the user to the operating system prompt ($).

A.5 Limitations and Further Developments in DYMOLA

A.5.1 Non-Linear Equations

For some simple cases, it would be very easy to implement the appropriate transformation rules to handle even non-linear equations, but most non-linear equations don't provide non-unique solutions. E.g., the problem:

$$x * x + y * y = 1$$

when solved for y has two solutions:

$$y = +\mathrm{sqrt}(1 - x * x)$$
$$y = -\mathrm{sqrt}(1 - x * x)$$

DYMOLA would have no way to know which of the two solutions to use. The same is true when the non-linear equation is solved numerically by automatically generating an IMPL block around the equation. The numerical algorithm will simply approach one of the two solutions, often depending on the chosen initial value, and that may be the wrong one.

There is no general answer to the automated solution of non-linear equations. The best that can probably be achieved is that DYMOLA preprocessor stops when it comes across a non-linear equation, and requests help from the user. It may then store this information away for later reuse in another compilation of the same model. One possible answer that the user may provide is to request the system to build an IMPL block around the equation, and tell it which initial value to use for the iteration.

A.5.2 Non-Linear Algebraic Loops

The problem with non-linear algebraic loops is exactly the same as with the solutions of non-linear equations. Depending on how the the set of equations is iterated, it is possible to end up with one solution, or another, or none at all. There is unfortunately no way how this problem can be solved once and for all. The best that DYMOLA may be able to do is interrupt the compilation, print the set of coupled algebraic equations on the screen together with the set of unknowns contained in these equations, and ask for help. Proper help may not always be easy to provide.

A.5.3 Further Extensions

A number of possible extensions in order for DYMOLA to handle certain problems in a completely automated manner are:

1. DYMOLA should be able to eliminate variables not only from equations of type a = b, but also from equations of type a + b = 0.
2. DYMOLA should be able to recognize equations that have been specified twice, and eliminate the duplicate automatically.
3. DYMOLA should be able to handle superfluous connections, i.e., if we specify w2 = −w1, it is obviously true that also theta2 = −theta1 (except for an integration constant). However, DYMOLA won't let us specify this additional connection at the current time. Superfluous connections should simply be eliminated during the model expansion.

Several of the DYMOLA features discussed here are presented in Appendix B with DYMOLA examples. The examples include DYMOLA programs and commands, resulting outputs in solving the model equations and plots obtained with ACSL simulation.

APPENDIX B

DYMOLA EXAMPLE PROGRAMS

The use of DYMOLA is illustrated in the following example programs. DYMOLA is used for electrical circuit modeling in the first example. The second example illustrates bond graph modeling in DYMOLA. Results from DYMOLA commands to solve the model equations are presented. DYMOLA is used to generate ACSL simulation programs and resulting plots of waveforms are included.

B.1 RLC Network

This example illustrates a simple RLC circuit described with a nonlinear model. The electrical circuit diagram of the RLC network is shown in Figure B.1.

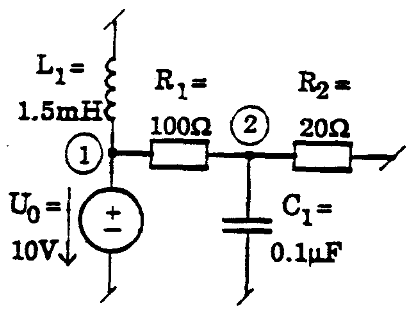

Figure B.1 RLC Network

The modeling elements used to describe the model equations of the RLC network are shown in Figure B.2.

```
model type vsource
   cut A(Va/I), B(Vb/−I)
   main cut C[A,B]
   main path P<A−B>
   terminal V=0.0
   V = Vb − Va
end

model type Common
   cut A(V/.) B(./.)
   main cut C[A]
   main path P<A−B>
   V = 0.0
end

model type resistor
   cut A(Va/I), B(Vb/−I)
   main cut C[A,B]
   main path P<A−B>
   parameter R=1.0
   local V
   V = Va − Vb
   R∗I = V
end

model type capacitor
   cut A(Va/I), B(Vb/−I)
   main cut C[A,B]
   main path P<A−B>
   parameter C=1.0
   local V
   V = Va − Vb
   C∗der(V) = I
end

model type inductor
   cut A(Va/I), B(Vb/−I)
   main cut C[A,B]
   main path P<A−B>
   parameter L=1.0
   local V
   V = Va − Vb
   L∗der(I) = V
end
```

Figure B.2 DYMOLA Modeling Elements

The following DYMOLA program describes the RLC network using the electrical component models.

```
@common.ele
@vsource.ele
@resistor.ele
@capacit.ele
@inductor.ele

model RLC

  submodel (vsource)   U0
  submodel (resistor)  R1(R=100.0), R2(R=20.0)
  submodel (inductor)  L1(L=1.5E-3) (ic I = 0.5)
  submodel (capacitor) C1(C=0.1E-6)
  submodel Common

  input  u
  output y1, y2

  connect Common - U0 - ((R1 - (C1//R2))//L1) - Common

  U0.Va  = 0.0
  U0.V   = u
  y1     = C1.V
  y2     = R2.I

end
```

Figure B.3 DYMOLA Program for RLC Circuit

Let us assume the above DYMOLA program is stored in a file called “rlc.dym”. The @ operator includes the element definitions stored in separate files. The program describes the RLC circuit using series and parallel connections of elements. Main path and cut declarations in the element models are needed for these connections. The DYMOLA preprocessor is invoked and the model to be compiled is specified with the following command sequence:

```
$ dymola
> enter model
− @rlc.dym
> outfile rlc.eq
> output equations
```

The enter model statement expands the set of equations in the model by the coupling equations. The generated equations redirected to the output file are shown in the next code segment.

```
Common              V = 0
U0                  V = Vb - Va
R1                  V = Va - Vb
                    R*I = V
C1                  V = Va - Vb
                    C*derV = I
L1                  V = Va - Vb
                    L*derI = V
R2                  V = Va - Vb
                    R*I = V
RLC                 U0.Va = 0
                    U0.V = u
                    y1 = C1.V
                    y2 = R2.I
                    R2.Vb = C1.Vb
                    L1.Vb = R2.Vb
                    Common.V = L1.Vb
                    C1.Va = R1.Vb
                    R2.Va = C1.Va
                    C1.I + R2.I = R1.I
                    R1.Va = U0.Vb
                    L1.Va = R1.Va
                    R1.I + L1.I = U0.I
```

Figure B.4 Output Equations after Model Compilation

The partition algorithm to determine what variable to compute from each equation is executed with the following DYMOLA commands:

```
>partition
>outfile rlc.sr1
>output sorted equations
```

which results in the following set of equations:

```
Common              [V] = 0
RLC                 [U0.Va] = 0
                    [U0.V] = u
U0                  V = [Vb] - Va
RLC                 [R1.Va] = U0.Vb
                    Common.V = [L1.Vb]
                    L1.Vb = [R2.Vb]
                    R2.Vb = [C1.Vb]
C1                  V = [Va] - Vb
RLC                 C1.Va = [R1.Vb]
R1                  [V] = Va - Vb
                    R*[I] = V
RLC                 R1.I + L1.I = [U0.I]
                    [R2.Va] = C1.Va
R2                  [V] = Va - Vb
                    R*[I] = V
RLC                 [C1.I] + R2.I = R1.I
C1                  C*[derV] = I
RLC                 [L1.Va] = R1.Va
L1                  [V] = Va - Vb
                    L*[derI] = V
RLC                 [y1] = C1.V
                    [y2] = R2.I
```

Figure B.5 Sorted Equations after Partition

The variables marked with "[ ]" are the variables for which each equation must be solved. This set of equations contains many trivial equations of the type a = b. DYMOLA is capable of throwing those out. This is accomplished with the following DYMOLA commands:

```
> partition eliminate
> outfile rlc.sr2
> output sorted equations
```

which results in the following set of equations:

```
RLC                   [U0.Va] = 0
U0                    RLC.u = [Vb] - Va
Common                [C1.Vb] = 0
C1                    V = [R1.Vb] - Vb
R1                    [V] = U0.Vb - Vb
                      R*[I] = V
RLC                   R1.I + L1.I = [U0.I]
R2                    [V] = R1.Vb - C1.Vb
                      R*[RLC.y2] = V
RLC                   [C1.I] + y2 = R1.I
C1                    C*[derV] = I
L1                    [V] = U0.Vb - C1.Vb
                      L*[derI] = V
RLC                   [y1] = C1.V
```

Figure B.6 Sorted Equations after Partition Eliminate

This is a much reduced set of equivalent equations. The next step is to perform symbolic manipulation on the equations. In DYMOLA, this is done in the following way:

```
> outfile rlc.sov
> output solved equations
```

which results in the following set of equations:

```
RLC                   U0.Va = 0
U0                    Vb = RLC.u + Va
Common                C1.Vb = 0
C1                    R1.Vb = V + Vb
R1                    V = U0.Vb - Vb
                      I = V/R
RLC                   U0.I = R1.I + L1.I
R2                    V = R1.Vb - C1.Vb
                      RLC.y2 = V/R
RLC                   C1.I = R1.I - y2
C1                    derV = I/C
L1                    V = U0.Vb - C1.Vb
                      derI = V/L
RLC                   y1 = C1.V
```

Figure B.7 Solved Equations

At this point, the circuit topology has been reduced to a trivial state-space model. Now, DYMOLA's code generator can be used to generate, for example, a ACSL simulation program. The experiment to be used is shown in Figure B.8.

```
cmodel

maxtime tmax = 2E-5
TERMT (t .GE. tmax)
cinterval cint = 2E-7
input 1, u(independ,10.0)

end
```

Figure B.8 ACSL Experiment

The experiment description is specific for each target language. The version shown in Figure B.8 is for ACSL. The simulation time, communication interval, input statement and termination condition are specified.

Let us assume that the ACSL experiment is stored in a file called "rlc.act". The experiment description is added to the model and the ACSL program is generated using the following DYMOLA commands:

```
> enter experiment
– @rlc.act
> outfile rlc.csl
> output acsl program
```

The resulting ACSL program is shown in Figure B.9.

```
"---------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"---------------------------------------------------------------"

PROGRAM RLC

  INITIAL

    CONSTANT ...
      R1XR=100.0, C=0.1E-6, L=1.5E-3, ...
      R2XR=20.0
    CINTERVAL cint = 2E-7
    CONSTANT tmax = 2E-5
    CONSTANT ...
      u = 10.0

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"----------------------------------------------Submodel: RLC"
      U0XVa = 0
"----------------------------------------------Submodel: U0"
      U0XVb = u + U0XVa
"----------------------------------------------Submodel: Common"
      C1XVb = 0
"----------------------------------------------Submodel: C1"
      R1XVb = C1XV + C1XVb
"----------------------------------------------Submodel: R1"
      R1XV = U0XVb - R1XVb
      R1XI = R1XV/R1XR
"----------------------------------------------Submodel: RLC"
      U0XI = R1XI + L1XI
"----------------------------------------------Submodel: R2"
      R2XV = R1XVb - C1XVb
      R2XI = R2XV/R2XR
"----------------------------------------------Submodel: RLC"
      C1XI = R1XI - R2XI
"----------------------------------------------Submodel: C1"
      C1XV = INTEG(C1XI/C, 0)
"----------------------------------------------Submodel: L1"
      L1XV = U0XVb - C1XVb
      L1XI = INTEG(L1XV/L, 0.5)
```

Figure B.9 ACSL Program for RLC Circuit

```
"-------------------------------------------Submodel: RLC"
      y1 = C1XV
      y2 = R2XI

    END $ "of DERIVATIVE"

    TERMT (t .GE. tmax)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure B.9 ACSL Program for RLC Circuit (cont.)

The ACSL program is compiled, and the resulting waveforms for the desired output variables y1 and y2 are shown in Figure B.10.

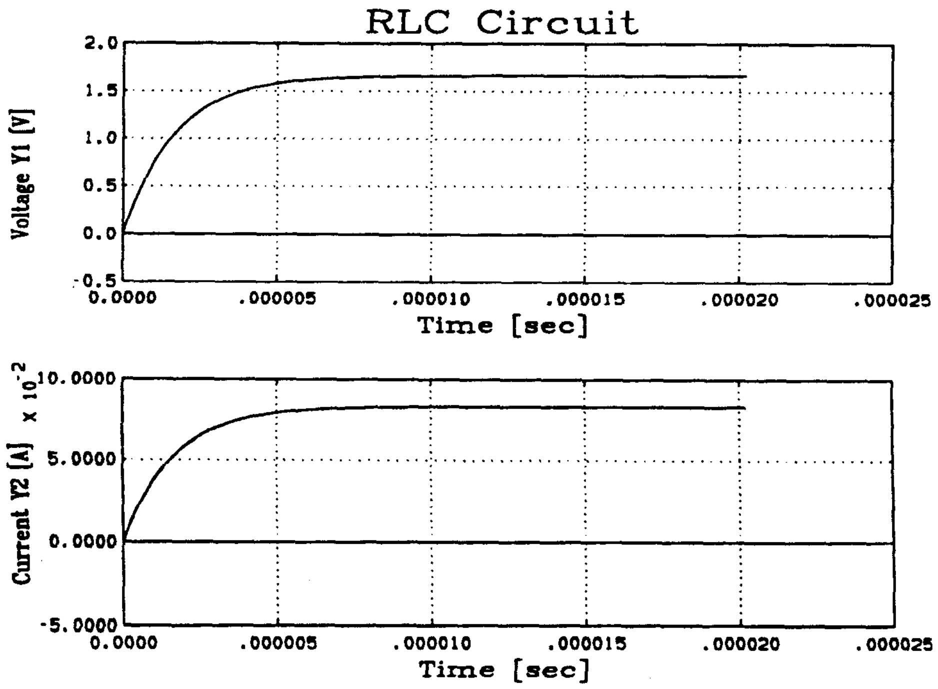

Figure B.10 Simulation Waveforms

B.2 Bond Graph Modeling of RLC Network

This example illustrates the use of DYMOLA as a bond graph modeling language. The RLC circuit of Figure B.1 is used for this example. DYMOLA bond graphs presented in Appendix A (Figure A.1), are repeated in Figure B.11 for easy reference.

```
model type bond
   cut A(x/y), B(y/−x)
   main cut C[A,B]
   main path P<A−B>
end
model type SE
   main cut A(e/.)
   terminal E0
   E0 = e
end
model type SF
   main cut A(./−f)
   terminal F0
   F0 = f
end
model type R
   main cut A(e/f)
   parameter R=1.0
   R*f = e
end
model type C
   main cut A(e/f)
   parameter C=1.0
   C*der(e) = f
end
model type I
   main cut A(e/f)
   parameter I=1.0
   I*der(f) = e
end
model type TF
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter m=1.0
   e1 = m*e2
   f2 = m*f1
end
model type GY
   cut A(e1/f1), B(e2/−f2)
   main cut C[A,B]
   main path P<A−B>
   parameter r=1.0
   e1 = r*f2
   e2 = r*f1
end
```

Figure B.11 DYMOLA Bond Graphs

The modeling elements are used to formulate a bond graph description of our simple RLC network. Figure B.12 shows the DYMOLA expanded bond graph with all elements attached to 0-junctions only.

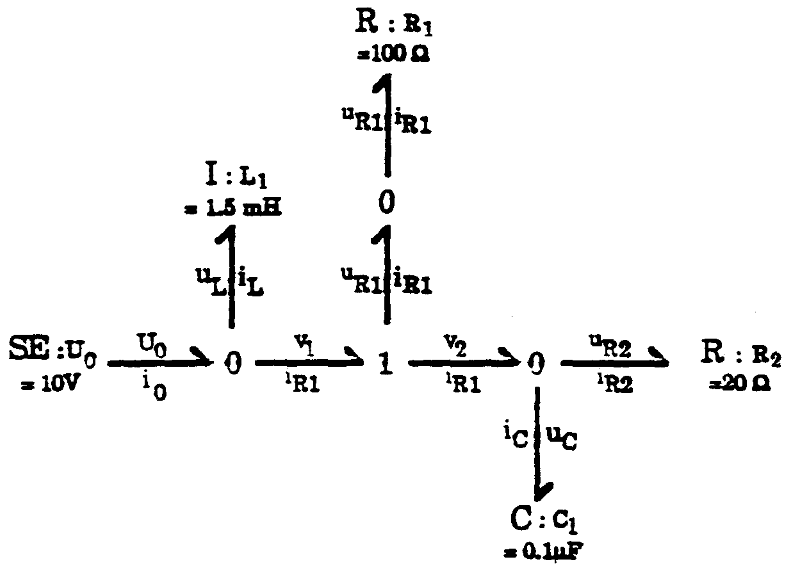

Figure B.12 DYMOLA Bond Graph of RLC Network

The causalities are not marked since DYMOLA is perfectly able to handle the causality assignment by itself. The DYMOLA program used to code the bond graph is shown in Figure B.13.

```
@bond.bnd
@se.bnd
@r.bnd
@c.bnd
@i.bnd

model RLC

  submodel (SE)   U0
  submodel (R)    R1(R=100.0), R2(R=20.0)
  submodel (I)    L1(I=1.5E-3) (ic f = 0.5)
  submodel (C)    C1(C=0.1E-6)
  submodel (bond) B1, B2, B3

  node   v1, ir1, vr1, v2
  input  u
  output y1, y2

  connect U0 at v1
  connect L1 at v1
  connect R1 at vr1
  connect R2 at v2
  connect C1 at v2
  connect B1 from v1 to ir1
  connect B2 from ir1 to v2
  connect B3 from ir1 to vr1

  U0.E0  = u
  y1     = C1.e
  y2     = R2.f

end
```

Figure B.13 DYMOLA's Bond Graph Program for RLC Circuit

Let us assume the DYMOLA program is stored in a file called "rlcb.dym". The program describes the RLC circuit using bonds. All elements, except for the bonds, are attached to 0-juntions only. If an element needs to be attached to a 1-junction, a bond is simply placed inbetween.

DYMOLA is entered and the model to be compiled is specified with the the same command sequence as in Example B.1.

```
$ dymola
> enter model
– @rlcb.dym
> outfile rlcb.eq
> output equations
```

The generated equations are shown in Figure B.14.

```
U0                  E0 = e
R1                  R*f = e
C1                  C*dere = f
L1                  I*derf = e
R2                  R*f = e
RLC                 U0.E0 = u
                    y1 = C1.e
                    y2 = R2.f
                    L1.e = B1.x
                    U0.e = L1.e
                    C1.e = B2.y
                    R2.e = C1.e
                    C1.f + R2.f = B2.x
                    B2.x = B3.x
                    B1.y = B2.x
                    B3.y + B2.y = B1.x
                    R1.e = B3.y
                    R1.f = B3.x
```

Figure B.14 Output Equations after Model Compilation

The first six equations are extracted from the models. The remaining equations are automatically generated coupling equations. The next command sequence:

```
> partition
> outfile rlcb.sr1
> output sorted equations
```

results in the following equations:

```
RLC                 [U0.E0] = u
U0                  E0 = [e]
RLC                 U0.e = [L1.e]
                    L1.e = [B1.x]
                    C1.e = [B2.y]
                    [B3.y] + B2.y = B1.x
                    [R1.e] = B3.y
R1                  R*[f] = e
RLC                 R1.f = [B3.x]
                    [B2.x] = B3.x
                    [B1.y] = B2.x
                    [R2.e] = C1.e
R2                  R*[f] = e
RLC                 [C1.f] + R2.f = B2.x
C1                  C*[dere] = f
L1                  I*[derf] = e
RLC                 [y1] = C1.e
                    [y2] = R2.f
```

Figure B.15 Sorted Equations after Partition

The elimination algorithms are applied and sorting is done with the following command sequence:

```
> partition eliminate
> outfile rlcb.sr2
> output sorted equations
```

which results in the following answer:

```
R2                  R*[RLC.y2] = C1.e
RLC                 [B3.y] + C1.e = u
R1                  R*[B3.x] = B3.y
RLC                 [C1.f] + y2 = B3.x
C1                  C*[dere] = f
L1                  I*[derf] = RLC.u
RLC                 [y1] = C1.e
```

Figure B.16 Sorted Equations after Partition Eliminate

This is a much reduced set of equivalent equations. Symbolic manipulation is performed with the commands:

```
> outfile rlcb.sov
> output solved equations
```

which results in the following answer:

```
R2                  RLC.y2 = C1.e/R
RLC                 B3.y = u - C1.e
R1                  B3.x = B3.y/R
RLC                 C1.f = B3.x - y2
C1                  dere = f/C
L1                  derf = RLC.u/I
RLC                 y1 = C1.e
```

Figure B.17 Solved Equations

The experiment description used for Example B.1, shown in Figure B.8, is also used here. It is specific for ACSL and entered as before with the command to generate an ACSL program:

```
> enter experiment
− @rlc.act
> outfile rlcb.csl
> output acsl program
```

The generated ACSL program for this example is shown in Figure B.18.

```
"--------------------------------------------------------------"
" ********* ADVANCED CONTINUOUS SIMULATION LANGUAGE ********* "
"--------------------------------------------------------------"

PROGRAM RLC

  INITIAL

    CONSTANT ...
      R1XR=100.0, C=0.1E-6, I=1.5E-3, ...
      R2XR=20.0
    CINTERVAL cint = 2E-7
    CONSTANT tmax = 2E-5
    CONSTANT ...
      u = 10.0

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

"---------------------------------------------Submodel: R2"
      R2Xf = C1Xe/R2XR
"---------------------------------------------Submodel: RLC"
      B3Xy = u - C1Xe
"---------------------------------------------Submodel: R1"
      B3Xx = B3Xy/R1XR
"---------------------------------------------Submodel: RLC"
      C1Xf = B3Xx - R2Xf
"---------------------------------------------Submodel: C1"
      C1Xe = INTEG(C1Xf/C, 0)
"---------------------------------------------Submodel: L1"
      L1Xf = INTEG(u/I, 0.5)
"---------------------------------------------Submodel: RLC"
      y1 = C1Xe
      y2 = R2Xf

    END $ "of DERIVATIVE"

    TERMT (t .GE. tmax)

  END $ "of DYNAMIC"

END $ "of PROGRAM"
```

Figure B.18 ACSL Program for RLC Circuit

The ACSL program is compiled, and the resulting waveforms for the desired output variables y1 and y2 are shown in Figure B.19.

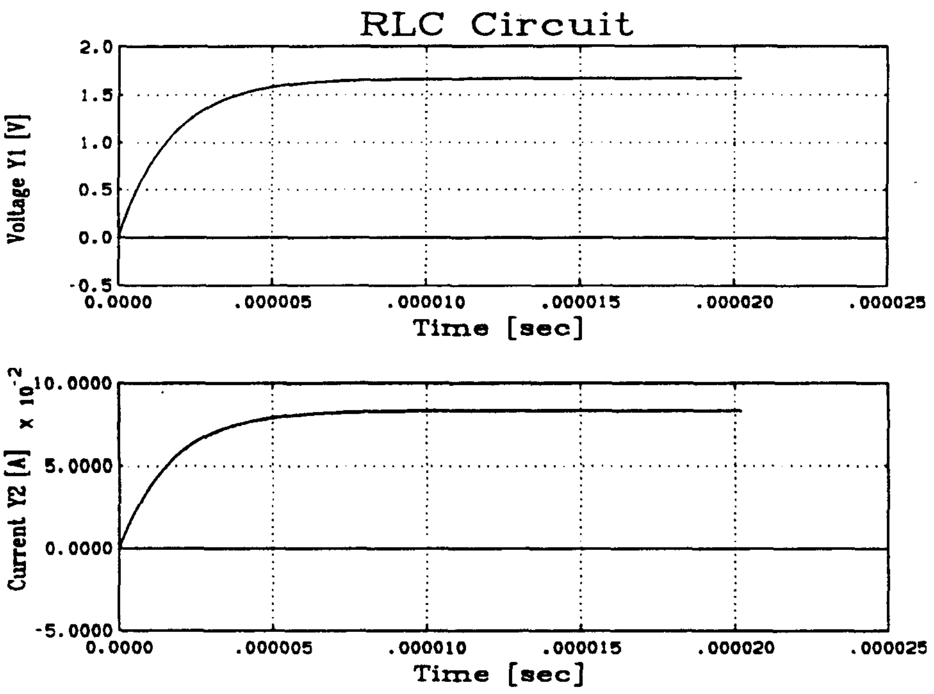

Figure B.19 Simulation Waveforms

REFERENCES

[1] Aho, A. V., Sethi, R. and Ullman, J. D. (1988). *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

[2] Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation and Compiling*, Vol. 1: *Parsing*, Prentice-Hall, Englewood Cliffs, NJ.

[3] Bobillier, P. A. (1976). *Simulation with GPSS and GPSS V*, Prentice-Hall, Englewood Cliffs, NJ.

[4] Casti, J. L. (1989). *Alternate Realities: Mathematical Models of Nature and Man*, John Wiley, New York.

[5] Cellier, F. E. (1990). "Hierarchical Non-Linear Bond Graphs – A Unified Methodology For Modeling Complex Physical Systems", in: *Proceedings ESM'90*, Nuernberg, FRG, pp. 1-13.

[6] Cellier, F. E. (1991). *Continuous-System Modeling*, Springer-Verlag, New York.

[7] Cellier, F. E. (1986). "Enhanced Run-Time Experiments for Continuous System Simulation Languages", in: *Proceedings of the 1986 SCSC Multiconference* (F. E. Cellier, ed.), SCS Publishing, San Diego, CA. pp. 78-83.

[8] Elmqvist, H. (1978). "A Structured Model Language for Large Continuous Systems", Ph.D. Dissertation, Report: CODEN: LUTFD2/(TRFT-1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[9] Elmqvist, H. (1975). SIMNON – *An Interactive Simulation Program for Non-Linear Systems – User's Manual*, Report CODEN: LUTFD2/ (TFRT-7502), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[10] Kheir, N. A., ed. (1988). *Systems Modeling and Computer Simulation*, Marcel Dekker, New York.

[11] Korn, G. A. (1989). *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

[12] Korn, G. A. and Wait, J. V. (1978). *Digital Continuous- System Simulation*, Prentice-Hall, Englewood Cliffs, NJ.

[13] Mitchell, E. E. L. and Gauthier, J. S. (1986). ACSL: *Advanced Continuous Simulation Language – User Guide/Reference Manual*, Mitchell and Gauthier Assoc., Concord, MA.

[14] Zeigler, B. P. (1976). *Theory of Modeling and Simulation*, John Wiley, New York.