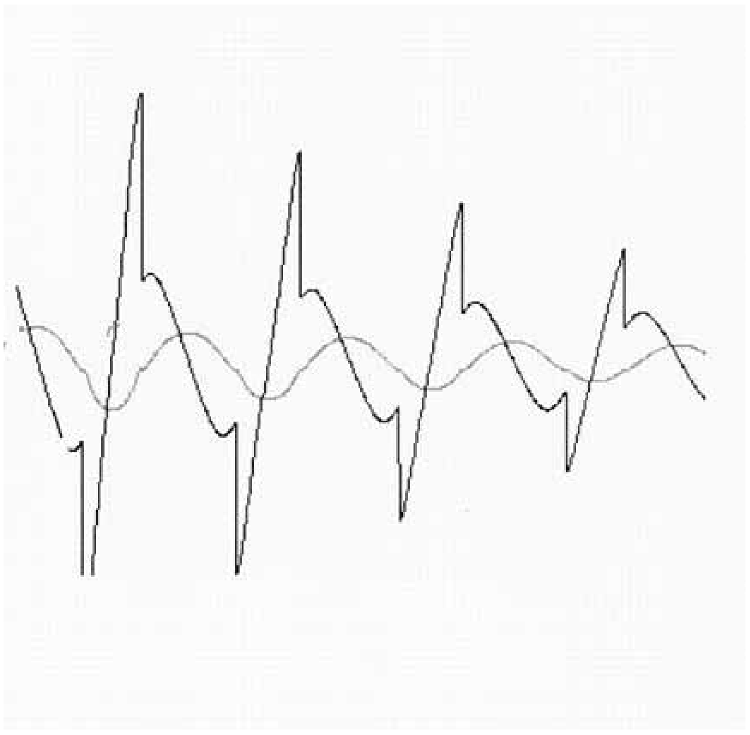
FORTSCHRITTSBERICHT
SIMULATION FBS 3



Johannes Plank

State Events in Continuous Modelling and Simulation

ISBN Ebook 978-3-903347-03-8
ISBN Print 978-3-901608-53-7

DOI: 10.11128/fbs.03

asim

ARGESIM

Fortschrittsberichte Simulation

FBS Band 3

Herausgegeben von **ASIM**
Arbeitsgemeinschaft **Sim**ulation, Fachausschuß 4.5 der GI

Johannes Plank

State Events in Continuous Modelling and Simulation

ARGESIM / ASIM – Verlag, Wien, 1997
ISBN Print 978-3-901608-53-7

Ebook Reprint 2020
ISBN Ebook 978-3-903347-03-8
DOI: 10.11128/fbs.03

ASIM Fortschrittsberichte Simulation / ARGESIM Reports

Herausgegeben von **ASIM**, Arbeitsgemeinschaft Simulation, Fachausschuß 4.5 der **GI** und der **ARGESIM**

Betreuer der Reihe:

Prof. Dr. G. Kampe (ASIM)
Fachhochschule Esslingen
Flandernstraße 101, D-73732 Esslingen
Tel: +49-711-397-3741, Fax: --397-3763
Email: `kampe@ti.fht-esslingen.de`

Prof. Dr. D.P.F. Möller (ASIM)
Inst. F. Informatik, TU Clausthal-Zellerfeld
Erzstraße 1, D-38678 Clausthal-Zellerfeld
Tel: +49-5323-72-2404, Fax: –72-3572,
`moeller@vax.in.tu-clausthal.de`

Prof. Dr. F. Breitenecker (ARGESIM / ASIM)
Abt. Simulationstechnik, Technische Universität Wien
Wiedner Hauptstraße 8 - 10, A - 1040 Wien
Tel: +43-1-58801-5374, Fax: +43-1-5874211,
Email: `Felix.Breitenecker@tuwien.ac.at`

FBS Band 3

Titel: State Events in Continuous Modelling and Simulation

Autor: Johannes Plank
`johannes.plank@thalesgroup.com`

Begutachter des Bandes:

Prof. Dr. F. Breitenecker TU Wien, Prof. Dr. D.P.F. Möller Univ. Hamburg

ARGESIM / ASIM – Verlag, Wien, 1997
ISBN Print 978-3-901608-53-7

Ebook Reprint 2020
ISBN Ebook 978-3-903347-03-8

DOI: 10.11128/fbs.03

TECHNISCHE UNIVERSITÄT WIEN

DISSERTATION

State Events in Continuous Modelling and Simulation

Concepts, Implementations and New Methodology

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch Naturwissenschaftliche Fakultät
von

Dipl. - Ing. Johannes Plank
Vierthalerstraße 7
A - 5280 Braunau am Inn
Matr. Nr. 8826610
geboren am 25. 01. 1970 in Braunau am Inn

Wien, im Jänner 1997

Abstract

This thesis deals in a comprehensive way with state events (events depending on states) in continuous modelling and simulation.

Starting from systems which are described by ordinary differential equations state events are introduced by means of simple examples.

A classification of state events in four classes is presented, where events that come up from similar origins or have similar effects on the proceeding of a simulation get combined.

Furthermore, methods are developed and discussed for inserting and describing state events in mathematical models and/or simulators.

Based on this methodology and on a Model Interconnection Concept a completely new concept, the *Meta Model Concept*, is introduced and described.

The aim of this concept is to simplify modelling and the handling of state events in continuous simulation.

An essential aspect of the Meta Model Concept is the inclusion of the experiment level to the description of state events.

The transfer of the events from the model level to the experiment level and also the description of state events as separate model components support among other things the reusability of simulation models or model components and also the installation.Therefore the modelling of events becomes much more simple.

This concept enlarges the description possibilities of state events and allows dynamic model structures and other complex dynamic model/experiment connections.

In addition, a *state of the art* report on the handling of state events in modern simulators is given.

The application of the methods and the Meta Model Concept is shown with concrete examples.

Deutsche Kurzfassung

Diese Dissertation beschäftigt sich in umfassender Weise mit State Events (zustandsbedingte Ereignisse) in der kontinuierlichen Modellbildung und Simulation.

Ausgehend von mit gewöhnlichen Differentialgleichungen beschriebenen Systemen, werden anhand von einfachen Beispielen State Events eingeführt.

Es wird eine Klassifizierung von State Events in vier Klassen präsentiert, in denen Ereignisse zusammengefasst sind, die durch einen ähnlichen Ursprung, bzw. deren Auswirkungen auf den Simulationsablauf in gleicher Weise charakterisiert sind.

Weiters werden Methoden entwickelt, mit denen State Events in mathematischen Modellen und/oder in Simulatoren eingebaut und beschrieben werden.

Basierend einerseits auf diesen Beschreibungsmethoden und andererseits auf einem Modellverbindungskonzept, wird ein völlig neues Konzept, das *Meta Model Concept*, vorgestellt und beschrieben.

Ziel dieses Konzeptes ist sowohl die Modellierung, als auch die Behandlung von State Events in der kontinuierlichen Simulation zu vereinfachen.

Wesentlicher Punkt des Meta Model Concepts ist die Einbeziehung des Experiment Level in die Beschreibung von Events.

Durch die Verlagerung der Ereignisse weg von der Modellebene auf die Experimentebene wird unter anderem die Wiederverwendbarkeit von

Simulationsmodellen bzw. Modellkomponenten unterstützt, aber auch der Einbau, also die Modellierung von Ereignissen stark vereinfacht.

Dieses Konzept erweitert nicht nur die Beschreibungsmöglichkeiten von State Events, sondern erlaubt auch dynamische Modellstrukturen und andere komplexe Modell/Experiment-Verbindungen auf dynamischer Basis.

Ergänzt werden die Ausführungen durch einen *State-of-the-Art* Report über die Behandlung von State Events in modernen Simulatoren und die Anwendung der Methoden und des Meta Model Concepts auf konkrete Beispiele.

Acknowledgement

This thesis was worked out at the Department of Simulation Techniques at the Technical University Vienna, Austria under the supervision of a.o.Univ.Prof.DI Dr.techn. Felix Breitenecker. I want to express my thanks to him for the fruitful discussions, the good co-operation, the joint study trips and his involvement.

Furthermore, credits to my colleagues at the department and outside who have supported me and my work.

Contents

1 Motivation and Introduction **5**

2 Foundations **9**

2.1 The Modelling Process 9

2.2 Simulators 11

2.2.1 Attempts for Standardisation 14

2.2.2 Numerical Integration Methods 16

2.3 Experiment 17

2.3.1 Interpretation 18

3 Events **20**

3.1 Events 20

3.2 Events and Discontinuities 22

4 State Events **24**

4.1 Introduction 24

4.2 Classifications for State Events 30

4.3 Description 36

4.4 Handling 37

4.4.1 Detecting 38
4.4.2 Locating 40
4.4.3 Passing 45
4.4.4 Restarting 46

5 State Events in Simulators – State of the Art 48

5.1 ACSL 50
5.2 SIMULINK 53
5.2.1 SIMULINK 1. 3c 54
5.2.2 SIMULINK 2. 0 56
5.3 ESL 59
5.4 MOSIS 63
5.5 SIMNON 65
5.6 SLIM 69
5.7 ANA 2.x 72
5.8 DYMOLA 75
5.9 ModelMaker 77

6 Event Methodology – Description of Methods 79

6.1 "All-in-one" - Method 79
6.2 "Discrete Section" - Method 82
6.3 "Concatenated Runs" - Method 84
6.4 "Component Exchange" - Method 86
6.5 "Sequential Models" - Method 87
6.6 "Extended Experiment" - Method 89

7 Event Methodology – Examples and Case Studies 92

7.1 Bouncing Ball 92

7.1.1 "All-in-one" - Method – SIMNON 94

7.1.2 "Component Exchange" - Method – SIMNON 95

7.1.3 "Sequential Models" - Method – SIMNON 96

7.2 Block on a Rough Surface 99

7.2.1 "Discrete Section" - Method – ACSL 100

7.2.2 "Concatenated Runs" - Method – SIMNON 103

7.3 Constrained Pendulum 105

8 The Meta Model Concept 106

8.1 The Model Interconnection Concept 107

8.2 The Extended Model Interconnection Concept 108

8.3 The Meta Model Concept 111

8.3.1 Formalised Description 113

8.3.2 Assessment of the New Concept 117

8.4 Application of the Meta Model Concept for Events 119

8.4.1 Bouncing Ball 119

8.4.2 Looping Pendulum 122

8.4.3 Constrained Pendulum 124

8.4.4 Block on a Rough Surface 124

8.4.5 Pilot Ejection 127

8.4.6 Reconditioning Plant 129

8.5 Other Application 131

8.5.1 Repeated Linearisation 131

8.5.2 Integration Algorithms – Stiff Systems 133

8.6 Implementation of the Meta Model Concept 135

Bibliography **139**

Index **146**

Chapter 1

Motivation and Introduction

The field of continuous system simulation, though a young discipline has undergone big changes in the last few years.

Firstly, the basic tools, the computers were developed very rapidly. Now they are capable of an amount of computing power that most of us did not dream of a few years ago. That is why some numerical methods that were formerly said to be too costly are now implemented and applied with no hesitation. Even new methods can be developed with less constraints concerning the costs.

Nowadays modern simulation languages offer a big variety of algorithms to deal with simulation models. Therefore, specialists are able to build up sophisticated models that correspond to more and more complex systems.

The demands to simulation languages are raised parallel to these facts – I only want to mention headwords like *modular modelling* or *object orientation* and *model libraries*. That would support the **reusability** of models or components of models.

In this situation *state events* get a new status. In modelling electrical or mechanical systems one will easily come across the need to use state events in this model and will also demand the corresponding statements and algorithms. Applying more complex structures and algorithms should be possible in a **user friendly** way. On the other side, the simulation languages are required to offer an environment that supports **user robustness** and does not react too sensitively if not all of the possible errors are blocked; including those that arise due to numerical reasons.

Furthermore simulation models should **correspond with the represented real system** not only in view of the results but also of the structure of the description. Models that are constructed according to this requirement are much more clearer to read and, obviously, also to be built up. Since it is often impossible to find a single physical representation of complex systems, state events can help to achieve this.

These four main requirements, the

- reusability and the
- correspondence with the real system (as far as possible) concerning the models or model components and the
- user friendliness and
- user robustness concerning the applied algorithms

are in some way the cornerstones of the considerations and reflections that led to this thesis and should be kept in mind.

This thesis gives a *classification* of state events. I worked it out in order to put together state events coming up due to similar reasons and causing similar problems for the numerical algorithms.

This thesis defines a new *methodology of handling state events* which is based on a comprehensive review of the state of the art of event description in nowadays simulators. I worked out six different methods of handling state events After presenting and discussing these methods I will illustrate them by giving two case studies that are modelled according to the presented methods in different simulation languages. Consequently, I show that different methods can be implemented in one simulation language as well as one special method in different simulators.

These investigations show that some of the "grown" methodologies are insufficient for modern concepts and consequently as the main result of this thesis a *meta model concept* for state events is presented. This completely new concept that breaks to some extent with some traditions in continuous system simulation as it merges experiment and modelling level shows a new and comfortable way of modelling and simulating state events. It also offers to cope with other techniques as consecutive linearisation of models and handling of partly stiff models.

Structure of the Thesis

Chapter 2 deals with foundations of continuous system simulation. I present this topic at the beginning of this thesis in order to explain what I mean by talking of e.g. *parameters*,... as there is no standardised vocabulary in this field.

In chapter 3 I introduce *events* and from chapter 4 on I concentrate on *state events*. This chapter contains a classification and furthermore I discuss the description and handling – including some numerical algorithms – of state events.

The next chapter shows how state events are used and handled in modern simulation languages. After a presentation of the possibilities I illustrate them with a model of the example of the *constrained pendulum* in each simulation language.

Chapter 6 focuses on the methodology of state event handling and shows six suitable methods. How case studies can be modelled by applying different methods will be presented in the subsequent chapter.

In chapter 7 I show how these methods can be applied for modelling concrete examples in simulators. It is also shown that different methods can be used for modelling the same example in the same simulation language.

Finally, I present my new *meta model concept* in chapter 8, explain its features and discuss the upcoming advantages and disadvantages.

In order to make the problems, phenomena, methods,... more evident, simple examples, well known benchmarks and case studies are presented.

Furthermore, I discuss the possibilities to implement the new concept in different simulators.

Chapter 2

Foundations of Modelling and Simulation

This chapter deals with the foundations of numerical continuous system simulation. I do not want to present a basic course in simulation but discuss some fundamental principles.

2.1 The Modelling Process

As neither exact definitions nor strict regimentations constrain the large field of simulation, it is quite difficult to come to a readily comprehensible discussion.

Therefore, I try to define the terms concerning the field of simulation used in this thesis and explain the principles of simulation in the way I understand it, in order to come to a common language and being precisely understood.

Simulation corresponds to very old and commonly used methods to solve problems. In brief, it is the transition to analogue models, finding a solution and trying to apply this solution to the initial problem. The reasons why we apply simulation and do not search for solutions on the original systems are supposed to be sufficiently known.

Simulation is the imitation or reproduction of real systems in order to make experiments on it without touching the process. But what is a "real system"? Following Zeigler in his monograph on modelling and simulation [64, p. 4], let us define a system as "some part of the real world, which is of interest." And then: "The system may be natural or artificial, in existence presently or planned for the future."

We pass a series of abstractions from reality to simulation. The definition above contains already the first step of abstraction: when we speak of a real system, we have to isolate it from its environment. We therefore divide the "real world" – this constitutes again a term worth of long discussions, but here we understand it in the usual way – in three parts,

the system the part we are interested in

the system boundary the factors that influence or are influenced by the system

the environment everything else

This step as well as the following require a large amount of knowledge, experience and skill because just here the reduction of the real world starts. We have to be aware that everything we neglect in one of the steps cannot be part of the simulation later.

That means that we have to decide before beginning this process what kind of results we want in the simulation runs. Therefore, the parameters we want to observe or change must not be part of the environment.

It is necessary to reduce the system and neglect parts of it because it is always impossible to know all the influences of a particular system. Furthermore, we have to try to keep the model small and clear so that we can handle it. Another reason is to take the time the computer needs for the dynamic calculation into account. This time is a very important factor concerning the costs.

In this actual step we have to define the system boundary according to our experience with the system's behaviour and also to the question we put to our simulation, i.e., what output we are interested in.

In the next step, our task is to build up a physical model of the system: We reduce the system to standard physical components that can be described by some known laws. This is comparable to the building of a model by using parts out of a box of bricks. This representation of the real system can then be described in a written model. In other words we describe the system by a combination of fundamental physical laws. In general, these are Newton's laws for mechanical models and Kirchhoff's laws for electrical models. At the end of this step we get a set of mathematical equations that describe the real system under certain constraints satisfactorily accurate.

The next task is to simplify the obtained set of equations and to prepare it for the use within a simulation language.

The topics of the next chapter are the characteristics of simulation languages and how the description of the model has to be prepared so that a simulation language can handle it.

2.2 Simulators

In the early days of simulation *simulation languages* supported the modelling of systems. Later, *simulation packages* were developed for the modelling of more complex systems. They were then extended with some features for simulating and called *simulation environments*. Roughly spoken, we can subsume these terms under *simulator*. A simulator can be defined as a kind of higher level programming language or simply a software tool that provides special functions for model implementations, mathematical functions for dynamic calculations and facilities for experimentation and for post-processing the obtained results.

Nowadays, the market offers a large variety of commercial, experimental and good simulation languages. Some of them were developed in the late 60's and since then have been suffering of historical constraints; others were invented only a few years ago and are still trying to produce not only system crashes.

Seriously, there are a lot of good working simulation languages on the market. Some of them are designed for special use, others are more general purpose simulation tools.

All the simulators I am discussing here are capable of dealing with continuous-time models. The models are built up by the use of ordinary differential equations (ODEs) that are describing the behaviour of the system we want to simulate in the state space. These ODEs represent initial value problems (IVPs), whereas boundary value problems (BVPs) are used only by means of optimisation.

Usually, the ODEs have to be transformed to a set of first order ODEs. The independent variable is mostly the time t. The right side of the equation also denotes the *structure* of the system that includes the dimension of the state space. If we change the "right sides" we also change the structure.

$$\begin{aligned} \dot{x} &= f(x,u,t) \\ \dot{x}(t_0) &= x_0 \end{aligned}$$

The model description in a simulation language always consists of two parts[1].

At first we have a structural or topological description of the model. This is a set of statements or blocks defining functions.

According to the description of the structure of the model simulation languages can be classified in two groups:

- equation-oriented
- block-oriented

[1]A detailed description can be found in [56, p.16]

This classification is in some way of historical origin. The two groups can be characterised by two different ways of thinking. On the one hand, there are the mechanical engineers and physicists who are traditionally used to deal with equations. On the other hand, the electronical engineers who prefer to divide systems into functional blocks and connect them via input/output ports.

The second part is the description of the parameters. We distinguish between three classes of parameter types:

1. model parameters:
   (a) constants: quantities that never change the value
   (b) parameters: quantities that are constant during a simulation run but that may be changed between the runs
   (c) initial values
2. integration parameters: parameters which control the numeric integration
   (a) integration algorithm
   (b) step size
   (c) start and stop of the simulation
   (d) additional values when using variable step size adjustment
3. output parameters:

   they are especially important in models of control engineering:
   (a) output values: specifies which quantities should be given out
   (b) output interval
   (c) output form: curves, numbers, ...
   (d) output device: display, printer, plotter, hard disk, ...

(e) output environment: Sometimes there are additional statements beside the simulation results necessary, e.g. a legend for a plot, a title, page numbers, axes and their labelling, ...

(f) output format

2.2.1 Attempts for Standardisation

Different simulation languages may use different shapes of model descriptions what makes writing and reading a very costly task. That is why already in the very early stadium of the development of simulation languages the *Simulation Council (Sci)* founded the *Committee for Continuous System Simulation Languages (CSSL)* in 1965 that worked on standardisation. In 1967 they came out with the CSSL-Standard (CSSL'67) [61].

The three aims of the CSSL'67 standard are

- a standardised user interface – for the unskilled user
- a flexible model interface – for the expert
- an expandable computer interface – for the future

The first was achieved by a clear, obvious form of model description statements, with operators capable of handling most problems including differential equations (with user hidden procedures), a complete set of problem-oriented diagnostics for both compilation and runtime errors, sorting algorithms, ...

The second aim requires a CSSL acting as an adjunct to an arbitrary procedural language so that the user can extend the set of operators by programming his own.

Unable to "foresee the exact path that would be taken" [61, p.283] the committee demanded to provide the possibility for flexible expansion.

In addition, already in those days the CSSL'67 defined structural elements such as the separation of INITIAL, DYNAMIC and TERMINAL regions in the model description as well as macros.

15 years later some new attempts were made for a new standardisation, the so-called CSSL'81. But this CSSL'81 never left the level of recommendations.

The main innovations of these recommendations as proposed by Crosbie and Hay in [22] are

- the separation of experiment and model
- submodel features
- features for describing and handling discontinuities and
- segment features

The third item is the most interesting for us. It proposes to include conditional statements and numerical routines for detecting discontinuities and ensuring accurate numerical integration across it.

By proposing submodel features the CSSL'81 anticipates already the developments of model modularity and also object oriented descriptions.

Apart from the "CSSL languages" like ACSL, ESL, MOSIS we can find high developed alternative languages like SIMULINK or SIMNON. For detailed information I refer to e.g. [40].

A recent attempt for standardisation is *VHDL-AMS*[34] , an extension of the *Very High Density Logic (VHDL)* standard, a modelling standard for the description of logic electronic circuits, by means of "analog components and signals", i.e. ODE model parts.

At first this attempt claimed for generality, but in fact just tends to a standardisation in electrical/electronical applications or almost only in electronic applications.

2.2.2 Numerical Integration Methods

The core of a simulation language is the built-in algorithms for numerical integration of ODEs. As the system is described with sets of ODEs, the state variables are calculated by using numerical methods. Here the term *algorithm* is used in contrast to *method* to differentiate between the theoretical method and its implemented respectively programmed form, the *algorithm*.

There are two different groups of numerical integration methods:

- one-step-methods
- multi-step-methods

Both methods apply – as we can deduce from the names – a step-by-step method. The approximate solution is updated with every integration step.

One-step-methods only include the information from the actual integration step in the next step. The actual values of the state variables serve as the "exact" initial values for the next step.

On the other side, multi-step methods use the information of two and more former steps for the next.

Nowadays, there are a lot of different and efficient methods for the integration of stiff systems, e.g. the famous Gear-methods. Although they can also be seen as implicit single or multi-step methods, they should be mentioned in a separate, third group:

- methods for stiff systems

For a detailed information I recommend for instance Lambert's book on methods of numerical integration [36] where the reader can find the description of almost all the methods implemented in modern simulation languages.

2.3 Experiment

Running a simulation is what we usually call making an experiment. As declared above, simulation is defined as a method of solving problems. In other words, we try to gain knowledge about the model that is applicable to the real world. The question is how to get this knowledge and the obvious answer is "in making experiments" with the model.

When experimenting with a model one has to keep in mind that the model is not valid for all possible experiments. Therefore, Zeigler defined in his monograph [64] the *experimental frame*, the set of experiments a model is valid for.

An experiment in its simplest form is just to calculate the ODEs over a certain period of time with a certain set of parameters (initial values,...) in order to gain knowledge about the system behaviour that means to observe the output parameters as well as the internal ones over the simulated time.

A series of experiments on the same model is called a study . In a study we change different parameters, e.g. the initial values, the step-size, the integration algorithm, ... and can observe the different results. Then the interpretation of the correspondence between results and parameter changes can start.

In the traditional CSSL languages these kinds of experiments have to be implemented in the INITIAL/DYNAMIC/TERMINAL structure or started interactively at runtime level and are therefore, more or less, the only possible ones.

In the CSSL'81 the authors already had a differentiated view of experiments in mind when they proposed the separation of the model description and the experiment description. As we can read in the proposals of Crosbie and Hay in [22, p.187] the experiment is seen as a special unit that can be performed upon different models and – also the other way round – a model should be allowed to be subject to different experiments.

The most general point of view is given by the concept of Breitenecker and

Solar [17]: ”An experiment is the performance of a certain method with a certain model” and a ”method is an operation or algorithm which defines an action using a model (*whatever can be done with a model*)”.

This definition includes the very important field of optimisation and linearisation in the term experiment and also enables the data exchange between model and method so that very sophisticated methods can be developed and applied. Here, the ordinary integration of the ODEs is seen as a basic method.

This definition also serves as a basis for the ”Extended Experiment method” that I propose in chapter 6.

2.3.1 Interpretation

At this point we come back to where we started from. After modelling a real system and making experiments with this model we hopefully get simulation results. Now the gathered information has to be interpreted in context with the real system.

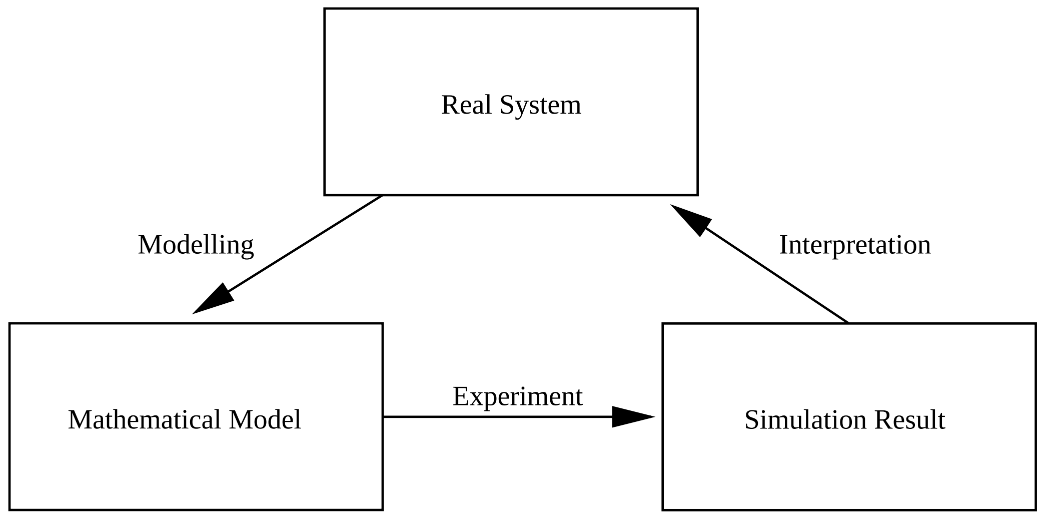


Figure 2.1: The Simulation Process

At the end of this short overview figure 2.1 shall illustrate the process of simulation: starting from a real system we build up a model with which we can make experiments. Then we have to interpret the results and transfer them to the real system.

Chapter 3

Events

In this chapter I will introduce the term *event* and its meaning in continuous system simulation. Furthermore, I will deal with the differences between *time* and *state events*, and so I come to the centre of interest of this thesis, the state events.

Finally, I will shortly discuss the difference between *event* and *discontinuity*.

3.1 Events

Modern simulation techniques more and more use events. This is the result of technical developments as well as of the fact that the simulated systems nowadays are getting very complex. But at first I should try to define what an event is.

When we look at the CSSL'67 standard we do not find the term at all. This can be seen as a "drawback of too early a software standardisation" as Cellier writes in 1993 [19, p.41]. And in fact we can agree only by comparing the abilities of the computers in those times and now and the consequential differences in the demands.

A very general definition of the term event that is also very close to the term used in natural language:

When something extraordinary is happening we call it an event.

As I am arguing in the field of continuous simulation I can define an event in a more concrete form in the following way:

An event occurs when the simulation run is stopped, sudden discrete changes, which do not consume time, are carried out and the simulation is restarted again.

Here, "discrete changes" characterises everything between the isolated change of a variable to discrete subprocesses as part of the simulated system.

The type of the condition that triggers an event leads us to a fundamental classification:

time events are triggered at a certain time (periodically or isolated)

state events are triggered when a condition dependent on the state of the system is fulfilled

An example for a time event is the regulation of a dynamic system by a digital controller. At certain intervals the controller periodically transmits new control inputs to the system.

A similar example can be formulated for state events when we consider again a dynamic system and a controller. But here the controller changes the input only when the state has reached a certain threshold, e.g. the water inflow is cut when the bottle is full.

In order to outline roughly a fundamental difference between the two groups of events we can state that time events are mostly applied in context with digital (discrete) controllers and with special input functions coming up from control engineering.

On the other hand, time events are an instrument to build up discrete-time models with continuous-system simulation tools.

In contrast to state events time events are much easier to handle as the time when they are triggered is known in advance. The difficulties with the handling of state events are discussed in the following chapters.

Taking in a very extreme point of view we could say that all the simulation of the model itself is only composed of time events: Due to the use of numerical integration methods we could interpret the dynamic calculation as a series of time events, every integration step marks the time when the state of the system is updated. But to my opinion this way of thinking leads us far away from where we started.

The basis of the continuous system simulation is the identification of systems with continuous or steady processes that we describe with ODEs.

On the other hand, one can say that the interpretation of natural processes as continuous is a simplification coming up from Newton's Physics.

So we have to keep in mind that the models we are considering when talking about general purpose simulation languages are "simple" mechanical or electrical models. The term simple is used here to denote systems that deal with "classical" masses, velocities, ..., that we can describe with Newton's Physics sufficiently precise. Therefore, we also may talk of a *continuous world* (Newton's Continuum).

3.2 Events and Discontinuities

Although the two terms *event* and *discontinuity* are mostly used interchangeably in the specialist literature I would like to outline the big difference I see between them and therefore dedicate a separate section to it.

As shown above in the definitions I have a very general idea of what I call an event. Every event causes something to change. More concrete, it causes a part of the model description or variables to change.

On the other side, there is the term discontinuity. When analysing this term we get the result that it describes something that is not continuous – being the difference to the term event and establishing also the constraint of that term. Talking e.g. about a continuous input function that is changed by an event we can obviously also call it a discontinuity in the function. But on the other hand there are also events changing the model structure. In this context we cannot use the term discontinuity as there is nothing that could be discontinuous. On the contrary, looking at the point of change, the function may be continuous or even differentiable. Even if we analyse the simulation results, we can see that an event needs not force the solution to be discontinuous.

Chapter 4

State Events

In this chapter I will concentrate on *state events*. After an introduction I will focus on a new classification that I developed. This classification will then be taken as a basis and will be repeatedly referred to. Furthermore, a methodology will be given for the description of state events including concrete examples and applied methods. The next focus is the state event handling presenting the *four stages method* and discussing each of the steps in detail. I will give and discuss some numerical methods that are applied in continuous simulation for finding state events.

4.1 Introduction

In continuous system simulation we apply state events for different reasons.

One aspect is the simplification of models. In modelling real systems we are often confronted with processes that go off very quickly compared to the dynamic of the rest of the system. In order to safe costs in development, these processes are then not modelled as processes but as events. This obviously also saves computing time.

As an example I will discuss the model of a bouncing ball. This is a very simple but capable example to show special methods, so that we will take it repeatedly in this thesis.

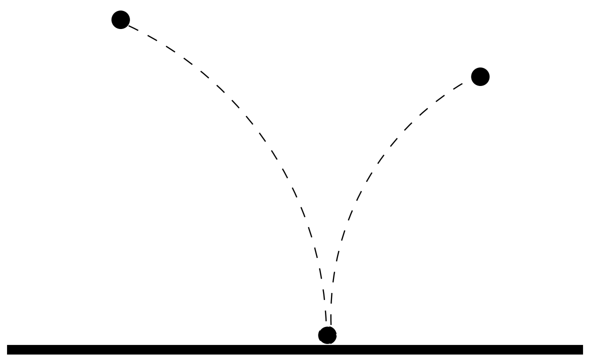


Figure 4.1: The simplified bouncing ball

The – very simplified – motion of the ball can be described by the following equation:

$$\begin{aligned} \ddot{x} &= -g \\ \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \end{aligned}$$

In a macroscopic view the ball may be seen as a concentrated mass point. Besides, the model would become more real if terms modelling the air resistance,… were added, but here we concentrate on the event and so this very simple model will be sufficient.

But what happens when the ball hits the ground? Afterwards the equation above is valid again. The process before is the crucial point. Obviously, the process of hitting the ground and jumping back constitutes a continuous process. The kinetic energy of the fall is gradually converted into elastic energy and then the other way round. The simplest way is now to replace the microscopical process of hitting the ground by an event. Here event means the stopping of the dynamic, inverting the direction of the velocity including a damping factor α

$$\dot{x}_{new} = -\alpha \cdot \dot{x}_{old}$$

and starting the calculations again. It is therefore the sudden – i.e. not time consuming – change of the velocity $\dot{x}$ to that value as if the microscopic process of hitting had just gone off. The microscopic view, where this process is also modelled, is presented in the next section.

In this case the event is triggered when the ball touches the ground that is when the height x equates zero:

$$x = 0$$

When this condition becomes true depends on the dynamic calculations as the height of the ball is one of the state variables.

A summary of the system governing equations and initial values:

state equation	$\ddot{x}$	$=$	$-g$
initial values	$\dot{x}(t_0)$	$=$	$\dot{x}_0$
	$x(t_0)$	$=$	x_0
event condition function	x	$=$	0
event	$\dot{x}_{new}$	$=$	$-\alpha \cdot \dot{x}_{old}$

Another aspect is the fact that it is often not possible to describe the behaviour of a real system by one single set of equations. In this case it is necessary to decompose the system into subsystems. The different subsystems act exclusively, i.e. only one subsystem describes the system at a time. The conditions which subsystem is valid have to be formulated in a corresponding way.

As an example I mention here a simple pendulum, a mass hanging on a string.

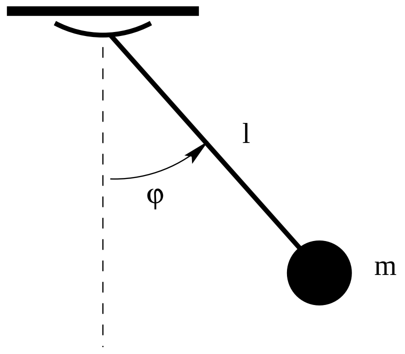


Figure 4.2: Mathematical pendulum

Reducing the model by the assumption that the mass is concentrated in a point, the string is not elastic and has no resistance . . . we can formulate this example with an ordinary differential equation of second order, where φ is the angular, g is the gravitational force, l the length of the pendulum and d a damping factor:

$$m \cdot l \cdot \ddot{\varphi} = -m \cdot g \cdot \sin\varphi - d \cdot l \cdot \dot{\varphi}$$

Taking into account that the pendulum reaches – when looping – a point where the centrifugal force is not big enough to tighten the string, the motion of the mass gets the form of a free fall.

The very simplified description of this motion in terms of ODEs is as follows:

$$\begin{aligned} \ddot{x} &= 0 \\ \ddot{y} &= -g \end{aligned}$$

Here again, the model would become more real if we added the description of the air resistance,. . .

The condition for the event is – as already mentioned above – the fact that the centrifugal force is not big enough to tighten the string any more.

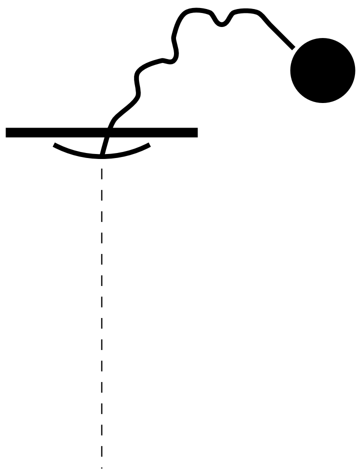


Figure 4.3: Pendulum during free fall

The centrifugal acceleration minus the corresponding component of gravity is calculated by:

$$f = \dot{\varphi}^2 \cdot l - g \mid \cos\varphi \mid$$

It is therefore a function of the state variable $\dot{\varphi}$ and the parameter l.

Here, the term event means the stopping of the dynamics, changing the equations and restarting the dynamics again.

Now we have two sets of equations with corresponding conditions and transformations:

state equation 1

$$\ddot{\varphi} = -\tfrac{g}{l}\sin\varphi - \tfrac{d}{m}\dot{\varphi}$$

initial values

$$\begin{aligned} \dot{\varphi}(t_0) &= \dot{\varphi}_0 \\ \varphi(t_0) &= \varphi_0 \end{aligned}$$

event condition function

$$\dot{\varphi}^2 \cdot l - g \mid \cos\varphi \mid = 0$$

event: switch to state equation 2

$$\begin{aligned} x &= l \cdot \sin\varphi \\ y &= -l \cdot \cos\varphi \\ \dot{x} &= l \cdot \cos\varphi \cdot \dot{\varphi} \\ \dot{y} &= l \cdot \sin\varphi \cdot \dot{\varphi} \end{aligned}$$

state equation 2

$$\begin{aligned} \ddot{x} &= 0 \\ \ddot{y} &= -g \end{aligned}$$

initial values

$$\begin{aligned} \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \\ \dot{y}(t_0) &= \dot{y}_0 \\ y(t_0) &= y_0 \end{aligned}$$

event condition function

$$\sqrt{(x^2 + y^2)} - l = 0$$

event: switch to state equation 1

$$\begin{aligned} \varphi_{new} &= -\tfrac{\pi}{2} - \arctan\tfrac{y}{|x|} && \text{if} \quad x < 0 \\ \varphi_{new} &= \arctan\tfrac{|x|}{y} && \text{if} \quad x \geq 0 \\ \dot{\varphi}_{new} &= y \cdot \cos\varphi_{new} + y \cdot \sin\varphi_{new} \end{aligned}$$

In the two examples described in the section above we were familiarised with two very different forms of state events. In the first example it was one state variable that was changed and in the second it was the set of equations.

Therefore, we can see that events arise for very different reasons and on the other side their descriptions in the model may vary a lot. In the following section I develop a classification for state events. The classes then represent events that join a common description.

4.2 Classifications for State Events

Two Classes Approach

In a first attempt let us divide the state events into two different classes as we have already done in the section above:

Class 1 event: Change of (state) variables or parameters

Class 2 event: Change of equations

Combinations of these two classes can be subsumed in the second one. If we want to classify the two examples given above, we put the bouncing ball in the first class: a state variable is changed here. The pendulum is member of the second class, as there is a structural change in the equations.

It is not easy to build up formal descriptions of the two classes. In the "Change of equations" we can describe a system with n possible different equation changes as:

$$\dot{x}(t) = \begin{cases} f_1(t, x(t)) & \text{if condition} \quad 1 \\ f_2(t, x(t)) & \text{if condition} \quad 2 \\ \vdots & \\ f_n(t, x(t)) & \text{if condition} \quad n \end{cases}$$

The indices given to the conditions and equations have nothing to do with the sequence, the chronological succession. This depends only on the condition that gets true or false depending on the calculated states. As I have already mentioned above I obviously require that only one condition can be true at a time.

It is more difficult to give a general description of a system of the first class. One possibility is to describe the phases before/after and at the event separately. This is obviously a chronological description, too. For the event "change of state variables" we can write:

$$\begin{array}{lcll} x(t_0) & = & x_0 & \text{the initial value} \\ \dot{x}(t) & = & f(t, x(t)) & \text{until condition becomes true} \\ x(t)_{new} & = & g(x(t_{event})) & \text{when condition is true} \\ \dot{x}(t) & = & f(t, x(t)) & \text{after condition was true} \end{array}$$

In this description I require that the condition for the event is false again after the changes are made. This proceeding can be seen as part of the condition. That means that in the condition there exists a mechanism that sets the condition false again, that changes the conditions or disables it after the changes are made. Here the important thing is *when* the condition *gets* true, whereas in the other class it is *if* a condition *is* true. There is another important difference in the two classes: In the first class events are detected with IF conditions, whereas the events in the second class occur due to a WHEN condition.

This classification can be seen as a very fundamental one. If we change a parameter or a state variable in a simulation run, it is nothing else than what we do if we start a new experiment with a certain model. In the initialisation we set the values and then we start the calculations. So we can identify the first class with starting a new experiment.

What about the second class? Changing equations cannot be part of an initialisation process. This change is more profound because the model describes a completely different process after the change. So we can say as well that the change of the equations is also the change of the model. And therefore, the second class represents an experiment with a new – or better different model.

From this point of view, a combination of the two classes is obviously subsumed in the second one. When we start a simulation we clearly have to set parameters and so there is no difference if we combine them.

When we now examine this classification in conjunction with the real system more carefully, we realise that it is much too rough. Events are mixed up in a class stemming from completely different origins in the real system and causing completely different actions.

Four Classes Approach

In the two classes approach we put together parameters and (state) variables. For instance, in mechanical systems we find them almost always in this combination. When a parameter is changed in mechanical systems, it causes a change of a state variable in most of the cases and therefore a discontinuous solution.

Let us look again at the pendulum. This time we do not let it loop but put in a pin.

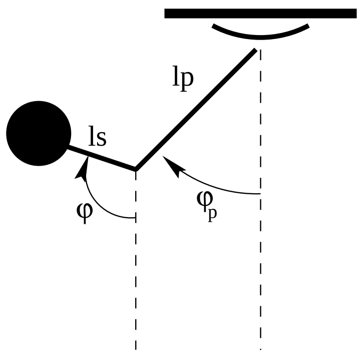


Figure 4.4: Constrained Pendulum

When now the pendulum reaches the pin at φ_p not only the parameter "length" of the string is reduced but also the angular velocity has to be changed due to the conservation of the momentum. The corresponding equations are:

$$l_{new} := l - l_p$$

$$\dot{\varphi}_{new} := \dot{\varphi}\frac{l}{l-l_p}$$

When the pendulum swings back again and leaves the pin, the equations are:

$$\begin{aligned} l_{new} &:= l \\ \dot{\varphi}_{new} &:= \dot{\varphi}\frac{l-l_p}{l} \end{aligned}$$

Therefore, in this model a parameter *and* a state changes discontinuously.

In order to summarise the equations we can describe the model in the following way:

state equation

$$\ddot{\varphi} = -\frac{g}{l}\sin\varphi - \frac{d}{m}\dot{\varphi}$$

initial values

$$\begin{aligned} \dot{\varphi}(t_0) &= \dot{\varphi}_0 \\ \varphi(t_0) &= \varphi_0 \end{aligned}$$

event 1 condition function

$$\varphi \cdot \varphi_p - \varphi_p^2 \geq 0$$

event 1

$$\begin{aligned} l_{new} &= l - l_p \\ \dot{\varphi}_{new} &:= \dot{\varphi}\tfrac{l}{l-l_p} \end{aligned}$$

event 2 condition function

$$\varphi \cdot \varphi_p - \varphi_p^2 \leq 0$$

event 2

$$\begin{aligned} l_{new} &:= l \\ \dot{\varphi}_{new} &:= \dot{\varphi}\tfrac{l-l_p}{l} \end{aligned}$$

On the other hand, there are systems without that combination of changes of parameters and state variables. For instance, a discrete controller may provide an input to a continuous system which may change discontinuously but does not cause any discontinuous changes of a state variable. Therefore, a distinction of these two types seems to be reasonable.

The second class should also be split up into two classes. When examining corresponding examples we can realise that there are two very different classes. Besides, it must be taken into account that even profound changes as those of equations need not cause discontinuous solutions.

The two different types of what I subsumed in the two classes approach as the change of equations are

1. the addition or the removal of components to equations that are needed only under certain state conditions and

2. structural changes

The addition or removal of components is necessary for example when additional forces start or stop acting in mechanical systems. As a concrete model we can use here again the bouncing ball. This time we also model the microscopic process of hitting, where the ball now may have a radius $r > 0$.

This process of hitting can – with some constraints – be seen as a spring and damper system. When the ball hits the ground the equation for $\ddot{x}$ is added up with the acceleration due to the spring force with spring constant k and the damping force with damping factor c divided by the mass of the ball m:

$$\ddot{x} = -g + k \cdot (r - x)\frac{1}{m} - c \cdot \dot{x}\frac{1}{m}$$

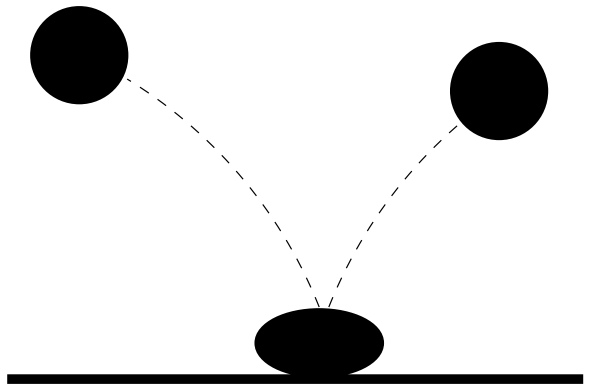


Figure 4.5: Bouncing ball with hitting process

This component is valid only when the ball touches the ground. Otherwise it is disabled.

Combining the equations we get the following system:

state equation:

$$\ddot{x} = -g + \begin{cases} 0 & \text{if condition 1} \\ k \cdot (r - x)\frac{1}{m} - c \cdot \dot{x}\frac{1}{m} & \text{if condition 2} \end{cases}$$

initial values

$$\begin{aligned} \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \\ \dot{y}(t_0) &= \dot{y}_0 \\ y(t_0) &= y_0 \end{aligned}$$

event condition function 1

$$(x - r) \geq 0$$

event condition function 2

$$(x - r) \leq 0$$

event: add or remove the component

An example for a structural change is the already mentioned looping pendulum. Here the dimension of the state space changes, too, but the solution is, nevertheless, continuous.

That is why I propose a classification consisting of four classes:

Class1 event: Change of parameters or input variables: e.g. discrete controller

Class2 event: Change of state variables: Change in one or more state variables and arbitrary parameters, e.g. bouncing ball - macroscopic, constrained pendulum

Class3 event: Change of components of equations: Depending on states, components are added or removed, e.g. bouncing ball - microscopic

Class4 event: Change of the structure: Conditional switching between different sets of equations, e.g. looping pendulum

4.3 Description

The description of state events in the model always has to consist of two parts: the description of changes that are caused by the event and the description of the condition that triggers the event.

As you may guess, simulation languages are far away from any standard concerning that theme. In order to illustrate this fact I give a comparison of the different ways of descriptions in the chapter "State Events in Simulators – State of the Art" later on.

It is quite common for the description of the condition to formulate an equation that has its zeros if the event has to be triggered. This *event condition* *function*[1] is an algebraic equation of state variables.

$$\Phi_i[t, x(t)] \qquad i = 1, \ldots, m$$

If $m > 1$ we can also combine these functions

$$\Phi[t, x(t)] = \Phi_1[t, x(t)]\Phi_2[t, x(t)]\Phi_3[t, x(t)] \ldots \Phi_m[t, x(t)]$$

but for numerical reasons it may be better not to combine more functions to a single function because this multiplication can lead to rapidly oscillating functions that are very difficult to handle.

Carver proposes in a paper on integrating over discontinuities [18] to transform event condition functions to additional differential equations:

$$\begin{aligned} \dot{x}_{n+1} &= \frac{d\Phi}{dt} \\ x_{n+1}(t_0) &= \Phi[t_0, x(t_0)] \end{aligned}$$

a method that has almost no evidence in commercial simulation languages.

As concrete examples and proposals for the description of events in simulation languages and also some models will be discussed later, I will return to the methodological level.

4.4 Handling

The process of handling events in a continuous system simulation run is divided into four computational stages:

1. Detecting the possible presence of an event

[1] I prefer this term to *discontinuity function* in order to use a more general expression; compare above: Events and Discontinuities.

2. Locating the event
3. Passing the event
4. Restarting the simulation

4.4.1 Detecting

The stage of detection is the most important of these four stages because all of the following methods and algorithms are only started when it is detected that in the recent interval a condition for an event is fulfilled. On the other hand, the process of detection has to be very cheap as it must – or should – be carried out at each step of the integration routine.

The simplest way to test if a condition of an event is fulfilled – here: if an event condition function crosses zero – is to check if the sign of the function has changed during the last interval.

Calling the beginning of the interval t_b and the end t_e we assume an event if

$$S_1 = \Phi_i(t_b)\Phi_i(t_e) \leq 0 \qquad 1 \leq i \leq m$$

It is easy to see that we can construct examples where this way of detection fails. E.g., if there are two zero crossings of the event condition function within this interval $[t_b, t_e]$, no event is announced.

A more sophisticated way is to include the first derivative of the event condition function

$$\dot{\Phi}_i = \frac{d\Phi_i}{dt}$$

and examine also

$$S_2 = \dot{\Phi}_i(t_b)\dot{\Phi}_i(t_e)$$

Now four situations arise. In case 1 we asume no zero crossing as the two values (at the beginning and at the end of the interval) are both postive or both negative and the gradients are both positive or both negative, too. Therefore, we think that the event condition function starting and ending in this interval at a positive (negative) value with positive (negative) gradients simply increases (decreases).

In case 2 the gradients are again both positive or both negative. But here the values at the ends of the interval have different signs. So the function can be assumed again as increasing or decreasing but – in contrast to above – due to the different signs of the values at the ends of the interval one zero crossing may be assumed.

Now, in case 3, the values at the ends of the interval have the same sign but the gradients have different signs. Therefore, we may take the function for crossing the time axis for two times.

In case 4 the values at the ends of the interval again have different signs and we can assume – without taking the gradients into account – one zero crossing.

case no.	S_1	S_2	assumed zero crossings
1	> 0	> 0	0
2	< 0	> 0	1
3	> 0	< 0	2
4	< 0	< 0	1

Here it is also very simple to show cases not covered by the four conditions. If $S_1 < 0$ then only the number of zero crossings can vary. Due to the continuity of the function there has to be at least one zero crossing. But there may also be three or five, ... A situation not covered by these cases is for example when no zero crossing is announced in case 1 but the function first increases then decreases, crosses zero, increases again and crosses zero for a second time. So we have two zero crossings but none is noticed.

Actually, this method of detection represents an interpolation-type method. We use the information (the values of the function and the values of the first derivative at the beginning and end of an interval) to make statements for its behaviour in between.

The argumentation is that due to the step-by-step nature of digital integration the values are only available at discrete values of t and that is why the precise instant at which the event occurred will never be known.

Another approach is given by methods that detect zeros in advance by extrapolation and reduce the steplength before a zero crossing is passed. The problem is that the extrapolation of the event condition function or of the system states leads to a whole series of problems and gives no reliable results. Therefore, the extrapolation algorithm would have to be strongly coupled to the actual problem and we will get no useful algorithm for general purpose simulation languages.

4.4.2 Locating

In the stage of locating we want to find the time when the event has to be triggered more accuratly. Obviously, this intention is restricted as we apply numerical algorithms and we are faced by accurracy constraints. In literature we find different approaches to locating but in practice there is only one type of methods employed, the *interpolation-type method.* Therefore I will confine myself to this type.

This method is a very simple routine to locate events and can be conjuncted with every general-purpose integration routine – in some applications a variable step method is required. The price for the simplicity is partly very serious problems that may falsify the following evaluations.

Here an event is detected by noting a change of sign in at least one of the event condition functions Φ_i in the evaluated interval.

The proposed subsequent actions and their main disadvantages are:

1. assume that the event occurred at the end of the actual interval

   The timing-error may seriously affect the following evaluations especially if more than one event has occurred. So there is a very small steplength necessary to locate the event sufficiently precise.

2. check the event condition function after each evaluation during an integration step and change the equations if necessary

   This proceeding may cause a large error estimation and force the algorithm to re-evaluate this step using a smaller steplength.

3. check the event condition function after the completed step; if there are events detected, repeat the evaluation with a shortened steplength so that the integration step ends at the event; interpolation gives an approach to that value of t

An example of a simulation language that uses the third method is MOSIS. Here the event condition function is evaluated after each integration step. If a change of sign is detected, an iteration process is started, the *modified regula falsi* iteration, to locate the zero crossing. In the next section I will describe this and other iteration processes for the location of roots.

As I have already mentioned above, it is quite possible that occurring events are missed if the procedure decides to search for an event only when a change of signs of the event condition function is referred on the boundary points of an interval. E.g., if two events (here two zero crossings) arise during an interval the algorithm described above will not react and continue as if nothing happened.

Nevertheless, in common simulation languages – provided that an event handling algorithm is available – the detecting is only done by noting a change of signs. If there is no event handling offered, then in most of the simulation languages the event gets triggered after the integration interval in which the condition became true.

Numerical Methods for the Location of Roots

For locating the zeros of the event condition function there are several different algorithms available. In this section I will describe some of the most used. Sometimes the algorithms get also applied in combinations, e.g. when numerical problems are coming up it is sometimes better to swap to a different algorithm or when it is necessary to calculate good starting values for more sensitive algorithms.

Assuming a continuous function $f : [a, b] \to \Re$ we know that the function takes every value between $f(a)$ and $f(b)$ in the interval $[a, b]$ at least once. Therefore, if the signs of the two values at the boundary of the interval are different and $f(a)f(b) \leq 0$, then we can conclude that there is at least one zero crossing in this interval, i.e. there is at least one value $c \in [a, b]$ with $f(c) = 0$ and c is named the *root* of the function.

An iteration process can now be started to locate the zero more accuratly.

Before going into details I want to remind that when these methods are applied in continuous simulation we do never know the event condition function exactly. And we have to rely on the discrete points calculated by the numerical integration algorithms, points that are "full" of numerical errors.

Continuous Binary Search

This method is, on the one hand, very simple but on the other hand it is one of the methods which guarantee convergence, no matter what the initial values look like. The only requirements are that they have different signs, so $f(t_a)f(t_b) \neq 0$ and the function f obviously has to be continuous but needs not necessarily to be differentiable.

Beginning with the interval $[t_a, t_b]$ we calculate the centre of the interval

$$t_c = \frac{t_a + t_b}{2}$$

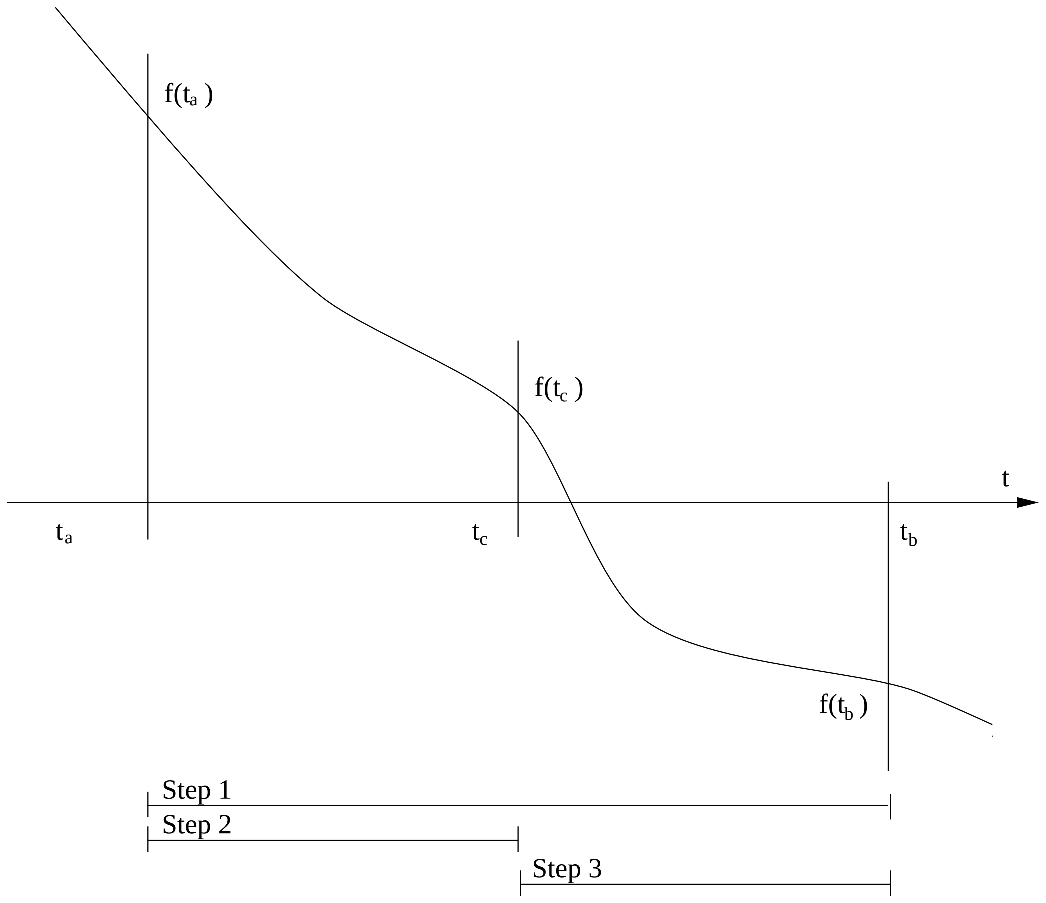


Figure 4.6: Continuous Binary Search

If $f(t_c) = 0$ we have succeeded and can stop here. If not, we check whether $f(t_a)f(t_c) < 0$. In this case we start the next iteration step by regarding the interval $[a, c]$. Otherwise $f(t_c)f(t_b) < 0$. Then we have to do an integration step with a reduced steplength to obtain $f(t_c)$ and continue with the interval $[c, b]$. So the zero is encirceled within an interval that gets smaller and smaller. The length of the interval after k steps is $\frac{t_b - t_a}{2^k}$.

Because of numerical reasons (limited length of the mantissa of machine numbers, rounding errors, ...) the length of the interval and therefore also the accuracy cannot be unlimitedly increased. Therefore, we have to give a condition when to stop the iteration.

Newton's Method

This method is founded on the fact that we can approximate a differentiable function by a linear function. Therefore, the function f also needs to be differentiable here at least in the neighbourhood of the zero.

Starting from the point $(t_0, f(t_0))$ we construct a tangent that we intersect with the time axis getting the next approximation point t_1. Thus, we get an iteration following

$$t_{n+1} = t_n - \frac{f(t_n)}{\dot{f}(t_n)}$$

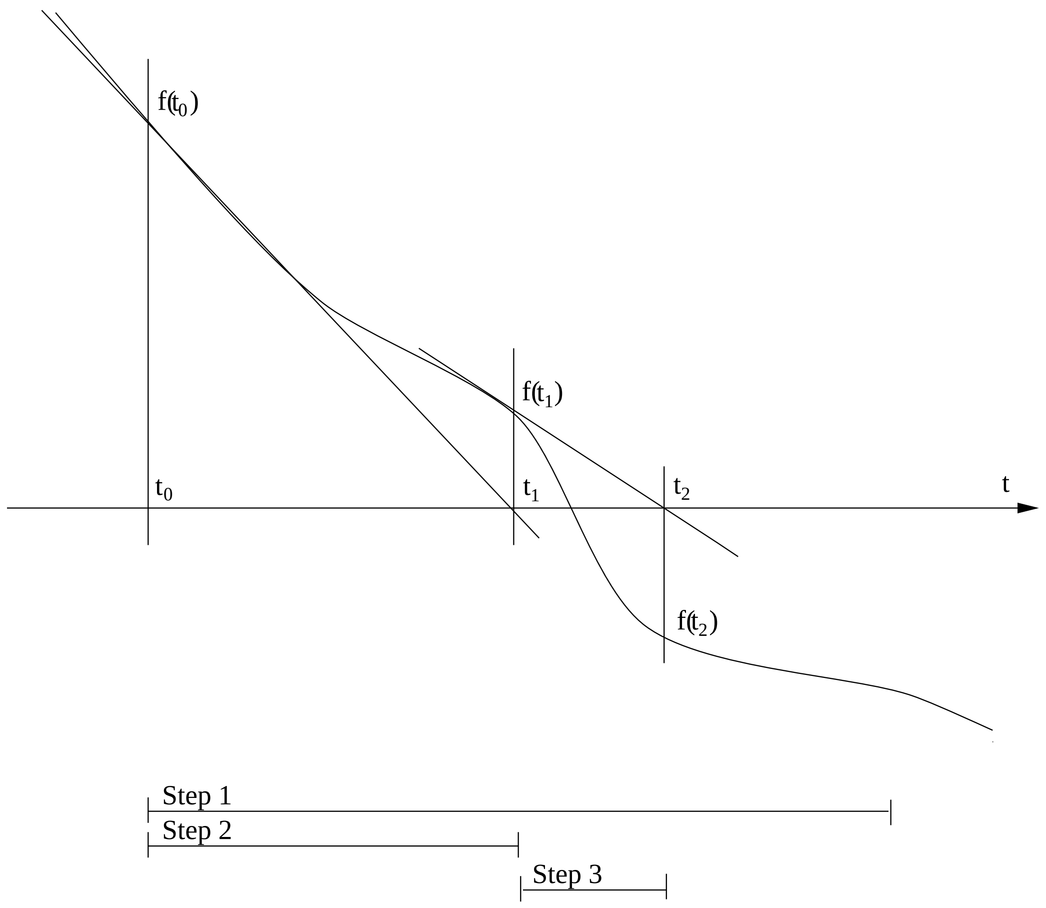


Figure 4.7: Newton's Method

A very strict requirement that this sequence converges is that we have to start *sufficiently* near to the zero. It is provable that there exists a domain of attraction but it cannot be given explicitly. An obviously bad starting point would for example be a point near an extremum of the function, in a section where the function is quite flat. In comparing with the Taylor series we can derive that the Newton's method converges for *good* starting values quadratically.

Regula Falsi – Method of False Position

The solution is approximated by a linear function between the boundary values of the actual interval. Starting with the interval $[t_0, t_1]$ we construct a line through the two points $(t_0, f(t_0))$ and $(t_1, f(t_1))$:

$$x = f(t_0) + \frac{f(t_1) - f(t_0)}{(t_1 - t_0)}(t - t_0)$$

Then we intersect this line with the t-axis getting

$$t_2 = -f(t_0)\frac{(t_1 - t_0)}{f(t_1) - f(t_0)} + t_0$$

Now we repeat the last integration interval with the reduced steplength $t_2 - t_0$ for getting the value of $f(t_2)$. If $f(t_2) = 0$ we have found the zero, otherwise $f(t_0)f(t_2) < 0$ or $f(t_1)f(t_2) < 0$ and we start the next step of the iteration with the interval $[t_0, t_2]$ or $[t_2, t_1]$, respectively.

4.4.3 Passing

After the locating procedure the dynamic calculations are stopped and the event is executed. That means that all the parameter changes, the changes of the state variables of components and structures are carried out.

When no state event handling is applied the passing gets a different meaning. In this case the changes are taken without synchronisation with the integration algorithm, and this can also happen during an integration step. For this

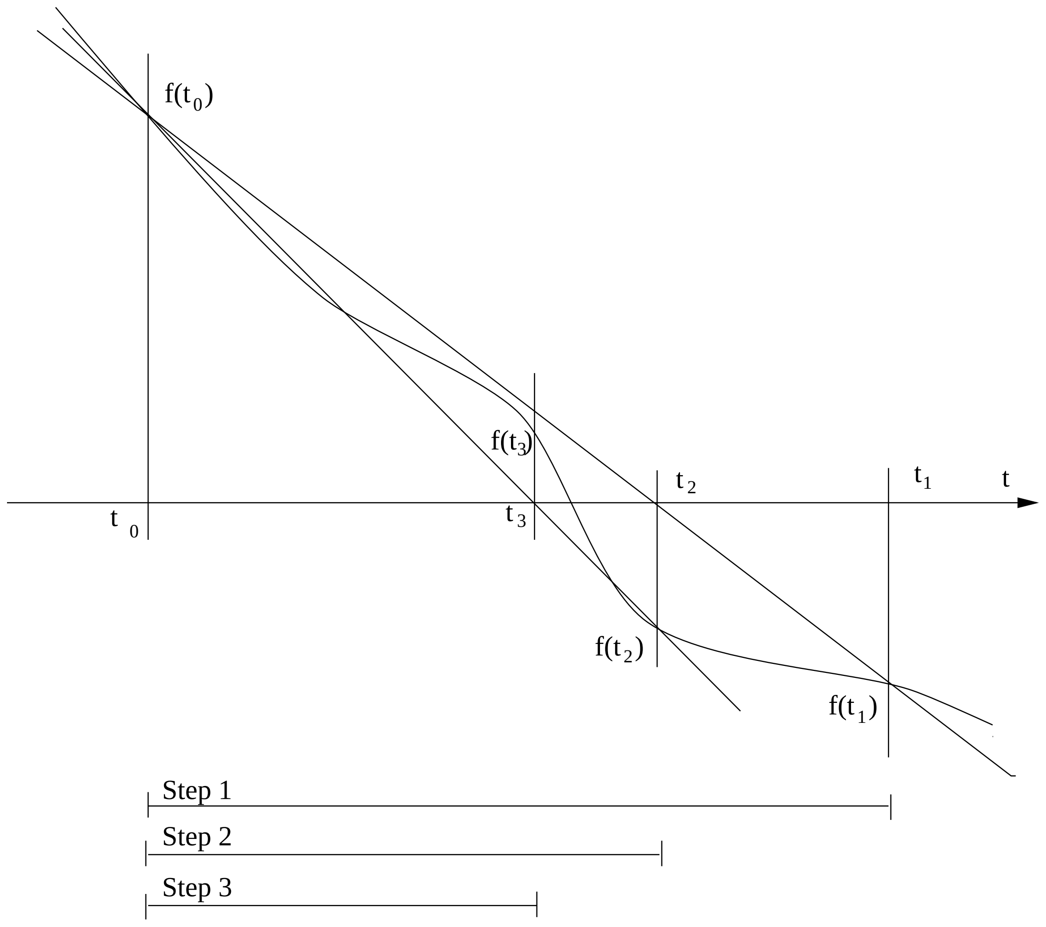


Figure 4.8: Regula Falsi

reason the local error reports a large value. In this case the last step – the step containing the event – is repeated until the local error is smaller than the tolerance or the minimum steplength is reached.

4.4.4 Restarting

When restarting the dynamic calculations after an event one has to take care that the condition for the event that we just passed may be still or again fulfilled. A chattering effect would then be inevitable.

Another task before restarting is to check if the actions of the just passed event cause another event to be triggered. In this case we first have to proceed this event and afterwards we have to check again if there are events to be triggered. Then the dynamic calculations are started again.

Chapter 5

State Events in Simulators – State of the Art

In this chapter I will analyse the possibilities of the description of state events and — when offered — the state event handling in different simulation languages. Obviously, my selection of simulation languages is not complete. I have chosen general purpose simulation languages for continuous systems that are in a way representative.

As not all of the presented simulators may be well-known I start every presentation with a short overview and will then focus on the state event possibilities. Finally, I demonstrate the "event features" of the simulator with the example "constrained pendulum". The software comparison "Comparison 7: Constrained Pendulum", published in *EUROSIM – Simulation News Europe*, the journal of the *Federation of European Simulation Societies* [16, starting with no.7, March 1993] makes use of this model and solutions are given here. This should illustrate the presented statements in a concrete model.

This comparison deals with the example of a simplified pendulum that hits a pin. I have already presented this model in section 4.2 in detail, therefore I only give a short description here.

The motion of a simplified pendulum can be described with a nonlinear ODE of second order. The position of the pin is given by φ_p. When the pendulum

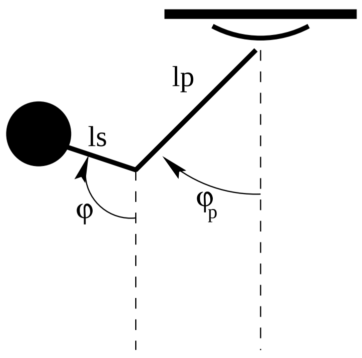


Figure 5.1: Constrained Pendulum

hits the pin, the actions we have to carry out are to change the length of the string and to change the actual velocity. When the pendulum leaves the pin again, we have to shorten the string and slow down the motion. The corresponding equations are

state equation

$$\ddot{\varphi} = -\frac{g}{l}\sin\varphi - \frac{d}{m}\dot{\varphi}$$

initial values

$$\begin{aligned} \dot{\varphi}(t_0) &= \dot{\varphi}_0 \\ \varphi(t_0) &= \varphi_0 \end{aligned}$$

event 1 condition function

$$\varphi \cdot \varphi_p - \varphi_p^2 \geq 0$$

event 1

$$\begin{aligned} l_{new} &= l - l_p \\ \dot{\varphi}_{new} &:= \dot{\varphi}\tfrac{l}{l-l_p} \end{aligned}$$

event 2 condition function

$$\varphi \cdot \varphi_p - \varphi_p^2 \leq 0$$

event 2

$$\begin{aligned} l_{new} &:= l \\ \dot{\varphi}_{new} &:= \dot{\varphi}\frac{l-l_p}{l} \end{aligned}$$

The values of the parameter and the initial values can be gathered from of the model descriptions and therefore I do not define them here.

The resulting motion of the pendulum, the angle and the angle velocity is shown in figure 5.2. It is the result of an experiment done with DYMOLA.

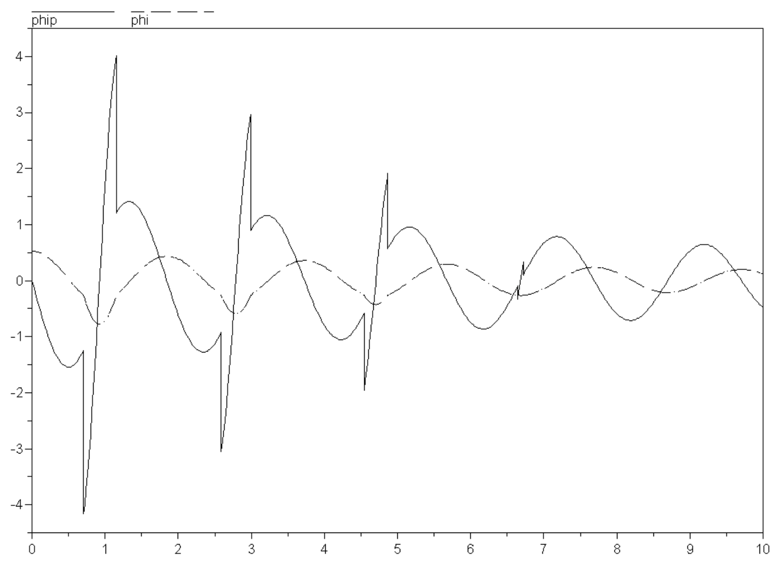


Figure 5.2: Motion of the Constrained Pendulum

5.1 ACSL

ACSL [2, 15] short for *Advanced Continuous Simulation Language*, is presumably one of the best-known simulation languages and is a typical example for an equation oriented language. The development of ACSL was started in 1974. The basic structures follow very closely the CSSL'67 standard, published in 1967 [61]. The main areas of application are time dependent, non-linear differential equations and also transfer functions. ACSL is based on FORTRAN 77 and is available for a lot of different computer architectures. The model description that can also contain operators and statements out of FORTRAN is first translated to a FORTRAN code and then compiled and linked. The experiments are then done at a special ACSL runtime level. At this level there are also functions available for e.g. linear analysis of the model (eigenvalues, steady states, ...) and plotting results. Detailed information can be found in the ACSL manual [2] and also in [15], a german book on simulation with ACSL with very detailed information. The following descriptions are based on Version 11.

The state event specification is done in ACSL with the `SCHEDULE` operator in the form

```
SCHEDULE db .XZ. ecf
```

That means that the `DISCRETE` block called `db` is executed when the event condition function called `ecf` crosses zero. The zero crossing can be specified by direction by using `.XP.` (crossing positive) or `.XN.` (crossing negative) instead of `.XZ.` (crossing zero). The `SCHEDULE` operators must be placed in a `DERIVATIVE` section.

Now let us take a glance on how it works during simulation. Before ACSL evaluates an integration step, all state variables are saved. After doing the step without regard to any events, the event condition functions are evaluated and tested if the sign changed using the following code:

```
IF(expr*expp .LT. 0 .AND. expp .NE. 0) GO TO event
```

where `expr` is the current expression and `expp` the previous value. That also means that when we start the integration step with an event condition function exactly equal zero, no event is signalled. If an event is found during the last integration step an iteration process according to *regula falsi* is getting started with the time borders of the last integration step, as starting interval. This iteration is continued until the interval has reached a minimum length ml evaluated as follows:

$$ml = max\{mint, epmx * li\}$$

where $mint$ is the minimum step size for the `DERIVATIVE` section ($1.0E-10$ by default), $epmx$ is a machine-dependent fractional multiplier on time (i.e. $1.0E-9$ on a 64 bit CRAY machine and $1.0E-6$ on 32 bit workstations) and li the left border of the actual interval

Another possibility to describe state events is to use the `IF-THEN-ELSE` statement. When using it, no state event handling will take place. When the condition becomes true the event is triggered. But when this is done, depends on the integration algorithm. As there is no synchronisation, the condition of the event is checked at the points of evaluation of the integration algorithm.

Numerical problems may arise so that it has to be checked whether the state change has already be done or not. If a step has to be repeated because of a too large error, the event could be triggered again and again. So we can state that this method is not suitable for step-size controlled integration algorithms.

Constrained Pendulum – ACSL

This model was written by Breitenecker [14]. As we can easily see the event is here described with the `SCHEDULE` operator and a single `DERIVATIVE` section. In this section we find the decision what action has to be taken by using a logical variable `swil` that denotes whether the pendulum leaves the pin or is just hitting.

```
PROGRAM Constrained Pendulum
LOGICAL swil, swnonlinear
   CONSTANT pi=3.141592654; pi6=pi/6;    !Calculate fractions of PI
   pi12=pi/12; mpi2=-pi/2;mpi6=-pi6;mpi12=-pi12; mpi24=-pi/24
```

```
INITIAL
   CONSTANT l=1; m=1.02; d=0.2; g=9.81; lp=0.7 !Pendulum parameters
   CONSTANT phi0=0.3; dphi0=0; phip=0.2     !Default initial values
                           ! Determine initial position of pendulum
   ls=l-lp; signphip=SIGN(1,phip); signphi0=SIGN(1,phi0)
   la=RSW((phi0-phip)*signphip .GE. 0.,ls,l)
   la=RSW(signphip .NE. signphi0,l,ls)
END !of INITIAL
DYNAMIC
   DERIVATIVE                         ! ---Dynamics of pendulum----
    phim=RSW(swnonlinear,SIN(phi),phi)         !Nonlinear or linear
    ddphi=-(g/la)*phim-(d/m)*dphi
    dphi=INTEG(ddphi,dphi0);
    phi=INTEG(dphi, phi0)
    SCHEDULE hit .XZ. (phi-phip);
   END !of DERIVATIVE
   DISCRETE HIT                      !Change of Velocity and Length
    swil=(phi-phip)*SIGN(1.,phip) .GE.0)  !Position before hit
    la=RSW(swil,ls,l); dphi=RSW(swil,dphi*l/ls,dphi*ls/l);
   END !of DISCRETE HIT
TERMT (t.GT.tend,'Stop on time limit')
END !of DYNAMIC
END !of PROGRAM
```

5.2 SIMULINK

SIMULINK [9, 10] is an extension to MATLAB [5, 6, 7]. The models in SIMULINK are defined with graphic block diagrams. The blocks are taken out of libraries and connected by lines. In these blocks the user can also include MATLAB functions. The experiments can be made either by using the SIMULINK menus or by entering commands in the MATLAB command window. Like in ACSL, special analysis tools are built-in. The versions we used for this thesis are MATLAB 4. 2c and SIMULINK 1. 3c. Some new features of SIMULINK 2. 0/MATLAB 5. 0 will be discussed later on.

5.2.1 SIMULINK 1. 3c

SIMULINK does not offer event handling directly. Events are "simulated" by means of "old and antique analog computational techniques". Consequently, in SIMULINK there are only a few blocks which allow to describe state events: in the NONLINEAR-Nonlinear Library:

- SWITCH-Block
- RESET-Integrator

in the SINKS-Signal Sinks Library:

- STOP-Block
- Hit-Crossing-Block

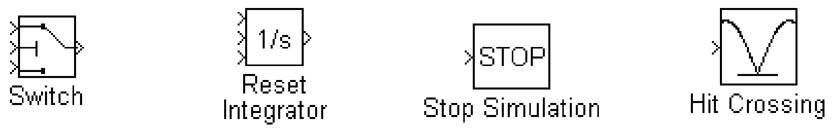


Figure 5.3: SIMULINK 1. 3c Blocks for handling state events

With more details:

SWITCH-Block

This block switches depending on the second input between the first and the third input.

RESET-Integrator

If the second input is equal to 0, the first input is integrated. If the second input is not equal 0, then the state will be reset to the third input.

STOP-Block

It stops the simulation when the input is $\neq 0$.

All these blocks are not synchronised with the integration algorithm. So there is no state event handling offered here. Therefore, it may happen very often that an event is totally missed. In order to avoid this missing there are three possible actions to take:

- to minimise the maximum steplength or the error tolerance – this leads to long computing times and is not effective as the algorithm is forced to achieve the high accuracy also in intervals where no events take place
- to save the data during the dynamic calculation and to do an interpolation by using the capabilities of MATLAB and starting the calculations again – a handmade state event handling, not really user-friendly
- to use the HIT-CROSSING-Block – this block forces the integration algorithm to decrease the steplength as it "Places discontinuity into integrator which slows the simulation down."[1] The events are produced by using the RESET-integrator where the result of the integration is fed back to the second input, the control input. This method is quite strange and I do not really recommend to use it. On closer examination we meet here the peculiar situation that the simulation language does not offer a numerical algorithm for state event handling but wants the user to add a "tricky" block to his model which violates extremely the integration algorithms.

Constrained Pendulum – SIMULINK 1. 3c

In this example we can see how difficult it is to build up WHEN events with more or less IF event blocks. So we had to combine them with a MEMORY-

[1]This declaration is shown when the block is unmasked.

Block and a RELATIONAL-Block. A model with explicitly defined parameters is shown in figure 5.4.

Besides, the logic of the event condition consists of more blocks than the description of the pendulum itself.

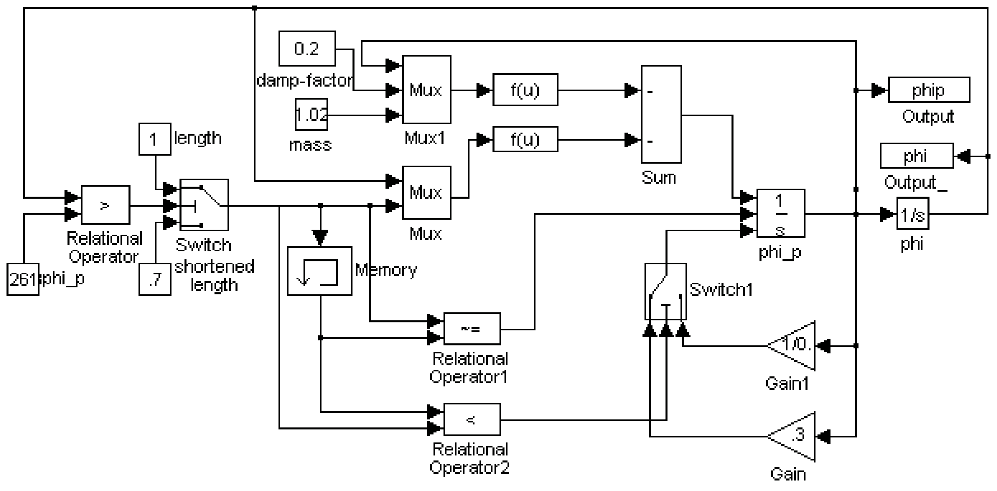


Figure 5.4: Constrained Pendulum – SIMULINK 1. 3c

5.2.2 SIMULINK 2. 0

In comparison with SIMULINK 1. 3c we meet in SIMULINK 2. 0 among other things some fundamental improvements and some new or enhanced blocks. The most important are shown in figure 5.5 and 5.6 and discussed here.

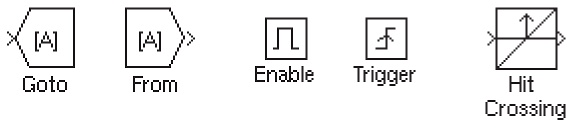


Figure 5.5: Some new or enhanced SIMULINK 2. 0 Blocks

- New Connections-Blocks were inserted, which may replace complicated connection lines: with the *Goto*-Block and the *From*-Block we can connect different blocks without using lines. The corresponding variables, which end in a Goto-Block, can be defined as *local* or *global*. According to this definition we can pick-up these variables in a subsystem or all over the model with a From-Block, where only the name of the variable has to be assigned. Therefore, the models can be designed more clearly.

- There are blocks, which have no inputs and outputs: Two special blocks for handling subsystems can be found in the Connections Library:

  - the *Enable*-Block
  - the *Trigger*-Block

  If one of these blocks or both of them are placed in a subsystem, then corresponding inputs will appear on top of the subsystem icon representing control inputs. The Enable-Block enables the subsystem, when the input signal crosses zero and while it remains positive. With the Trigger-Block the subsystem is executed once when the input signal crosses zero. The direction of the zero-crossing when the subsystem is triggered can be chosen as *raising*, *falling* or *either*. A combination of the two blocks assigns a subsystem that is triggered only if it is also enabled.

  The combination of Subsystem-Blocks and Enable or Trigger-Blocks results in a new type of block: a conditionally executed subsystem.

- The Integrator-Block includes and replaces the RESET-Integrator-Block as it offers a variety of settable parameters, like an *External reset* and an *Initial condition source*.

  The External reset can be chosen as *none*, *raising*, *falling* or *either*. Using the first option we get an ordinary Integrator-Block. With the other options we can assign a resettable integrator, where the direction of the zero-crossing that triggers the reset, can also be specified. Like in the version 1. 3c, no numerical algorithm will be started to locate the event more accurately.

The Initial condition source can be assigned as *internal* or *external*. Choosing an external source, an additional input will appear on the Integrator-Block.

The different icons of the Integrator-Block in SIMULINK 2. 0 can be seen in figure 5.6. Integrator1 is the ordinary Integrator-Block. Integrator2 shows the icon when the option *External reset: rising* and *Initial condition source: external* is chosen, when we change the External reset to *falling*, the result is Integrator3. Finally, Integrator4 shows the icon when the option *either* is used.

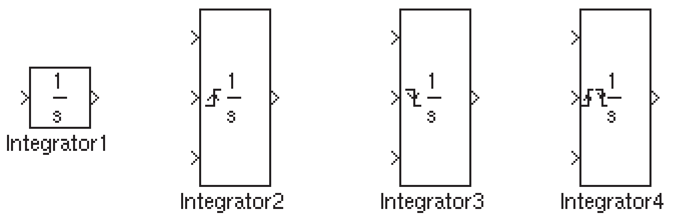


Figure 5.6: The Integrator-Block in SIMULINK 2. 0

- Also in SIMULINK 2. 0 we can find a *Hit-Crossing-Block*. We can assign different parameters: the *Hit crossing offset* and the *Hit crossing direction*. The output – if selected – is 1 when a hit crossing is detected and 0 otherwise. The Hit-Crossing-Block is synchronised with the integration algorithms. Therefore, this block is reasonable even if no output port is shown. The numerical algorithm for the detection is not documented.

Constrained Pendulum – SIMULINK 2. 0

The constrained pendulum can be modelled much more easier in the new version, SIMULINK 2. 0. This is mainly due to the zero-crossing capabilities.

Therefore, we could replace the complicated logic in the SIMULINK 1. 3c-model, figure 5.4, with conditionally executed subsystems in combination with a resettable Integrator-Block. The model is shown in figure 5.7.

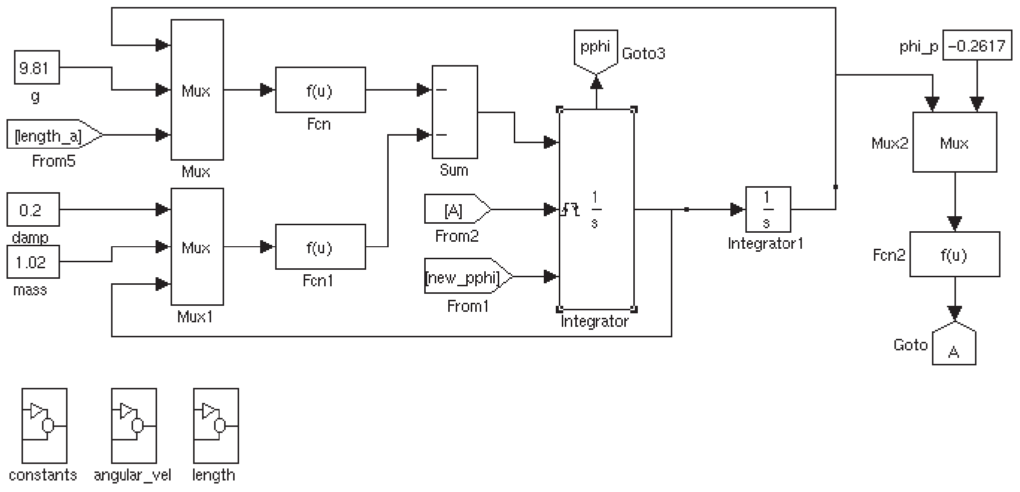


Figure 5.7: Constrained Pendulum – SIMULINK 2. 0

5.3 ESL

ESL [4], the *European Simulation Language* is again an equation-oriented language, based on FORTRAN 77. It was developed at the University of Salford, England, on behalf of the European Space Agency. In 1981 a specification of the CSSL'67 standard was worked out, the CSSL'81 standard, that never became a real standard as the involved groups did not come to an agreement. The main features of the proposal like modularity, separation of model and experiment, hierarchical structures, segments, . . . have then been realised in ESL since 1983. By the way, ESL allows the user to use ODEs of order higher than one.

State event handling is offered in ESL in a considerable efficiency.

The description of the state event condition is done in ESL by a relational statement between real variables with >, >=, <, <=. The event is assumed to occur when the relation changes from false to true or vice versa. The use of equality and inequality is not allowed in this statement as "the possibility of two quantities being exactly the same is remote" (cf [4, p.6.33]). So we can use here the usual state event condition function on one side of the relation and on the other simply 0. This anticipates the internal process that translates a given relation between A and B to

$$\Phi = A - B$$

Before explaining the algorithmic part of the event handling we have to take a glance at the error handling. The accuracy an event is handled with is state dependent controlled by `DISSERR` (default value: 0.0001). The band of tolerance is then defined for crossing from negative to positive with the interval

$$(0, DISSERR * max(A, B))$$

and for the other direction with the interval

$$(-DISSERR * max(A, B), 0)$$

If the relation is `>=` or `<=` then 0 is element of the first or the second interval, respectively.

When now a "zero"-crossing is detected a locating procedure is started. A line given by linear interpolation using the two values of Φ is intersected with the centre of the band of tolerance. This gives a new steplength. After recalculating the actual interval and Φ still not being within the band of tolerance, the locating procedure is repeated, from now on using quadratic interpolation.

The accuracy is restricted also by a minimum steplength *mint* depending on the computer precision (*eps*):

$$abs(t) * eps * 10$$

If t is smaller than the defined communication interval $CINT$ then $abs(t)$ is replaced by $abs(CINT)$ in the formula above.

The description of an event in a model can be made in three different ways. For correct handling it has to be included in the `DYNAMIC` region.

1. The description can be made with a logical assignment[2]

   ```
   L1:= A < B
   L2:= A* C >= 5.0
   ```

2. or with an `IF` clause

   ```
   y:= IF A < B THEN y1 ELSE_IF A * C >= 5.0 THEN y2
   ```

3. or by using a `WHEN` statement

   ```
   WHEN A < B THEN
        y := y1
   WHEN A * C >= 5.0 THEN
        y := y2
   END_WHEN
   ```

What are the differences between these three types? The first two statements work according to the level-triggering principle. The assignments are carried out every time the `DYNAMIC` section is executed.

The `WHEN` statement is only executed when the logical expression changes from false to true. So it represents an edge triggered statement. And only the `WHEN` statement starts the event handling. Another feature is that the `WHEN` statements are not getting sorted and the events are therefore handled in the sequence they are written in when it should happen that two or more occur at the same time/state.

[2] `A, B, C` are assumed to be real variables

Constrained Pendulum – ESL

As this example deals with an edge triggered event I describe the events with a `WHEN` statement. In the `INITIAL` section I initialise the actual length `la` by using an `IF` statement.

```
study
model cpendel(:=real: phi0,phip0);
constant real: g/9.81, pi/3.14159265
real: m, l, la, lp, pphi, d;

initial
-- parameters for pendulum
m:=1.02; l:=1.0; lp:=0.7; d:=0.2; pphi:=0.2;
-- initial values
phi:=phi0;
phi'=phip0;
-- determine initial length
la:=IF (phi*SIGN(pphi))>(pphi*SIGN(pphi)) THEN l-lp
    ELSE l;

dynamic
phi'':=-(g/la)*SIN(phi)-(d/m)*phi';

WHEN (phi*SIGN(pphi))<(pphi*SIGN(pphi)) THEN
       la:=l
       phi':=phi'*(l-lp)/l;
WHEN (phi*SIGN(pphi))>(pphi*SIGN(pphi)) THEN
       la:=l-lp;
       phi':=phi'*l/(l-lp);
END_WHEN;

communication
plot t, phi, phi', tstart, tfin, -5, 5;
```

```
end cpendel;

-- experiment
real: phi0/0.3, pphi0/0.0;
tstart:=0.; tfin:=10.; cint:=0.1; algo:=rk5;
cpendel(:=phi0, phip0);

end_study
```

5.4 MOSIS

MOSIS [58, 59], the *Modular Simulation Language*, was – and is still being – developed at the Department of Simulation Techniques at the Technical University Vienna, Austria. It is an all-purpose compiling language following the CSSL'67 standard. MOSIS is based on C and offers special features for modular development of simulation models and parallelisation on MIMD multiprocessor systems with distributed memory. But it can also be used on single-processor systems. The models are built up in an equation oriented language, then the description has to be translated, compiled and linked. Before starting an experiment in the runtime command window, an instance of the model has to be created, which then can be executed.

The state event handling facilities are similar to those of ACSL. In MOSIS we find a statement for this purpose containing several parameters:

`sevent(condition, crossing-type, discrete block, enable`.

The `discrete block` consists of the actions that are carried out when the event gets triggered, the `condition` is a logical expression. There are three different values for the `crossing-type`:

type =	0	detection of crossings from negative (< 0) to positive (≥ 0)
	1	of crossings from positive to negative
	2	of general zero crossings

When the value of the condition function is equal to zero at the beginning of an integration interval, there will be no event triggered as no zero crossing occurrs.

The parameter `enable` enables or disables the `sevent` statement. If it is equal to zero the event is disabled and will not be handled.

After the detection of a zero crossing (simply by noticing a change of signs) a *modified regula falsi* iteration process is started. This iteration will proceed until the length of the interval containing the state event is smaller than the parameter `severr`. The default value for `severr` is $1E - 8$

The difference to the *regula falsi* is a precaution of numerical problems. When at the ends of the interval the condition function results in strongly different absolute values, the regula falsi will get into troubles. Due to the limited length of the mantissa of machine numbers the sum of the two values can happen to be equal to the bigger value. In this case, the regula falsi will fail and therefore MOSIS switches to continuous binary search. This method is slower but will work correctly.

Constrained Pendulum – MOSIS

In this model, written by Schuster and Breitenecker [60], we find an application of the `sevent` statement. The description is very similar to that in ACSL.

```
ccode
{ double sign(double x1,double x2)
  { if(x2>0) return x1; else if(x2<0) return -x1;
else return 0; } }
model pendulum() {
 const PI=3.1415926535;
 double l=1.0,m=1.02,d=0.2,g=9.81;
 double phi0=0.3,dphi0=0,phip=0.2,lp=0.7;
 double ddphi,la,ls,signphip,signphi0;
 int swil,iter,linear; //iteration, model type
 state phi,dphi;
 double sign(double,double);
 preinitial{iter=linear=0;ialg=3; tend=10;}
 initial {  ls=l - lp;
  signphip=sign(1,phip); signphi0=sign(1,phi0);
  la=((phi0-phip)*signphip >=0)?ls:l;
  la=(signphip!=signphi0)?l:ls;              }
 dynamic {
  derivative { // "linear" determines model type
   dphi‘=(-g/la)*(linear?phi:sin(phi))-
        - (d/m)* dphi, dphi0;
   phi‘=dphi,phi0;                            }
   sevent(phi-phip,2,hit); // state event sched.
   terminate(iter && (dphi>=0));              }
  discrete hit {
    printf("state event hit: t=%.15g\n",t);
    swil=((phi-phip)*sign(1,phip)>=0);
    la=swil?ls:l;
    dphi=swil?(dphi*l/ls):(dphi*ls/l);   }  }
```

5.5 SIMNON

SIMNON [8, 23], *Simulation of Non-linear Systems* was developed at the Department of Automatic Control, Lund Institute of Technology, Sweden, for solving differential and difference equations and for the simulation of dynamic systems.

SIMNON is one of the few languages that do not follow the proposals of the CSSL'67 standard and that are commercially successful. The model description can be made with modules, thus complex systems can be decomposed and therefore the description becomes very clear. The model is then built up as an interconnection of these subsystems that consist of differential or difference equations.

When a system gets activated, SIMNON translates the statements into a Pseudo-code for the "SIMNON machine", a hypothetical computer. When everything is correct, this code is sorted and is finally translated into machine code. This code allows a very efficient simulation.

There are three different systems for the model description:

1. *continuous system* for differential equations
2. *discrete system* for difference equations
3. *connecting system* for describing the interconnections between the systems

Experiments can be made by entering the corresponding commands in the command dialog window. These commands can also be packed together into a *macro*.

For handling state events SIMNON is offering only few possibilities and no numerical algorithms. There are only two statements that allow the description of state events in the model, an `IF-THEN-ELSE` statement and the conditional termination `CTERM`. The `IF` statement is used in the equations like in the following example:

```
dx = IF x > x1 THEN equ1 ELSE equ2
```

As there are no numerical algorithms to handle events the calculations are continued until the condition becomes true and then the actions are carried out.

A nice possibility for describing state events is given by the combination of the `CTERM` statement with an experiment macro. When the condition for the conditional termination is fulfilled, the simulation is stopped and the further action can be defined at the experiment level and being automatically controlled with a macro.In a macro we can describe complex experiments with the command language that permits the activation of several systems for doing sequential runs. It is also possible to do some evaluations on this level in order to initialise the next experiment with parameters depending on the final values of the results of the last run.

In principle, here two models which are terminated conditionally are used. At experiment level it is switched between these models in sequential manner.

The same technique can be used in MATLAB/SIMULINK where at MATLAB (= experiment level) we can switch between models and calculate the events in MATLAB itself (this method is only necessary in case of more complex events).

ACSL offers also a runtime environment, *ACSL MATH* [3] where the same strategy can be used – but it is more efficient to use the state event handler.

ACSL MATH is an environment that allows to start ACSL models. As there is a big variety of mathematical functions provided and it is possible to start experiments, we can use it for running models sequentially. In between of the runs we can calculate the new parameters and initial values out of the final values of the previous run. The commands can also be put together in a script file.

A constraint is given as we may load only one model at a time.

Constrained Pendulum – SIMNON

I built up this model by a combination of a macro and `CTERM` statements. The conditions for the event can be found at the end of the model. As I have to formulate the condition for being handled of this level triggered statement, I introduced a logical variable `hit`. If `hit` is true, the pendulum has hit the pin and so it is clear what actions have to be taken when the simulation is stopped.

The macro consists of two main parts, firstly, the activating of the system, assignment of the parameters and a first simulation run.

In the second part there is the simulation loop, the model `conpend` gets reinitialised with conditions depending on the actual state and the simulation is restarted.

```
CONTINUOUS SYSTEM conpend
STATE phi phip
DER dphi dphip
TIME t
dphi=phip
dphip=-g*sin(phi)/l-d*phip/m

hit : 0
pphi:-0.2618
g   : 9.81
l   : 1
m   : 1.02
d   : 0.2

st1=if pphi > 0 and not hit then cterm(phi > pphi) ELSE 2
st2=if pphi > 0 and     hit then cterm(phi < pphi) ELSE 3
st3=if pphi < 0 and not hit then cterm(phi < pphi) ELSE 4
st4=if pphi < 0 and     hit then cterm(phi > pphi) ELSE 5
END

MACRO MACPS
syst ps
init phi  :  0.5236
init phip :  0
par  pphi : -0.2618
par  d    :  0.2
par  l    :  1

plot phi phip
simu 0 10 0.001

label restart
```

```
disp phi/dphi phip/dphip t/Time
disp st1/t1 st2/t2 st3/t3 st4/t4
write 'restart'
if t1. eq 1 goto hit
if t3. eq 1 goto hit
if t2. eq 1 goto leave
if t4. eq 1 goto leave

label hit
  let newphip.=dphip./0.3
  par  l    : 0.3
  par  hit  : 1
  init phi  : dphi.
  init phip : newphip.
  simu Time. 10
goto cont

label leave
 let newphip.=dphip.*0.3
 par  l    : 1
 par  hit  : 0
 init phi  : dphi.
 init phip : newphip.
 simu Time. 10

label cont
if Time. LT 10 goto restart
end
```

5.6 SLIM

SLIM [41] stands for *Simulation Language for Introductory Modelling* and was developed by D.J. Murray-Smith at the Department of Electronics and

Electrical Engineering, University Glasgow, Scotland. It has been designed as an inexpensive teaching tool intending to provide an introduction to some principles of continuous system simulation. SLIM is again a simulation language following the CSSL'67 standard and is based on FORTRAN 77. In contrast to ACSL it does not compile the model but translates and interprets it. Although it was developed for the use on PC/DOS based environments it has already been ported to other operating systems without difficulty.

The description of a model is made in an ASCII text using a manner similar to that of FORTRAN. For state events we can find no special handling tools and therefore we are forced to program especially the state changes "directly".

A common solution is the use of the `IF` statement for checking the event condition function. The event itself is "programmed" by some tricky jumping in procedural code in order to change parameters, states,... The following line shows an example of an `IF` statement with the meaning: if the arithmetic expression is negative then go to label 10, if it is zero go to label 20 and if it is positive then go to label 30.

```
IF (A-B)10,20,30
```

Used in a `DERIVATIVE` section the `IF` statement is evaluated only after a successfully finished integration step.

These labels can also be used for reinitialising the model and starting experiments from within the model or also for conditional termination.

Constrained Pendulum – SLIM

The model was written by Murray-Smith [42]. Due to the labeling the model is not really easy to read. The logic for the event occupies much more lines than the description of the dynamics. As we can see, there are several auxiliary variables necessary for this conditional jumping for indicating the different states.

```
C  Setting flags to indicate phase of motion
      IF(X10-PHIP)4,4,2
2     MARK=-1
      MARK1=-1
      AL=ALI
      GOTO 5
4     MARK=2
      MARK1=2
      AL=ALS
5     CONTINUE
7     T=T1
      DYNAMIC
        DERIVATIVE
          DERIV1=X2
          DERIV2=-(G/AL)*SIN(X1)-(D/AM)*X2
          X1=INTEG(DERIV1,X10)
          X2=INTEG(DERIV2,X20)
C  Check whether angle of string has reached
C  critical angle PHIP
          IF(X1-PHIP)20,20,30
C  Applies where the length is ALS (short)
20        MARK=2
          GOTO 32
C  Applies where the length is ALI (long)
30        MARK=-1
32      DERIVATIVE END
C  Output T, X1 and X2 to file (and screen)
          TYPE T,X1,X2
C  Test for time reaching TMAX
        IF(T-TMAX)33,33,12
C  Test whether pendulum has reached the
C  critical angle PHIP from either direction
33      IF(MARK-MARK1)40,37,35
C  Applies if the pendulum shortens
```

```
35       T1=T
         X20=X2*ALI/ALS
         MARK1=2
         MARK=2
         X10=PHIP
         AL=ALS
         GOTO 45
C  Applies if the pendulum lengthens
40       T1=T
         X20=X2*ALS/ALI
         MARK1=-1
         MARK=-1
         X10=PHIP
         AL=ALI
         GO TO 45
37      DYNAMIC END
45      GOTO 7
12      STOP
        END
```

5.7 ANA 2.x

ANA 2. x – we used ANA 2. 11-95 – [31] is a simulation package for system analysis, simulation of linear, non-linear and switching systems but also for dealing with fuzzy and neural networks systems. It was developed at the Technical University Vienna, Austria, at the Institute for Electrical Control Engineering.

ANA 2. x is a graphical block-oriented language. The built-in libraries are especially designed for electrical control systems. But the user can also program his own blocks in the model description language *ANAmdl* and expand the libraries.

ANA 2. x offers for the description of state events the `SWITCH` statement. Here we can define several `CASES`, states the system can take. The condition and the corresponding action get defined by using the `ONRISE - DO` statement. We have to note that this statement defines the condition for changing the state and not for remaining in it. The actions are carried out when the condition becomes true. The actions are put into a `PROCEDURE` section that gets called.

In the built-in libraries we can find a graphical `MULTIPLEXER` block that uses this construct. The block has three inports, `u1, u2, u3` and one outport `y`. Depending on the third input which is compared with a threshold parameter `k` either the first or the second input is passed to the outport. This behaviour is decribed by

```
y = u1 : abs(u3) > k
y = u2 : abs(u3) <= k
```

During the simulation the `ONRISE` condition gets permanently checked. Before an event gets triggered, i.e. a `PROCEDURE` section is proceeded, the event gets located more accurately by a continuous binary search iteration process.

I can subsume that the applied method is similar to the `SCHEDULE` statement and the `DISCRETE` sections of ACSL and offers good possibilities for describing state events.

Constrained Pendulum – ANA 2.x

This template block was written by Goldynia [32]. The events are described with `ONRISE` statements and two `PROCEDURE` sections.Like in ACSL, this example can be described very easily in ANA 2. x.

```
BLOCK COMP7;
OUTPUT
 Phi "[rad] angle";
 Omega "[rad/s] angular velocity";
PARAMETER
 l = 1 "[m] length";
 m = 1.02 "[kg] mass";
 d = 0.2 "[kg/s/rad] damping";
 Phi0 = PI/6 "[rad] start angle";
 Omega0 = 0 "[rad/s] start angular velocity";
 lp = 0.7 "[m] distance of pin";
 phip = -PI/12 "[rad] angle of pin";
STATE
 phi "[rad] angle";
 omega "[rad/s] angular velocity";
VAR
 g DISCRETE "gravity";
 ls DISCRETE "shortened length";
 la "actual length";
 state DISCRETE;
SIM
 SWITCH state
 CASE 1:  la = l;  ONRISE phi < phip DO SetState2;
 CASE 2:  la = ls; ONRISE phi >= phip DO SetState1;
 ENDSWITCH
 phi .= omega; omega .= -g/la*SIN(phi) -d/m*omega;
 Phi = phi; Omega = omega;
ENDSIM
PROCEDURE SetState2; // shorten the length
 state = 2; omega = omega*l/ls; STORE ALL;
ENDPROCEDURE
PROCEDURE SetState1; // enlarge the length
 state = 1; omega = omega*ls/l; STORE ALL;
ENDPROCEDURE
INIT
   g = 9.81; ls = l - lp; state = 1;
   phi = Phi0; omega = Omega0;
```

```
ENDINIT
ENDBLOCK COMP7;
```

5.8 DYMOLA

DYMOLA [25], the *Dynamic Modeling Language* is an equation-based object-oriented modelling environment. The equations need not be formulated explicitly. This is done by a symbolic model transformation. Therefore, the same model can be used in different systems with different causalities. Large models may be decomposed hierarchically into submodels that are connected via `cuts`. Once the model is written, the user can decide in what format his description should be put out.

DYMOLA itself is a modelling language using symbolic methods for deriving the mathematical model. DYMOLA's output are model descriptions in some languages. DYMOLA supports now ACSL, DESIRE, SIMULINK, SIMNON and obviously DYMOSIM, the genuine simulator of DYMOLA.

The description of state events in DYMOLA can be made in different ways. The first possibility is to use `if` statements that are really recognised as events and then can be correctly translated. In case the event condition function is zero, a special algorithm is built in to detect also this special "crossing". When the function then leaves zero, no zero crossing will be announced. In this case, DYMOLA shifts the thresholds by a small value in the order of the smallest machine number; if there is a zero crossing in the positive direction, then the shift is also in the positive direction and vice versa. The `if` statement cannot be used for changing the states.

Changes of the model structure and jumps of state variables can be described by using the `when - then - endwhen` statement, so-called *instantaneous equations*. Like in ESL, this statement gets evaluated only when the condition becomes true.

In the `when` construct we may reassign state variables with an `init` statement.

This special statement is necessary as every variable may be assigned only once in a model.

When we now choose ACSL as the target language, all these constructs would be translated to `SCHEDULE` statements and DISCRETE sections.

Constrained Pendulum – DYMOLA

In this description I used both constructs of DYMOLA for events: an `if` statement and a `when` equation since the initial length of the string of the pendulum has to be determined and then – during the simulation run – the change between the two lengths is also taken over by this statement. The `when` equations are necessary for changing the state variable.

The usage of the `when` equation in this example is not so easy. The problem is that in DYMOLA the fact that every variable may only be assigned once holds also for the `init` statement. Here the velocity has to be reassigned after each of the two possible events hitting or leaving the pin. So I introduced the auxiliary variables `a` and `b` which notice a change of the length and trigger the reassignment.

```
model cpend {* (-100, -100) (100, 100)}
  {* window 0.35 0.04 0.59 0.73 }
  parameter l=1 lp=0.7 g=9.81 m=1.02 d=0.2 pphi=-0.2
  local phi= 0.3 phip=0. la a=0. b=0. neu1=0. neu=0.
    la = if phi*(pphi) > pphi**2 then l-lp else l
    der(phip)*m*la+(g*m)*sin(phi)+(d*la)*phip=0
    der(phi) = phip
    b=la
when phi*(pphi) < pphi**2 then
      neu = phip*(l-lp)/l
endwhen
when phi*(pphi) > pphi**2 then
      neu1=phip*l/(l-lp)
endwhen
```

```
when a<b or b<a then
init(phip)= if phi*(pphi) > pphi**2 then neu1 ELSE neu
new(a)=la
endwhen
end
```

5.9 ModelMaker

ModelMaker is a block-oriented simulation language for PC for solving algebraic equations and first order ODEs. ModelMaker has a very capable event block that offers a lot of features. After entering the state event condition, one can also enter a tolerance within which the event is triggered. Then the actions that are to be taken are given. What is most surprising here is the way the event itself can be handled: there are buttons to set the event active or inactive when the simulation run starts. When it is set inactive, another event can activate it. Another decision has to be taken whether the event should be reactivated after it was triggered. With that button one can switch between edge and level triggered events as we have mentioned it above.

Another possibility to describe events is given by the *conditional definition* of *compartment, variable, defined value and flow* blocks. The unconditional definition is also the default value but we can formulate several conditions and corresponding equations that are then evaluated dependent on which condition is true. If more conditions are true at the same time, then the first in the list becomes valid and the corresponding equation is taken for the evaluations.

On the algorithmic side it looks quite different. The applied algorithms often do not trigger the event even if it should be recognised. It seems that the corresponding state event handling algorithms are not implemented well.

Constrained Pendulum – ModelMaker

On the right side of the graphical representation of the model (figure 5.8) we can find the model parameters, like mass, length of the string, ... as *defined values*. The two rectangles at the top are *compartments* that contain the two first order ODEs of the pendulum equation. The two circles at the bottom represent the state events, *component events*, one for hitting the pin, the other for leaving it. Getting the information for triggering from the *compartment* x, the position, they are linked to the velocity y and can therefore change it. The actual length of the string can be changed via the links to the *defined value ls*, short for "length of the string". And *ls* transfers the value to the *compartment y*.

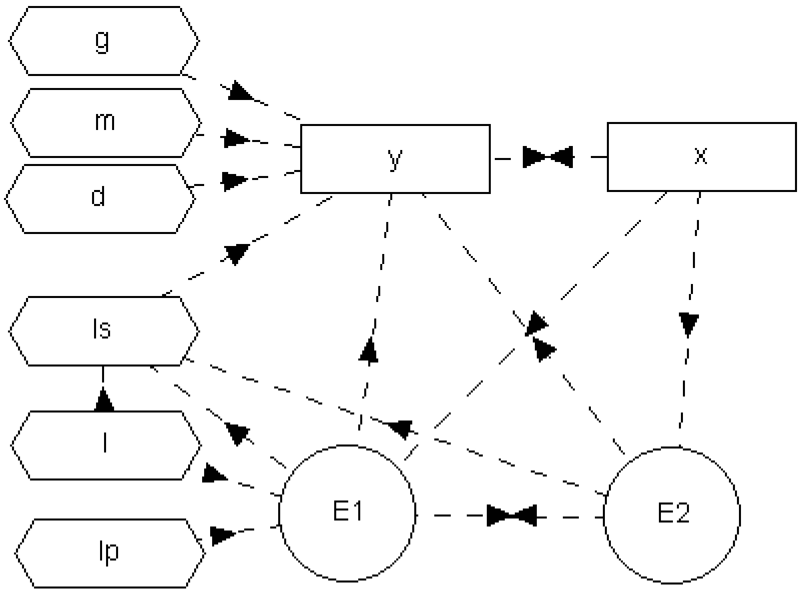


Figure 5.8: The Constrained Pendulum – ModelMaker

Chapter 6

Event Methodology – Description of Methods

In chapter 4 events were classified due to their nature. Chapter 5 has shown that in various simulators different methods are used in describing these event classes.

In general, there are a lot of possible approaches for modelling systems with state events. In order to characterise the approaches which are partly "well constructed" and partly "grown" I worked out six different methods that I will present and discuss in this chapter.

After the description of these methods I will discuss their nature from the viewpoint of simulation methods, from the aspect of model development and also from a numerical point of view.

6.1 "All-in-one" - Method

In the *All-in-one method* we describe the whole system in a single model, a large monolithic block. We put everything into this only description, all the equations, the event condition functions, the events themselves, ...

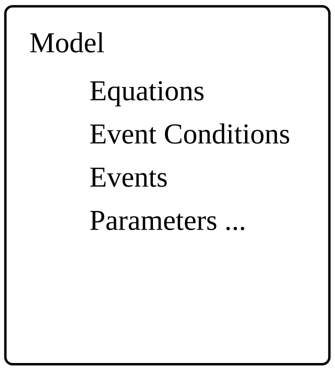


Figure 6.1: The All-in-one Method

Aspects of Simulation Methodology

From the simulation methodological point of view we are in some sort of a dilemma. On the one hand, this method represents the most natural one. Starting from a real system we represent it in one single model. The simulated system is therefore seen as an indivisible unit that is represented in one model.

This causes on the other side a quite considerable problem. The model is valid only for a certain real system. Therefore, it is almost not possible to reuse the model for simulating other systems.

In addition, it is very confusing to ”read” these models as everything gets mixed up in the description. It is very difficult to extract the logical structure of such a model and also the accompanying equations.

Aspects of Model Development

From the modelling point of view the all-in-one method is a very difficult task to do. The model must contain the system description that – if we have a structural variable system to simulate – consists of at least two different sets of model equations.

The description of the event condition often leads to a very sophisticated logical structure. Especially in block-oriented simulation languages the description of the logical structure can be bigger than the model itself. This fact can easily be verified.

The description of the event leads to another problem. For modelling structural variable systems there are different possibilities to deal with the state variables: on the one hand, we can initialise the state space with the highest dimension occurring in the model and use only that amount of state variables that we need for the moment. On the other hand, we can use two or more state variable sets in parallel. During simulation we have to swap to the actual valid one and force the others to be "silent".

The requirements to the simulation language in terms of the system description are not very high. We need logical statements that allow us to define the logical structure and express the event condition. We need facilities that allow us to swap between different sets of equations or to change components of the equations. Furthermore, there must be access to the state variables and parameters to redefine them during the runs.

Numerical Aspects

The requirements to the numerical algorithms of the simulation language are quite high. We require for good simulation results that the numerical algorithms are working closely together with all the elements used in the model description. This means especially the logical statements that indicate the existence of a state event. In order to achieve a good simulation result, the numerical algorithms should react on the existence of these logical statements. If this is not the case we have to minimise the step size from the beginning of the calculations in order to achieve a certain accuracy or we have to add special integration control algorithms.

Conclusion

In terms of clarity of the simulated model this method is not really capable. The consequence is therefore a bad maintainability. Furthermore, there is no support to reuse the model. Finally, we meet here a difficult numerical situation concerning the synchronisation of the integration algorithms and the state events.

6.2 "Discrete Section" - Method

In the *Discrete Section method* the description of the condition of the event and finite jumps of parameters and variables are separated from the rest of the model description. The condition is described with separate statements and the event is becoming a structural element.

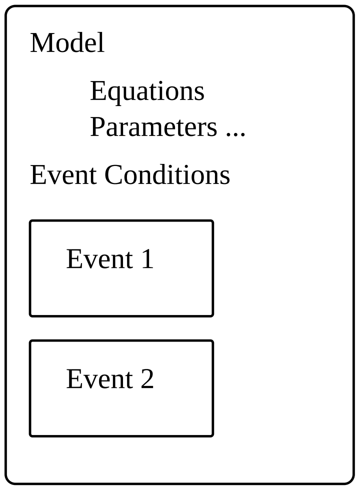


Figure 6.2: Discrete Sections

Aspects of Simulation Methodology

In this method we notice a parallel to the process of abstraction when modelling a real system. The event description is here – at least partly – separated from the system description in its own structural elements. In other words, we can separate state events of the class one and two of the four classes approach completely from the system description. The events of the classes three and four must remain – like in the method above – in the model description and there cause the same problems.

Aspects of Model Development

A step towards structured modelling is presented in this method. Here we describe the event condition in separated structural components. In order to activate them there is an explicit event condition description. Further, we meet a separated description of events of class one and two. These are the types of events defined as discrete changes of parameters and state variables. Accordingly, these discrete changes are moved to discrete sections, a term that reminds us of CSSLs. And in fact we find this method mostly in languages of this type. The corresponding blocks in modern block oriented languages like SCICOS [43] are "zerocrossing blocks" for the condition and event driven blocks like the "selector blocks" and resetable integrators for the event description.

Numerical Aspects

Due to the description in separate elements it is possible to apply special numerical algorithms for the event finding and for synchronisation. We can call this proceeding *event handling.*

Conclusion

What is remarkable here is the step towards structured programming as we use special statements for the event condition description that activate the corresponding event. This method is designed especially for class one and two events. Other types have to be handled like before or they are transformed to parameter changes. When dealing with changes of components or with structural changes we can describe the system with a single system of equations, where parts of it are enabled or disabled by parameter changes like in the following equation:

$$\dot{x} = a \cdot f(t,x) + b \cdot g(t,x)$$
$$a, b \in \{0,1\}$$

From the modelling point of view, this is a falsification of the "real" situation and should only be applied when there are no other possibilities.

6.3 "Concatenated Runs" - Method

With the *Concatenated Runs method* or *Stop'n'Go method* we describe a method that stops the dynamic calculations, reinitialises certain parameters and starts the calculations again.

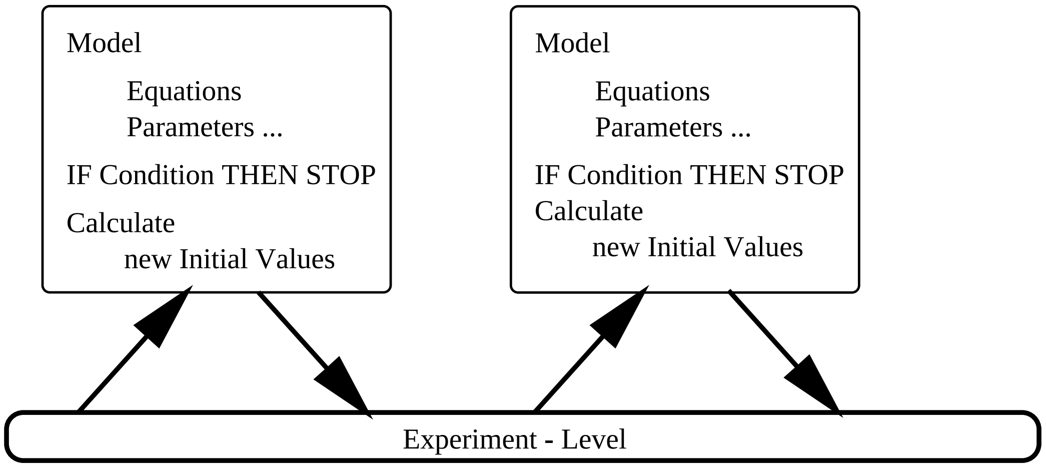


Figure 6.3: The Stop'n'Go Method for two runs

Aspects of Simulation Methodology

To be precise, the Stop'n'Go method starts new experiments and those sequential experiments sum up to the one *big* experiment of the system we are simulating. The event condition here is translated in a conditional termination statement. The event itself is again separated; here it is not only separated from the rest of the model description but also separated from the simulation runs as the events are passed in between of two runs.

The events can therefore also be placed on the experiment level. This method is again especially applicable for events of class two and three. The events of these classes are completely separated here. We can say that the (real) process that we replaced by an event in this method is represented as an interruption of the process followed by the jump in a parameter or state variable.

Aspects of Model Development

Due to the separation it is easier to concentrate on the model description in the narrow sense. The description of the condition is reduced to the conditional termination statement. If more than one event has to be included we have to give some additional information which condition stopped the actual run or we have to define more locations in the model description from that the model is restarted so that the correct consequent actions – the events – are worked off. On the other side, we have to be aware that here the model description and the simulation control – like starting an experiment – is mixed up.

Numerical Aspects

The simulation program has to assure that these termination statements are triggered with sufficient accuracy. For the solution of the (series of) initial value problems we need no additional numerical features.

Conclusion

This method is again especially applicable for the class two events and class three events and is a extension of the Discrete Section method. When the event is described on the experiment level we can say that the experiment takes over some parts of the model description. This idea is developed further on in the next chapter.

6.4 "Component Exchange" - Method

In the *Component Exchange method* an event is passed by exchanging components of the model.

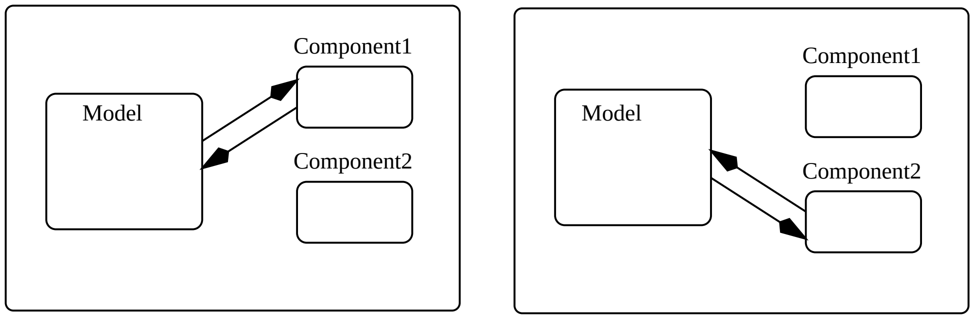


Figure 6.4: Component Exchange

Aspects of Simulation Methodology

This method marks a first step towards modular modelling. In contrast to the Discrete Section method described above the events of class 3 and 4 here get separated from the model description. The event condition obviously has to be checked permanently but there are two different methods for applying the results.

At first there is the permant event method. The condition is checked and decides which component to take. The other reminds to the *Discrete Section method*: here the event condition switches the component "on" or "off" via e.g. a parameter. Therefore, the latter method is also something like a type shift method as it transfers an event of class three to class two event, a proceeding that is not recommended by the author.

The two methods are also realisations of level and edge triggering. Looking at the first one: here the event is always triggered when a certain level is passed. In the second method the component gets switched "on" when the level is reached, and switched "off" when the level is left again.

Aspects of Model Development

In the model description the event, or rather the part that includes the event, is separated of the rest. Only the event conditions must remain in the "model body". Especially, handling events of class three and four is very easy here. This model structure also supports the maintainability and reusability as the different components can easily be exchanged to other ones in order to build up new models on the same body.

Numerical Aspects

Again, in contrast to the *Discrete Section method* the numerical treatment is not as easy because these components do not cause discrete changes but they supply the model with continuous data and so they have to be part of the dynamic calculations when they are triggered.

Conclusion

This method offers an easy maintainability of the model. The component exchange marks a step toward modular modelling. We will meet this method in a modified version again later when we present the Model Interconnection Concept and its extensions.

6.5 "Sequential Models" - Method

In the *Sequential Model method* different models are sequentially simulated and they build up the system in this way. We start the simulation with one model. When a condition for an event is fulfilled the calculations are stopped, new initial values are calculated and the next model is started.

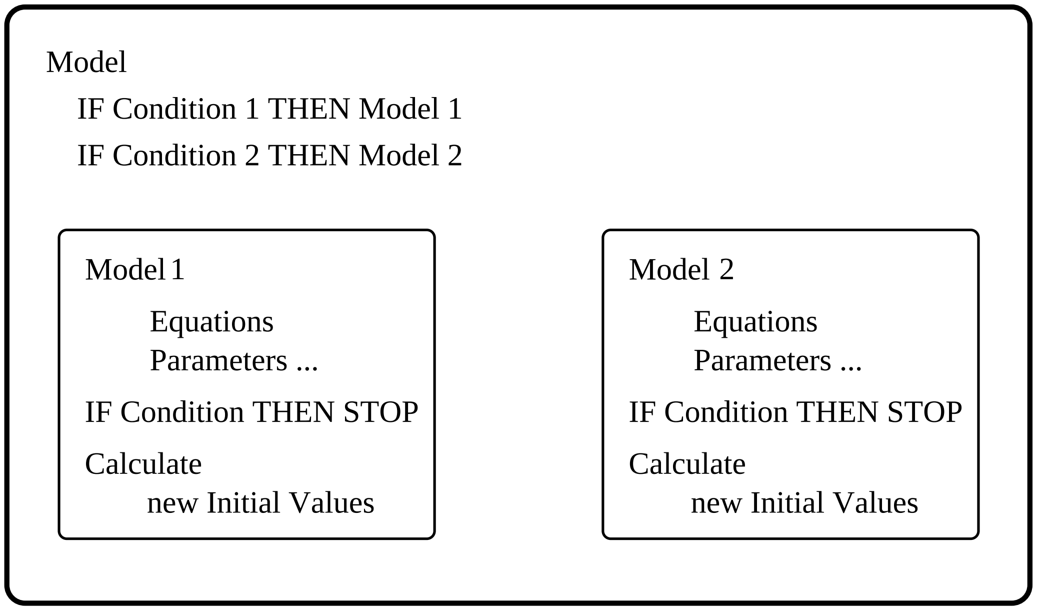


Figure 6.5: Two Sequential Models

Aspects of Simulation Methodology

The model description is here completely split into at least two parts that are sequentially processed. If the difference between these models is only a change of parameters or a reinitialisation it may be subsumed in the *Concatenated Runs method.*

Here, we state again the mixture of model description and experiment description as this method represents a concatenation of different simulations. In this case it is indeed more profound due to the fact that this is not only a concatenation of experiments but of experiments with different models.

Aspects of Model Development

The event conditions have to be placed right on top of the model description in order to take the decision which model to take first.

As the models are completely separated they could also be taken out and simulated on their own. On the other side this method is not very economical, as even if there are similar parts in the models this fact is not made use of.

Numerical Aspects

The requirements to the numerical algorithms are very low. Only the basic event capabilities are inevitable.

Conclusion

The big advantage of that method is that the models almost do not have to be adapted when simulated sequentially. The organisation of the simulated models can also be done out of a macro and can therefore be placed nearer to the experiment. Nevertheless, we can go further as I will show with the next method.

6.6 "Extended Experiment" - Method

All the event condition descriptions are separated from the model in the *Extended Experiment method*. They are now part of the experiment. Experiment here means more than only applying the basic method integration. The extension of the experiment is that it has also to handle the events. The actual running model is being observed and if a state reaches a threshold, i.e. when the condition for an event is fulfilled, the simulation run stops. The experiment decides which model to take next, calculates new inital values or parameter values and continues the simulation.

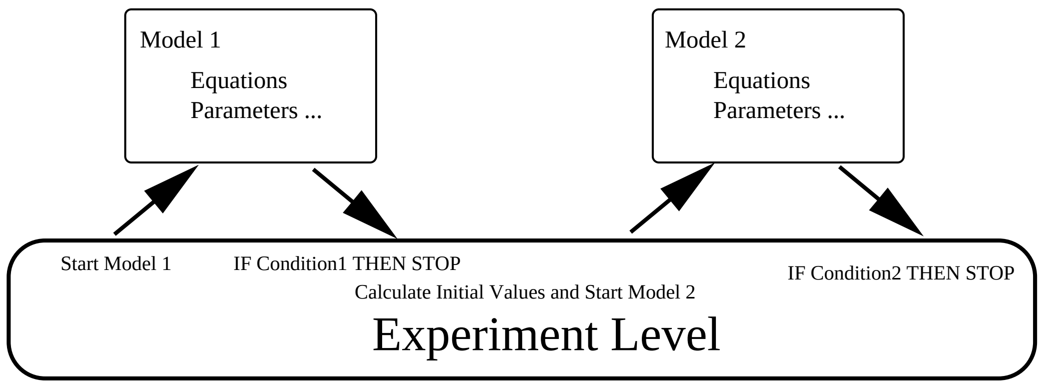


Figure 6.6: An Extended Experiment with two models

Aspects of Simulation Methodology

Events are completely separated from the models. The model is reduced to a very basic level. The experiment is seen as an active part of the system, as it controls the simulation run depending on the results given by the different runs. A new requirement is that the experiment must be capable of calculating, because in this method the experiment must put the new initial values for the next run at disposal of the actually valid model.

Aspects of Model Development

This method has the big advantage that the model description is completely reduced to one part of the system. On the one hand, the reusability aspect is here realised perfectly. On the other hand, the description of the resulting model is divided into the model description, the event description and the connection description. The last two are part of the experiment description.

Numerical Aspects

As the experiment stops the calculations of the integration algorithm, one part of its tasks is the state event finding procedure. The subsequent actions after stopping can therefore be

- Restarting the same model at the level before the last integration step with a shortened step length.
- Triggering the event at the begining or end of the last integration interval.
- Applying special interpolation routines for the evaluation of the final values out of the values the integration algorithm provided.

Conclusion

I think that this method is the most interesting one. The reasons will be given later, as I focus on this method or rather its fundalmental ideas, when I present the *Meta Model Concept*.

Chapter 7

Event Methodology – Examples and Case Studies

In this chapter I will illustrate the methods presented in the previous chapter by investigating examples, benchmarks and case studies in different simulation languages. I will also show that in some simulation languages different methods for describing state events can be used. Furthermore, not all of the presented methods can be realised with every simulation language or example.

7.1 Bouncing Ball

As the first example I present model descriptions of the bouncing ball (microscopic view) with a spring damper system. A more detailed description of this example can be found in section 4.2.

The motion of the ball is simplified modelled with an ODE of second order. When the ball hits the ground the elastic deformation is modelled by a spring and damper system. An additional force is then applied to the ball, the force of the spring with spring constant k and the damping force with damping factor c. The values of the parameters can be gathered from the model descriptions. The corresponding equations are given in the following way:

state equation:

$$\ddot{x} = -g + \begin{cases} 0 & \text{if condition 1} \\ k \cdot (r - x)\frac{1}{m} - c \cdot \dot{x}\frac{1}{m} & \text{if condition 2} \end{cases}$$

initial value

$$\begin{aligned} \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \\ \dot{y}(t_0) &= \dot{y}_0 \\ y(t_0) &= y_0 \end{aligned}$$

event condition function 1

$$(x - r) \geq 0$$

event condition function 2

$$(x - r) \leq 0$$

event: add or remove the component

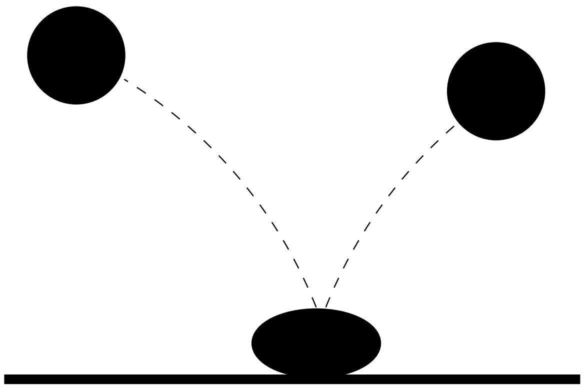


Figure 7.1: Bouncing ball with hitting process

7.1.1 "All-in-one" - Method – SIMNON

The following model is written in the simulation language SIMNON and is taken out of the SIMNON distribution [23].

```
CONTINUOUS SYSTEM BOUNCEQ
" Version:      1.0
" Abstract:
" Description:  Bouncing ball demo
" Revision:     1.0
" Author:       SSPA Systems, Gothenburg, Sweden
" Created:      1993-05-28

state y v" Height and speed
der dy dv

dy=v
dv=-g+f/m   " Newton II
f=if y<r then fspring+fdamp else 0
fspring=(r-y)*k
fdamp=-c*v
c: .2       " Damping factor
g: 9.81     " Gravitational acceleration
k: 1000     " Spring constant
m: .1       " Mass
r: .05      " Radius

y: 1        " Initial height
v: 0        " Initial velocity

END
```

So we here can find the description of the state event with an "ordinary" `IF` - statement. No special algorithms get used here, the evaluation of the statment is carried out only at every integration step.

The same example can be modeled in the same language applying the *Component Exchange method.*

7.1.2 "Component Exchange" - Method – SIMNON

Here I use the `Connecting System` that links two `Continuous Systems`: the first system is the model with the equations for the falling ball, the second is the component containing the force coming up from the hitting process. If the ball hits the ground the component is linked, otherwise the input is set to 0.

Here, I used the same parameters as in the model above.

```
CONNECTING SYSTEM BallCon
sd[ballequ]= IF x[ballequ] < r THEN sd[ballcomp] ELSE 0.
h[ballcomp]=h[ballequ]
v[ballcomp]=v[ballequ]
r[ballcomp]=r
r:0.05
END

CONTINUOUS SYSTEM BallEqu
INPUT sd
OUTPUT h v

STATE x y
DER dx dy

dx=y
dy=-g+1/m*sd
h=x
v=y
g:9.81
m:0.1
```

```
END

CONTINUOUS SYSTEM BallComp
INPUT h v r
OUTPUT sd

fs=(r-h)*k
fd=-c*v
sd =fs+fd
k:1000
c:0.2
END
```

7.1.3 "Sequential Models" - Method – SIMNON

A third possibility is presented by applying the *Sequential Models method.* The system description is split up into two different models. The first, `BALLF`, consists of the description of the fall and the second, `BALLSD` of the spring and damper system combined with the fall equations.

The macro file `BALLFSD` takes over the control of the experiment that means the initialising – here only the passing of the final values of parameters to the initial value settings – and also the starting of the correct model.

The parameters are again the same as in the first example and therefore are not commented in the following description.

```
CONTINUOUS SYSTEM BALLF
state y v
der dy dv
time t
dy=v
dv=-g
st=CTERM(y<r)
g: 9.81
```

```
r:0.05
y: 1
v: 0
END

CONTINUOUS SYSTEM BALLSD
state y v" Height and speed
der dy dv
time t
dy=v
dv=-g+1/m*((r-y)*k-c*v)
st=CTERM(y>r)
c: 0.2
g: 9.81
k: 1000
m: .1
r:0.05
y: 1
v: 0
END

MACRO BALLFSD
let sconst=1000
let dconst=0.2
let height0.=1
let veloci0.=0
let mass=0.1
let radius=0.05
let time.=0
let ftime.=10.             !simulated time
let dist=height0.-radius
newplot
axes H 0 ftime. V 0 height0.
IF dist LE 0 GOTO hit
```

```
label fall
syst ballf
par r[ballf]: radius
init y[ballf]: height0.
init v[ballf]: veloci0.
store y[ballf] v[ballf]
plot y[ballf]
simu time. ftime.  0.001
disp y[ballf]/height0
disp v[ballf]/veloci0
disp t[ballf]/time
write 'y=' height0. 'v='veloci0.

if time. GE ftime. goto ende

label hit
syst ballsd
par c[ballsd]: dconst
par k[ballsd]: sconst
par m[ballsd]: mass
par r[ballsd]: radius
init y[ballsd]: height0.
init v[ballsd]: veloci0.
store v[ballsd] y[ballsd] t[ballsd]
plot y[ballsd]
simu time. ftime. 0.001
disp y[ballsd]/height0 v[ballsd]/veloci0 t[ballsd]/time
write 'y=' height0. 'v='veloci0.
let dist=height0.-radius
IF dist GT 0 GOTO fall

label ende
END
```

7.2 Block on a Rough Surface

The system of the block on a rough surface is a well-known benchmark. A detailed description and a solution can be found in the ACSL manual [1, p. A-102 ff].

The example deals with a block resting on a rough surface. A spring is attached on the one side to the block and on the other side to a moving point. Therefore, the spring gets extended or compressed so that a force is applied on the block and causes its deceleration or acceleration.

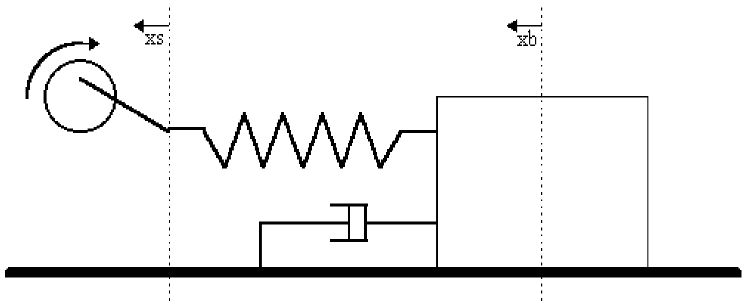


Figure 7.2: Block on a Rough Surface

We start when the block rests. When the force applied to the block exceeds the breakout force bf , the block starts to move. Due to the friction the blocks gets slower again until it is grabbed. Now it sticks until the breakout force is reached again and it starts to move again. We denote the sum of the applied forces with $f1$ and the Coulomb friction with $f2$. $f2$ depends on whether the block moves or is stuck. When it moves $f2$ represents a sliding friction, when not, $f2$ cancels all other forces.

state equations

$$\ddot{x} = (f1 + f2)\frac{1}{m}$$

initial values

$$\begin{aligned} \dot{x} &= \dot{x}_0 \\ x &= x_0 \end{aligned}$$

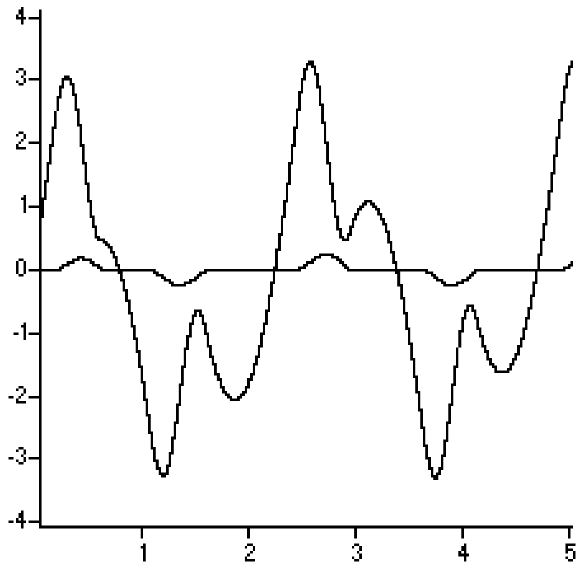


Figure 7.3: Velocity and Sum of Forces

event condition function

$$| f1 | -bf = 0$$

event: reinitialise the velovity and reassign the sliding friction

The exact equations and parameters can be gathered from the solutions.

Figure 7.3 shows the velocity and the sum of forces calculated by the SIMNON model.

7.2.1 "Discrete Section" - Method – ACSL

The first model taken from [1, p.A-102 ff] is using the `SCHEDULE` statement in combination with the powerful `DISCRETE` section. In this `DISCRETE` section all the parameters are reassigned. They have to be kept constant until the section gets scheduled again. This method seems to be convenient for this example.

```
PROGRAM - friction test
      !------------------models a block sliding on a rough
      ! surface under the action of a force applied through a
      ! spring. the spring force must exceed a breakout coulomb
      ! friction force in order to start to move. once it starts
      ! to move the frictional force has a value equal to the
      ! sliding friction in a direction opposing the motion
      !------------------define simulation environment
      CINTERVAL      cint = 0.020
      ALGORITHM      ialg = 4
      MAXTERVAL      maxt = 0.050
      MINTERVAL      mint = 1.0e-6
      NSTEPS         nstp = 1
      !------------------define global constants
      PARAMETER     (pi = 3.14159     )
INITIAL
      !------------------give the states the initial cond values
      RESET("NOEVAL")
END ! of initial
DERIVATIVE
      !------------------forcing function is displacement of
      !                  free end of spring attached to body
      CONSTANT       xfa = 0.05        , freqf = 0.40
      xf      = xfa*SIN(2.0*pi*freqf*t)
      !------------------sum of forces on body excluding friction
      CONSTANT       ksp = 100.0       , kbvd = -0.50
      sumfb   = ksp*(xf - xb) + kbvd*xbd
      !------------------force due to coulomb friction exactly
      !                  cancels other forces when stuck
      fbcf    = RSW(stukb, -sumfb, fbsf)
      !------------------acceleration of body
      CONSTANT       mb = 1.0
      xbdd    = (sumfb + fbcf)/mb
      !------------------integrate for velocity and position
      xbd     = INTVC(xbdd, xbdic) ; CONSTANT xbdic = 0.0
      xb      = INTEG(xbd, xbic)   ; CONSTANT xbic  = 0.0
      !-------------------define *phi* function for stick/unstick
      fib     = RSW(stukb, ABS(sumfb) - kbbf, xbd)
      !-------------------schedule the execution of the discrete
      !                   block *stick* on a zero crossing
      SCHEDULE stick .XZ. fib
      !-------------------specify termination condition
      TERMT(t .GE. tstp, 'Time Limit') ; CONSTANT tstp = 4.99
END ! of derivative
DISCRETE stick
      !-------------------handle coulomb friction
      !-------------------initialise stuck flags
     INITIAL
      LOGICAL        stukb
      stukb   = xbd .EQ. 0.0
     END  ! of initial
      !-------------------stuck flag toggles unless force exceeds
      !                   breakout force on crossing zero
      CONSTANT       kbbf = 2.50
      stukb  = .NOT.stukb .AND. ABS(sumfb) .LT. kbbf
      !-------------------sliding friction opposes applied force
      CONSTANT       kbsf = 2.00
      fbsf    = RSW(stukb, 0.0, SIGN(kbsf, -sumfb))
      !-------------------reset velocity to exactly zero. you must
      !                   know *xbd* is a state variable for this
      xbd     = 0.0
      !-------------------record status
      CALL LOGD(.TRUE.)
      !-------------------define debug dump flag and condition
      LOGICAL        dump ; CONSTANT  dump = .FALSE.
      IF(DUMP) CALL DEBUG
END ! of discrete
END ! of program
```

7.2.2 "Concatenated Runs" - Method – SIMNON

In contrast to the preceeding model I use the *Concatenated Runs method* here. When the block starts moving or stops, the same model gets reassigned with new paramters and initial values and then it gets started again.

Since SIMNON offers only a `CTERM` statement for conditional termination I had to introduce some more auxiliary logical variables so that the right condition for stopping is taken. Another auxiliary variable was taken to support the reassignment of the parameters in the `MACRO` since the command level does not offer the necessary capabilities for the calculations.

```
CONTINUOUS SYSTEM PUSH
STATE x  y
DER   dx dy
TIME  t
"Spring
ampa=amp*SIN(2*pi*frequ*t)
"Sum of forces
sum=spc*(ampa-x)+dc*y
"Coulomb friction
cf=IF stuck THEN -sum ELSE sf
"Motion of block
dy=(sum+cf)/m
dx=y
"auxiliary variables for reassignment
hsf=-SIGN(sum)*sfc
hstuck=NOT(stuck) AND (ABS(sum)<bf)
m    : 1.0   "mass
sf   : 0.0   "sliding friction
spc  : 100.  "spring constant
dc   :-0.5   "damping constant
sfc  : 2.0   "sliding friction constant
amp  : 0.05  "amplitude
frequ: 0.4   "frequency
```

```
pi   : 3.14159
bf   : 2.5   "breakout force
stuck: 1.    "logical varibles
f    : 0     "forward motion
b    : 0     "backward motion
x    : 0     "block position
y    : 0     "block velocity
"conditions for termination
st1=IF stuck THEN CTERM((ABS(sum) > bf))  ELSE 2
st2=IF not(stuck) and f THEN CTERM(y < 0) ELSE 3
st3=IF not(stuck) and b THEN CTERM(y > 0) ELSE 4
END

MACRO PUSHMAC
syst push        "activate system
store x y sum
plot x y sum
simu 0 5         "do a first run
label restart

disp hstuck/stick sum/s y/y0 hsf/h1 t/time
"logic block for assignment of motion indicators
if stick. LT 1. goto cont1
par stuck:1
par f:0
par b:0
goto cont3
label cont1
par stuck:0
if s. LT 0 goto cont2
par f:1
par b:0
goto cont3
label cont2
```

```
par b:1
par f:0
label cont3
init y:0.         "reinitialise velocity
par sf:h1.        "reassign sliding friction
simu 0 5 -cont    "restart simulation
if time. LT 5 goto restart
END
```

7.3 Constrained Pendulum

In chapter 5 I discussed the solutions of the constrained pendulum. The detailed description is given in section 4.2. The events in this problem, the hitting and the leaving of the pin are both class two events (four classes approach) as the parameter length *and* the velocity, a state variable have to be changed.

The solutions can be classified as follows:

ACSL-solution	Discrete Section Method
SIMULINK-solution	All-in-one Method
ESL-solution	Discrete Section Method
MOSIS-solution	Discrete Section Method
SIMNON-solution	Concatenated Runs
SLIM-solution	All-in-one Method
ANA 2.x-solution	Discrete Section Method
DYMOLA-solution	Discrete Section Method
ModelMaker-solution	Discrete Section Method

Chapter 8

The Meta Model Concept

The methods presented in chapter 6 are partly "grown" and partly "well constructed". It turns out that the event classification is inconsistent with respect to the possibilities of the implementation of the event description methods in the simulator, and vice versa.

- Some methods can be implemented in one specific language.
- One specific method can be implemented in some language, but not in all.
- Some modelling techniques (macroscopic, microscopic view, ...) result in one event class and a corresponding method.
- Some methods (which can be implemented) require a specific modelling technique.

Therefore in this chapter I develop the *Meta Model Concept*.

After shortly describing the *Model Interconnection Concept*, I will present the *Extended Model Interconnection Concept* that I developed in linking the *Extended Experiment method* (presented above) and the *Model Interconnection Concept*.

Then, based on this combination, the *Meta Model Concept* is developed, a concept that I developed for the easy integration of state events in continuous simulation models. With this concept I describe a completely new approach to state events, suited not only for modelling tasks but also for runtime control and even algorithmic matters.

8.1 The Model Interconnection Concept

The *Model Interconnection Concept* was developed by Schuster [57] and implemented in the simulation language "MOSIS" (=Modular Simulation System) [58].

"The main idea behind this concept is that a complex simulation model can be built up from several smaller models that communicate with each other." writes Schuster in his thesis [57, p. 33]. The main parts are therefore the *model* and the *model links*. The definition of "model" gets generalised so that a model can be

- the mathematical/algorithmic description of a real system
- a test model (for evaluation or as data supply for other models)
- a constant or a fixed function or
- an interface to a real system (man in the loop, hardware in the loop)

The simulation model is then split up to such smaller autonomous (sub-) models. The communication between the models is strictly fixed as unidirectional and takes place only at certain times (communication intervals).

The advantages of this concept are the support

- of parallel simulation methods
- of easy exchange of submodels in order to generate new models

- of coupling different simulation languages and
- of modular development

From an object oriented point of view the model description can be seen as a class definition. For the simulation an instance of this model has to be created, in other terms an object that can communicate with other objects.

8.2 The Extended Model Interconnection Concept

The starting point for the proposed extensions to the *Model Interconnection Concept* is the *Extended Experiment method*. The main part of this method is the transfer of the event from the model to the experiment level. The event condition remains in the model description and is reduced to a single statement, the *conditional termination*. The condition for the termination here obviously is the state event condition. If it becomes true, the simulation run is stopped and the experiment decides what to do next. So only the calculation of the actual model is stopped. The simulation of the system continues in some way. The term *extended* is here applied as the experiment provides not only the parameters for the dynamic calculations but also takes over the event passing and therefore a task of the model.

In this concept the event is replaced by different actions: class one and two events correspond to restarting the model after a reinitialisation with parameters and initial values that are calculated depending on the final values of the last simulation run. Class three and four events are replaced by starting a simulation run with new models.

In this first step of linking the Model Interconnection Concept and the Extended Experiment Method I propose the following actions:

Class One Event: Change of Parameters and Input variables

Using the generalised definition of a model I put the different parameters in separate models. The model does not need to consist of only a constant. If the parameter depends on other parameters or initial values, this model can also consist of a function that evaluates the parameter value at the beginning of an experiment and holds it constant throughout the run. This model can therefore be seen as a CSSL model consisting only of either an initial section or a discrete section with an infinite sampling time. Thus, the action of the experiment is to call the right *parameter model PM* after a conditional termination, and afterwards to restart the calculations with the new parameters. Here the experiment has to decide which parameter model has to be taken next after a termination.

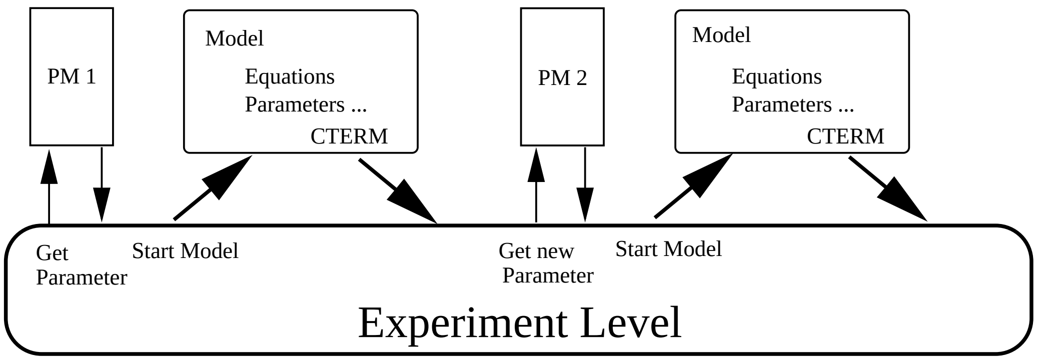


Figure 8.1: Class one and two events

As an example let me mention here a cooking process: when we put a pot of water on the stove, we apply the maximum of heat power. Therefore, we start the parameter model HMAX and initialise the pot model POT with it. Then the pot gets heated up very quickly. When the water has reached a certain temperature, the dynamic calculations are stopped and then we will apply a reduced heat power described in another parameter model HR. So we reinitialise the pot model POT with the parameter model HR and start the calculations again and so on.

Class Two Event: Change of State Variables

For this class of events the simulation run is stopped, the new values of the state variables are evaluated as described above and the run gets started again with a reinitialisation. Another possibility is that the new values are calculated within the models in a "terminal section" after the conditional termination statement has become true.

The advantage of the two methods above is that the calculations of the new parameters and state variables for reinitialising are done in separate models. Thus, we do not require the experiment level to be able to calculate. It has only the task of starting the right model at the right time. Figure 8.1 shows the proceeding with events of class one and two.

Class Three Event: Change of Components

The model is here again split up into different submodels. The conditional termination signals that a component of a model has to be changed. In this case we simply link the new component and remove that one that is no longer valid. After that the new constructed model is started again. A component means here a function of state variables. The change is not as profound as in the next class.

Class Four Event: Change of the Structure

Here I proceed as above, i.e., I exchange models. But in addition I have to evaluate the initial values with which I start the new model. As I have already mentioned above the exchange of models here means also the change of the state space.

In order to be able to link the new model to the others, I suggest to use separate models that build an interface to them. These interface models have the task to prepare the output variables from the exchangeable part as input variables for the static part of the model or to another exchangeable part as well.

8.3 The Meta Model Concept

In this concept I combine the results of the steps described above, and, taking again a step forward, I present the *meta model concept* for describing and handling state events in continuous system simulation.

The main idea is to leave completely the terms *model* and *experiment* and give them new meanings:

The process that we are simulating is now divided into small components. Component denotes parameters, constants, functions that describe parts of the real process, ... Every component is then put in a separate model $M_1, M_2, \ldots, M_n$. Every model has its input ports and output ports where it can be linked to other models.

Additionally, we need three special model types. At first the *interface model IM*. As I have already mentioned above, the *interface model* builds up the interface between two models. It calculates new initial values out of the terminal values of other models or manages that a model gets only "prepared" data from other models. Prepared data can be scalar data that are needed as vectors, ...

The second special model is the *result model RM*. This model stores the results of the simulation run. It is connected via interface models to those models that produce data the user is interested in.

The third and most important model is the *meta model MM*. The *meta model* mainly consists of event conditions and events. Depending on these event conditions links between models are cleared or set up. The *meta model* also starts and stops calculations within models. So this *meta model* organises the simulated "model" – here in its ordinary sense – that is only built up dynamically when the simulation runs. Therefore I call it the **meta** *model*.

Let us now take a closer view to the parts of this concept.

The Model

The model denotes here again the generalised model out of the *Model Interconnection Concept*. It can therefore be either just a parameter or a function or a model that could be separately simulated. The model has defined inports and outports that allow the linking to other models. So we have to define output variables and input variables.

The Interface Model

The *interface model* has the task to transfer data and prepare them "on the fly".

An example for data preparation is given by the looping pendulum that we have already described. Here the swinging phase is modelled by using polar co-ordinates. On the other hand, the phases of the free fall are modelled by using Cartesian co-ordinates. So the *interface model* can here be used to convert the results to Cartesian co-ordinates for example, and the results can be plotted in one.

On the other hand, prepared data can also make good errors which are due to the application of numerical algorithms.

When we simulate a tank system that is emptied, it can happen that we get confronted with negative contents. This can be seen easily: We apply a numerical integration algorithm with a step by step mode of operation to the equations describing the state variable "content of the tank". In addition, we have a restricted accuracy. In most of the cases, the algorithm will not stop when the content is exactly at zero but it will calculate a little bit further. This fact can be dangerous for other components when they are prepared for getting "useful" (in this case: positive) values only. Therefore the *interface model* prevents such problems.

The advantage of these *interface models* is that the models do not have to be adapted, but only small *interface models* have to be built up and put in between of those sensitive models.

The Meta Model

The *meta model* is the central part of the simulated system. Its main tasks are:

- to build up a representation of a "real" system with a web of models that can be simulated
- to supervise the simulation and check the event conditions
- to start – if necessary – a locating procedure
- to carry out the event actions – to link components or to remove links and
- to start or stop simultaneously the calculations in the corresponding models

For an easier handling, the *meta model* can also enable and disable components. In other words, the *meta model* sends to the component the message whether it is active or not. Subsequently the model produces the corresponding output. Otherwise, when the model is inactive it stops the calculations and gives a predefined output, e.g. 0. So the links need not to be removed.

8.3.1 Formalised Description

A formalism describing what is going on in this concept will obviously be the description of the contents of the main part, the meta model. The meta model therefore represents the environment we are living in, that creates the simulated system, starts and stops calculations, creates and removes links ...

In order to formalise the new concept, I now group the models. At first there is the *model group MG* consisting of

1. the *models*,
2. the *interface models* and
3. the *result model*.

Secondly, there is the *event group EG* consisting of the conditions and actions to be carried out. Additionally, the *priority* of an event has to be assigned.

The Model Group

If we now build up a new system for simulation we describe it in a *meta model*. For building up a system we need some models that can be linked together. These models are predefined or copies of library models. The library models are named here

$$M_1, M_2, \ldots, M_n \qquad n \in \mathrm{N}$$

When loading these models I indicate a copy of a model, here copy 1 of model 1 by C_1M_1. In case of an object-oriented modelling approach one would call these copies "instances".

The description of the *interface models* and the *result model* is similar apart from a little difference: we replace M_i by IM_i or RM_i. The copies of such models are obviously called C_jIM_i and C_jRM_i.

So we can now state the set of used models in our system as

$$\begin{aligned} MG = \{ \quad & C_1M_1, C_2M_1, \ldots, C_{i_1}M_1, C_1M_2, \ldots, C_{i_j}M_j, \\ & C_1IM_1, C_2IM_1, \ldots, C_{k_1}IM_1, C_1IM_2, \ldots, C_{k_l}IM_l, \\ & C_1RM_1, C_2RM_1, \ldots, C_{m_1}RM_1, C_1RM_2, \ldots, C_{m_n}M_n \quad \} \\ & \qquad \text{i, j, k, l, m, n} \ \in \mathrm{N} \end{aligned}$$

The Event Group

Now the *meta model* needs to know what to do under which condition. We have to describe these conditions C and actions A in the *event group*. Furthermore, priorities p have to help in concurring events.

The actions the *meta model* carries out with the models is to create or to remove links between pairs of them. Linking of models consist of connecting output variables to input variables. In a formalised manner I describe for example: "when C1 is true do action $A1$ that is create a link between the variable out_1 of C_1M_1 and in_1 of C_1M_2" with

$$A1 = \{out_1.C_1M_1/in_1.C_1M_2\}$$

Each of the actions $A1, A2, \ldots$ corresponds to a condition $C1, C2, \ldots$ and the event group is a set of pairs of them,

$$EG = \{(C1, A1), (C2, A2), \ldots, (Cn, An)\}$$

Other actions than the one I described above are also possible: when a condition $C2$ is true and the action is to remove the link that was created before, I will write

$$A2 = \{-out_1.C_1M_1/in_1.C_1M_2\}$$

Obviously, more actions also can be placed in a set and we also can combine the types "removing" and "creating" in one set.

This same description is needed for describing a state event or also a time event. In fact in this concept we use events to build up system descriptions.

Additionally, there are two more possible actions: an event is also capable of enabling or disabling events – including itself.

When an event is triggered it may occur that other events are no more meaningful in the system description or that an event, once it is triggered, must not be reactivated. And also vice versa: an event that was removed can be activated again by this statement. A description similar to the one above is used for these cases:

$$A3 = \{-C1, -C3, C4\}$$

This statement means that when condition $C3$ is fulfilled condition $C1$ and also itself should be removed from the active part of the model so that they cannot be triggered any more. On the other hand, condition $C4$ is activated.

Furthermore, it is also possible for the *meta model* to enable or disable components. In order to come to a more efficient simulation it is possible to disable a component that is already linked and reduce its outputs to predefined constant values. So the action here is to stop the simulation of this model and replace the outputs by constant values.

Continuing with the established link created in the example above, we now want to disable C_1M_1. Obviously, we have also to assign the output variable out_1 with a constant value *const*:

$$A4 = \{-C_1M_1(out_1 = const)\}$$

When we enable the model again we can do this by

$$A5 = \{+C_1M_1\}$$

Besides, only models that were already linked to other models can be disabled and, consequently, only disabled models can be enabled.

Finally we have to add a command that enables a reinitialisation of a model. When we want to change a state variable in a model we have to give a command that carries out this resetting. In this concept we describe the action of changing the variable $x1$ to $x1new$ by

$$A6 = \{init(x1.C_1M_1 = x1new\}$$

The Priority of an Event

In addition to the conditions C and the actions A we have to provide an integer denoting the priority of an event. Due to the formulation of the conditions but also due to numerical errors the situation may arise that two or more events should be triggered at the "same time". In this case the priority denotes a sequence what event has to be handled first.

After triggering an event and after the actions have been carried out, the conditions are checked again and if necessary further events are triggered.

The proposed proceeding is to give a unique integer number to each event out of the event group. Then the event can be arranged in a sequence.

Therefore, I expand the event group by an integer number p for each pair of condition and action, that assigns the priority:

$$EG = \{(C1, A1, p1), (C2, A2, p2), \ldots, (Cn, An, pn)\}$$

The conditions are checked according to that order. When a condition is found to be true, the actions are carried out and after that all the conditions are checked again, starting at the beginning of the list.

The Meta Model

In the *meta model* we define all these conditions and actions that deal with copies of models out of the *model group.*

Besides, it is also necessary to include runtime commands for starting and stopping, for creating copies of models, for setting parameters, ...

Furthermore, we decide in the *meta model* which integration algorithm to use and the integration parameters are set here, too.

8.3.2 Assessment of the New Concept

The main advantages of this concept are

The modular concept: The process that we want to simulate is divided into small parts that are calculated independently. Therefore, at the model development level we can build up the model components very easily. Different groups can describe parts of the model independently. They get linked together eventually by using interface models.

Besides, we can define separate numerical integration algorithms to be used in different components that are specially adapted and adjusted to the model they are used in.

Another advantage is the possibility to replace models by hardware and therefore getting a "hardware–in–the–loop simulation".

The modular concept gives us also the possibility of parallel simulation. We may simulate different components on different processors.

The reusability of model components: As the model components do not contain any statements depending on different events and following actions, they can easily be built up independently and independent from the other components and can also be used in different simulations. Thus, we can build up a library of models. When building up a new system we only have to take some of these library models and only have to design the meta model and the interface models.

The maintainability of model components: It is very easy to maintain the components or even exchange them with others as they are acting independently. If new developments are made and the components have to be changed or updated, the new components can simply be simulated without changing the "model".

There are very few additional demands to simulators when using this concept:

Dynamic linking: This is the main part of the concept: the dynamic linking of components. These components build up a state dependent model. Therefore, the linking has to be dynamic, that means it must be possible to create new links to models and on the other side to remove links concerning to the event conditions.

Independent calculation: For this demand the simulator needs a communication management that organises the data flow from and to the linked components even if they are not calculated on the same processor.

Exact time management: Although the event is now replaced by the action of linking we are still confronted with the problems I have presented above. It is necessary to find the "exact" time when the event has to be triggered. So the meta model must have the capability to intervene in the integration algorithms.

8.4 Application of the Meta Model Concept for Events

8.4.1 Bouncing Ball

Here I will give two descriptions of the bouncing ball. The first models the hitting process with a spring and damper system whereas the second model simply models this process with a change of the state variable $\dot{x}$. As I introduced and discussed these examples of the bouncing ball in chapter 4, I here reduce the descriptions to the essential parts. Some models of the bouncing ball can also be found in chapter 7.

Bouncing Ball - Microscopic

As a first example I want to show the model description of the simplified motion of the bouncing ball where the process of hitting the ground is described by a spring and damper system. The system description is as follows:

state equation

$$\ddot{x} = -g + \begin{cases} 0 & \text{if condition 1} \\ k \cdot (r - x)\frac{1}{m} - c \cdot \dot{x}\frac{1}{m} & \text{if condition 2} \end{cases}$$

initial values

$$\begin{aligned}\dot{x}(t_0) &= \dot{x}_0\\ x(t_0) &= x_0\\ \dot{y}(t_0) &= \dot{y}_0\\ y(t_0) &= y_0\end{aligned}$$

event condition function 1

$$(x-r) \geq 0$$

event condition function 2

$$(x-r) \leq 0$$

event: add or remove the component

When we translate this description, we build up a model M_1 with the first part of the state equation. The component is described in a second model M_2.

The conditions $C1$, the event condition function 1 – for the event when the ball hits the ground – as well as and $C2$, event condition function 2 – when it leaves the ground again – are given by

$$\begin{aligned}C1 &= \{(x-r) \geq 0\}\\ C2 &= \{(x-r) \leq 0\}\end{aligned}$$

The event gets described in different actions. Let us start when $C2$ gets true, when the ball hits the ground.

The action $A1$ is to link the component M_2 to the model of the motion M_1 and the action $A2$ to link the necessary variables to this second model. So the links between the variables have to be added in the following two lines:

at first the additional accelaration a calculated in model M_2 has to be linked to M_1 and then the position x and the velocity $\dot{x}$ of M_1 have to be linked to M_2. In $A3$ we then set $C2$ inactive as the event should only be triggered when the condition gets true. Furthermore, we have to enable the second condition $C1$ that checks when the ball leaves the ground again.

$$\begin{aligned} A1 &= \{outa.C_1M_2/ina.C_1M_2\} \\ A2 &= \{pos.C_1M_1/inpos.C_1M_2, vel.C_1M_1/invel.C_1M_2\} \\ A3 &= \{+C1, -C2\} \end{aligned}$$

When $C1$ gets true the created links have to be removed, $C2$ has to be set active and $C1$ set inactive. So we have to define the following actions:

$$\begin{aligned} A4 &= \{-outa.C_1M_2/ina.C_1M_2\} \\ A5 &= \{-pos.C_1M_1/inpos.C_1M_2, -vel.C_1M_1/invel.C_1M_2\} \\ A6 &= \{-C1, +C2\} \end{aligned}$$

Now we have the following *event group*:

$$EG = \{(C2, A1), (C2, A2), (C2, A3), (C1, A4), (C1, A5), (C1, A6)\}$$

The *result model RM* for collecting the data also has to be created and linked to the corresponding variables of C_1M_1.

Finally, we can compose the *meta model* with the instructions for creating copies of M_1 and M_2, the conditions and the actions. Furthermore, the commands for starting the simulation have to be added.

Bouncing Ball - Macroscopic

In this example of the bouncing ball we simplify the description of the hitting process by describing it with a simple reinitialisation of the velocity. The description of this system is

$$\begin{array}{lrcl} \text{state equation} & \ddot{x} & = & -g \\ \text{initial values} & \dot{x}(t_0) & = & \dot{x}_0 \\ & x(t_0) & = & x_0 \\ \text{event condition function} & x & = & 0 \\ \text{event} & \dot{x}_{new} & = & -\alpha \cdot \dot{x}_{old} \end{array}$$

For the translation of the event condition function we have to add here a term that checks whether the velocity is negative so that only zero crossings from positive to negative trigger the event.

$$C1 = \{(x \leq 0) \quad AND \quad (\dot{x} \leq 0)\}$$

The event is described in a separate model M_2 that consists only of this simple function. So when the condition gets true we only have to start this second model and to change the state variable xdot in M_1. The action $A1$ is given in the following line

$$A1 = \{init(xdot.C_1M_1 = C_1M_2(oldxdot))\}$$

Therefore the *event group* is built up with only one condition and action pair

$$EG = \{(C1, A1)\}$$

8.4.2 Looping Pendulum

In describing the looping pendulum – a more detailed description can be found in section 4.1 – we have again two different models. The first model

M_1 describes the swinging pendulum and the second M_2 the fall of it. The conditions for swapping from M_1 to M_2 and vice versa are that the centrifugal force declines under a certain threshold ($C1$) and otherwise that the distance r from the mass to the point of rotation is equal to the length of the string l ($C2$):

$$\begin{aligned} C1 &= \{(\dot{\varphi}^2 l - g \mid \cos(\varphi) \mid) \leq 0\} \\ C2 &= \{(l - r) \geq 0\} \end{aligned}$$

As we describe the motion in the first model with polar co-ordinates and in the second with Cartesian co-ordinates we need additionally an *interface model* that transforms the values. So we can link then the two models and put the *interface model* IM_1 in between. When we start the simulation one model has to be set inactive.

When an event occurs, let us say $C1$ is true, the consequent actions are to set M_1 inactive and assign the output values to the final values ($A1$) and to set M_2 active ($A2$). The input serves as an initial value for the model. In addition, we have to remove the link from the *result model* to M_1 ($A3$ and $A4$) and create a link from M_2 via the *interface model* to the *result model* ($A5$ to $A7$). The *interface model* here transforms the data to polar co-ordinates so that the results are all in polar co-ordinates. Finally the condition $C1$ itself has to be set inactive and the other condition $C2$ to be set active ($A8$).

So the actions that have to be carried out when $C1$ is true are

$$\begin{aligned} A1 &= \{-C_1M_1(outphi = finphi, outphidot = finphidot)\} \\ A2 &= \{+C_1M_2\} \\ A3 &= \{-outphi.C_1M_1/inphi.C_1RM_1\} \\ A4 &= \{-outphidot.C_1M_1/inphidot.C_1RM_1\} \end{aligned}$$

$$\begin{aligned}
A5 &= \{outx.C_1M_2/inx.C_1IM_1, outy.C_1M2/iny.C_1IM_1\} \\
A6 &= \{outphi.C_1IM_1/inphi.C_1RM_1\} \\
A7 &= \{outphidot.C_1IM_1/inphidot.C_1RM_1\} \\
A8 &= \{-C1, C2\}
\end{aligned}$$

When $C2$ gets true the actions look similar with only little changes.

8.4.3 Constrained Pendulum

The constrained pendulum – the example was introduced in section 4.2 and dicussed also in chapter 5 – the pendulum that hits a pin is described with three separate models: the model of the swinging motion (M_1), the models that describe the process when hitting the pin (M_2) and when leaving it again (M_3). The last two models are again simple functions that calculate here the new angular velocity (phidot) and the new length of the string (l).

The corresponding conditions are described by

$$\begin{aligned}
C1 &= \{\varphi \cdot phipin) \geq phipin^2\} \\
C2 &= \{\varphi \cdot phipin) \leq phipin^2\}
\end{aligned}$$

When the condition $C1$ gets true we have to start M_2 and to set $C1$ inactive and $C2$ active. Some time later $C2$ will get true and then we have to start M_3 and to set $C1$ active again and $C2$ inactive. In both of the cases we have to change the parameter l and the state variable phidot.

8.4.4 Block on a Rough Surface

A detailed description and two solutions of the block on a rough surface are given in section 7.2. The description of the system is as follows:

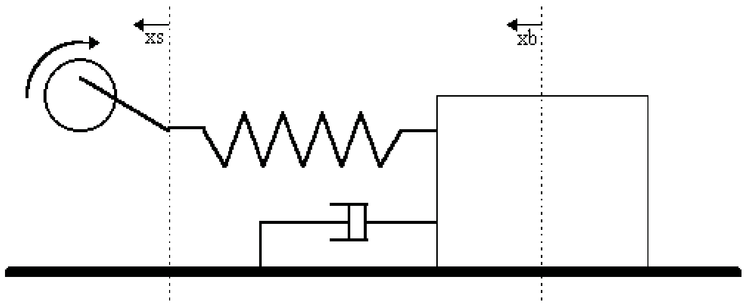


Figure 8.2: Block on a Rough Surface

state equations

$$\ddot{x} = (f1 + f2)\frac{1}{m}$$

initial values

$$\begin{aligned} \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \end{aligned}$$

event condition function

$$\mid f1 \mid - bf = 0$$

event: reinitialise the velovity and reassign the sliding friction

The block on a rough surface can here be modelled with three separate models: the model of the motion of the block and two different models that represent the friction forces when the block sticks or is unstuck. When the block sticks, the friction force cancels all other forces that are applied to the block. So this model has to get these forces as input from the model of the motion. The second friction model is only a constant.

The model of the motion can be split into two models when separating the forcing function that provides the force on the block due to the spring attached on the one end to a rotating wheel and on the other end to the block.

In order to make different experiments we easily can replace the forcing functions by other ones.

Therefore, we have got three models: M_1 describes the motion of the block, M_2 the forcing function and finally M_3 represents the friction force when the block is moving. When the block sticks the friction force is a constant that is only necessary for triggering the event when the block starts moving again and has no other links.

The conditions for the events are $C1$ and $C2$: When the block moves, we check if the velocity gets zero ($C1$) and the block is grabbed by the friction. When it sticks we have to control if the breakout force bf is reached ($C2$) by the forces due to the forcing function (ff).

$$\begin{aligned} C1 &= \{\dot{x} \leq 0\} \\ C2 &= \{ff \geq bf\} \end{aligned}$$

The consequent actions when one of the condition gets true are the following: When $C1$ is true, when the block gets grabbed, we may stop M_1 and set the output values to constant values, the position (x) linked to M_2 is set to the final position and the velocity is set (exactly) to zero.

$$\begin{aligned} A1 &= \{-C_1 M_1(outx = finx, outxdot = 0)\} \\ A2 &= \{-C1, C2\} \end{aligned}$$

When the block starts moving M_1 has only to be activated again.

$$\begin{aligned} A1 &= \{+C_1 M_1\} \\ A2 &= \{C1, -C2\} \end{aligned}$$

8.4.5 Pilot Ejection

This system of a pilot ejection is a well-known benchmark. A solution can be found in the ACSL manual [2, pp. A.13 ff]. It deals with the motion of a pilot relative to that of his aircraft from which he is ejected. With the simulation of this model one may determine whether the pilot strikes the airplane or not, after he was ejected and one also can find out the necessary ejection velocity so that he will be save.

The system can be divided into two parts

1. the motion of the airplane (M_1)
2. the motion of the pilot (M_2)

M_1 simply consits of a constant value as the velocity of the airplane is assumed not to change. We denote this value with v_a.

M_2 describes the phase of the motion in which the seat is conducted in rails that are inclined backwards with the angle θ. v_p denotes the initial velocity of the pilot and θ_p the angle of the motion of the pilot. Both values are constant here and get described in M_3. The corresponding equations for the horizontal (x) and the vertical position (y) are:

$$\begin{aligned}\dot{x} &= v_p \cdot \cos\theta_p - v_a \\ \dot{y} &= v_p \cdot \sin\theta_p\end{aligned}$$

When the pilot left the rails, when his vertical position is higher than y_r, his motion is retarded by air resistance and gravity. Gravity also influences the flight direction, therefore we have an additional equation for $\dot{\theta}$. This additional acceleration is described by

$$\begin{aligned}\dot{v}_p &= -D\frac{1}{m} - g \cdot \sin\theta_p \\ \dot{\theta}_p &= -g \cdot \cos\theta_p \cdot \frac{1}{v_p}\end{aligned}$$

where m is the mass of pilot and seat, g is gravity and D is a function of velocity, the aerodynamic drag coefficient c, the air density ρ and the effective cross-sectional area a of pilot and seat defined by

$$D = \frac{1}{2}\rho \cdot c \cdot s \cdot v_p^2$$

M_4 contains these two equations and will then be linked to the model of the motion M_2.

In order to summarise the system description, we may write:

state equations

$$\begin{aligned}
\dot{x} &= v_p \cdot \cos\theta_p - v_a \\
\dot{y} &= v_p \cdot \sin\theta_p \\
\dot{v}_p &= \begin{cases} -D\frac{1}{m} - g \cdot \sin\theta_p & \text{if rails left} \\ 0 & \text{if in rails} \end{cases} \\
\dot{\theta}_p &= \begin{cases} -g \cdot \cos\theta_p \cdot \frac{1}{v_p} & \text{if rails left} \\ 0 & \text{if in rails} \end{cases}
\end{aligned}$$

initial values

$$\begin{aligned}
x(t_0) &= x_0 \\
&\vdots
\end{aligned}$$

event condition function

$$y \geq y_r$$

event: add components

When simulating the system, we start with M_1 and M_2. In the first phase, when the pilot gets ejected, we have to link the parameters of M_3 to M_2:

$$\{pilotvel.C_1M_3/pilotvel.C_1M_2, pilottheta.C_1M_3/pilottheta.C_1M_2\}$$

When he leaves the rails the next phase of the motion starts. So we formulate the condition for this event whith

$$C1 = \{y \geq y_r\}$$

where y_r denote the end of the rails.

The corresponding action is to remove the links between M_3 and M_2 $(A1)$ and create links to M_4 $(A2)$

$$\begin{aligned} A1 &= \{-pilotvel.C_1M_3/pilotvel.C_1M_2, \\ &\qquad -pilottheta.C_1M_3/pilottheta.C_1M_2\} \\ A2 &= \{pilotvel.C_1M_4/pilotvel.C_1M_2, \\ &\qquad pilottheta.C_1M_4/pilottheta.C_1M_2\} \end{aligned}$$

Then the constant values get replaced by a new component that calculates these values with the function described above. With this new "model" we can simulate the fall of the pilot with his seat.

8.4.6 Reconditioning Plant

This reconditioning plant is part of a project about a plant for combusting sludge. It consists of two tanks where the concentration of some pollutants is diminished due to different processes, e.g. the addition of air.

The plant consists of two tanks. The first gets filled with sludge. When the concentration of the observed pollutants goes under a certain threshold, the sludge of the first tank is pumped into the second. Before, sludge of the second tank has to be pumped into the combustion unit. Finally, the first tank gets refilled again.

Therefore, this model consists of two tanks where the decrease of the concentration of the pollutes progresses (M_1 and M_2). In addition, there are the processes of emptying the second tank (M_3) the pumping of sludge from the first to the second tank (M_4) and the refilling of the first tank (M_5). Figure 8.3 illustrates these processes.

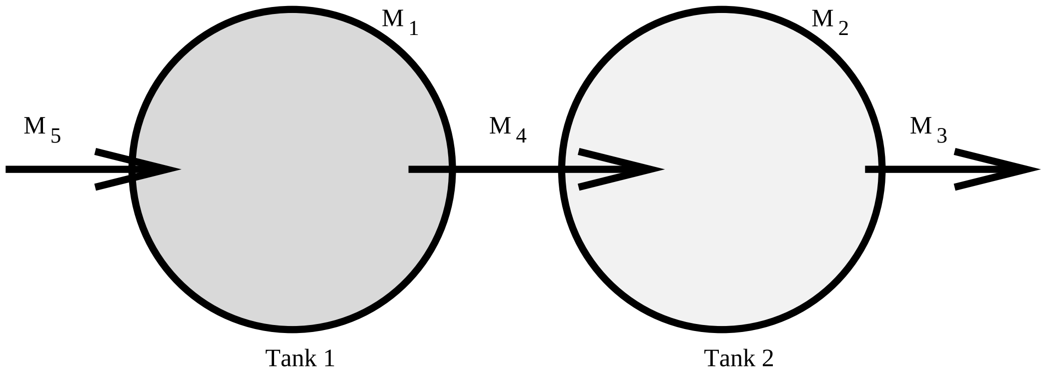


Figure 8.3: Reconditioning Plant

We start the simulation of the system. The condition $C1$ for pumping some sludge from the second tank to the combustion unit ($A1$) is that the concentration of the pollutants is below a certain threshold. So we have to link M_1 and M_3. A similar condition $C2$ is necessary for the action $A2$ of pumping sludge from the first tank to the second. In addition, we have to check if there is some capacity left for more sludge. And also for the third action $A3$ we have to check whether there is enough space left ($C3$).

$$\begin{aligned}
C1 &= \{conc2 < threshold2\} \\
A1 &= \{out.C_1M_3/in.C_1M_2, out.C_1M_2/in.C_1M_3\} \\
C2 &= \{(conc1 < threshold1) AND (cap2 > 0\} \\
A2 &= \{out.C_1M_1/in.C_1M_4, out.C_1M_4/in.C_1M_1, out.C_1M_4/in.C_1M_2\} \\
C3 &= \{cap1 > 0\} \\
A3 &= \{out.C_1M_1/in.C_1M_5, out.C_1M_5/in.C_1M_1\}
\end{aligned}$$

The conditions for stopping these actions of pumping are similar to the ones above and are created by adding minus signs at the corresponding places.

For more efficiency of these pumping processes we should add some more conditions and insert some threshold so that the pumps only start when it pays off.

In this system we have a combination of parallel and sequential processes. For instance, while the sludge is pumped the decrease of concentration of the pollutants is going on.

8.5 Other Application

The presented concept can now be applied also to a wider range of problems than only to state events.

8.5.1 Repeated Linearisation

Sometimes it is necessary to replace a nonlinear system by a linear one. A good method to achieve this is to use Jacobi matrices. Regarding a system $\dot{x} = f(x,t)$ we can transform it to a linear system

$$\dot{x}_L = J(x_P)(x_L - x_P)$$

with $J(x_P)$ being the Jacobi matrix, evaluated at a fixed "point", the centre of expansion x_P

$$J(x_P) = \frac{\partial f}{\partial x}(x_P) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_P) & \dots & \frac{\partial f_1}{\partial x_n}(x_P) \\ \vdots & \dots & \vdots \\ \frac{\partial f_n}{\partial x_1}(x_P) & \dots & \frac{\partial f_n}{\partial x_n}(x_P) \end{pmatrix}$$

Obviously, this linearised system is a good approximation to the original system only in a certain surrounding of the centre of expansion. And here we

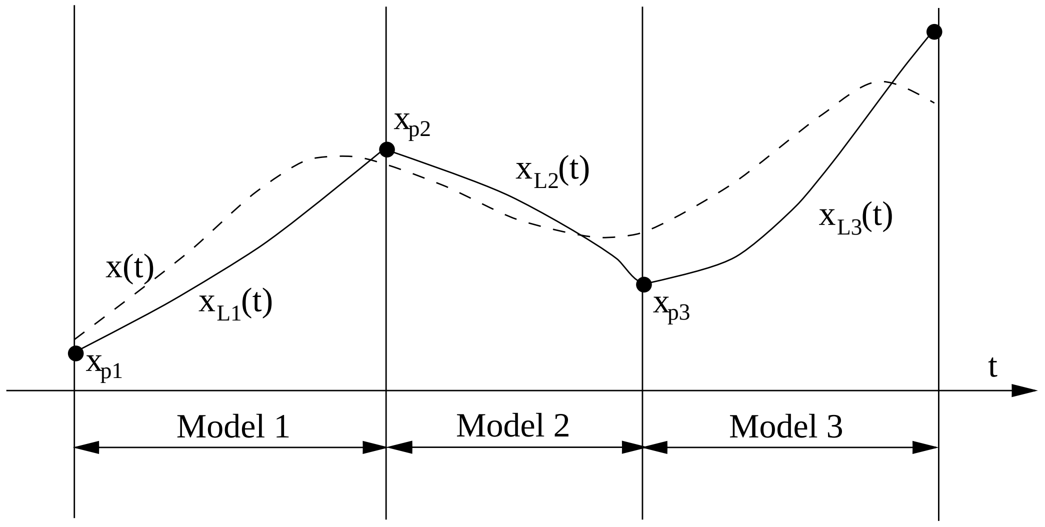


Figure 8.4: Repeated linearisation

can make use of the presented concept as we may formulate the relinearisation as a state event. We regard the distance to the centre of expansion x_P as an event condition function, insert a certain threshold, depending on the eigenvalues of the system. When the function crosses the threshold we trigger the event "relinearisation". That means that we stop the dynamic calculations, re-evaluate the Jacobi matrix and start the calculations again with the new system.

This process is shown in figure 8.4. One line shows the solution calculated with linearised functions. The dotted line denotes the exact function $x(t)$ (that we will never know). The vertical lines mark the points of linearisation. In between of two lines the solution is calculated with the linearised model until, at the next line, the model gets linearised again. The new centre of expansion is then the final value of the previous calculations.

Formalising this process we write for the condition, i.e. that the distance between the centre and x gets bigger than a threshold a:

$$C1 = \{\|x - x_P\| > a\}$$

Then the action is to relinerise the model M_1. This is described in a separate model M_2. So the action is to start the "linearisation" model that updates M_3, the linearised version of model M_1:

$$A1 = \{init(C_1.M_3 = C_1.M_2(newcentre)\}$$

8.5.2 Integration Algorithms – Stiff Systems

During a simulation experiment it will sometimes be necessary to change the applied integration algorithm. Dealing with "stiff systems" we will get involved in such a situation quite quickly.

When a system is *stiff* is a question that cannot be easily answered. Lambert [36, p. 216] says that stiffness is more a *phenomenon* than a *property* since it cannot be defined in precise mathematical terms. Nevertheless, he tried to give a "definition" in the following way [36, p. 220]

Definition: If a numerical method with a finite region of absolute stability, applied to a system with any initial conditions, is forced to use in a certain interval of integration a steplength which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be stiff in that interval.

The comparison with the *exact solution* is in most of the cases impossible since it is simply unknown. So I will take a different definition here that does not cover the whole phenomenon but can be taken as an approach.

This widely used definition of stiffness is to look at the eigenvalues λ_i of the linearisation of a system $\dot{x} = f(t, x)$ at a fixed point. The λ_i are obtained by calculating the eigenvalues of the Jacobi matrix that is built up as I have demonstrated above. With

$$\lambda_{max} = \max_i\{\lambda_i\} \quad \text{and} \quad \lambda_{min} = \min_i\{\lambda_i\}$$

Then we can state that *a system is said to be stiff if all its eigenvalues have a negative real part and λ_{max} and λ_{min} differ a lot.*

The sum of the terms $ae^{\lambda_i t}v_i$ with a being a constant and v_i the corresponding eigenvector is part of the solution of this linearised system.

Since we have required that the real part of the eigenvalues is negative, all the terms tend to zero,

$$\lim_{t\to\infty} e^{\lambda_i t}v_i = 0$$

When now λ_i is large we get a term that represents a *fast* part of the solution, a term that tends fast to zero. When λ_i is small then it decays *slowly*.

When we now apply a numerical integration algorithm to this system, we have to adjust the steplength to the fastest term of the solution. On the other side, we have to consider what the stability region of the integration algorithm looks like. If we use Runge-Kutta algorithms we face finite regions of stability that means that the absolute value of the product of steplength and largest eigenvalue has to be smaller than a specific constant. For a Runge-Kutta method of order this constant is about 2.785, for Runge-Kutta-Fehlberg 45 it is 3.282.

Therefore, when λ_{max} is large the steplength has to be adjusted very small. We have to bear in mind that the small steplength is necessary because of a term that does not really contribute to the solution.

But the small steplength may lead us to numerical problems and unreliable results. In this situation it is necessary to change to a different integration algorithm, to an explicit algorithm of higher order or better to implicit ones. Implicit algorithms may even have an infinite region of stability. The costs for this advantage are longer computing times as more operations for one step are needed.

Therefore, it is worth swapping to a different algorithm only if necessary. This can again be formulated within the meta model concept as an event. The condition for the event is when the difference of the eigenvalues exceeds a certain threshold. In other words, the event is triggered when the ratio between the maximum and the minimum of the eigenvalues exceed a certain threshold. These eigenvalues are only valid locally around the point of linearisation. Like in the application described above they have to be updated from time to time. So when the distance to the last point of linearisation is bigger than a certain threshold, the system gets linearised again and so the eigenvalues can be refreshed.

In this presented application of the *meta model concept* we again merge the descriptions of the model and the experiment. But the separation of these two parts as it is proposed by Zeigler [64] is not left completely. In the models we still can find the separation. Only the *meta model* consists also of statements that are part of the experiment. Here, the decision of which integration algorithm to take is made in this special model.

8.6 Implementation of the Meta Model Concept

For an Implementation of the *Meta Model Concept* it is necessary to adapt especially the experiment level for new tasks. The experiment level is seen here also as an acting module for system descriptions. It is also required that it provides some capabilities for calculating.

As mentioned above, we can extract three main requirements to a simulator where the concept gets implemented:

Dynamic linking: the possibility to create and remove links between models during the simulation.

Independent calculation: a real parallel processing and the necesseary data flow management.

Exact time management: when event occurs, the different models must be synchronised, in order to minimise the problems due to timing errors

Though it is desirable that this new concept gets implemented in its entirety I want to give in this section possibilities to apply parts of the concept in different simulators.

ACSL/ACSL MATH

In ACSL I see no possibility to implement the meta model concept. In combination with ACSL MATH we can define at least static systems that are built up with several models. The models can then be simulated sequentially. The control is done in the ACSL MATH environment. Here we also have a good mathematical functionality for calculations between the runs.

A dynamic linking cannot be described in this language. Another constraint is that in every ACSL model there has to exist at least one state variable. So there is no chance for adding e.g. *interface models*.

SIMNON

In SIMNON we can implement a statically linked system. Here there are features for simulating models in parallel. By using the `CONNECTING SYSTEM` we can link the models even in a very sophisticated way as we may use `IF` statements. I used this method in the model of the bouncing ball with spring damper system in chapter 7.1.2 where I described the component exchange with the following line

```
sd[ballequ]= IF x[ballequ] < r THEN sd[ballcomp] ELSE 0.
```

So I can choose the component that is linked but the component that is not linked continues the calculations even if it is not necessary.

When we combine a `MACRO SYSTEM` at experimental level with a `CONNECTING SYSTEM` we can even define a semi-dynamic structure. Semi dynamic structure means, that we define multiple `CONNECTING SYSTEM`s that we load from the

`MACRO SYSTEM` and simulate in this way a series of systems with different connection structures.

Constraints are given due to the restricted mathematical capabilities of the macro level. But here we have a real parallel system at our disposal.

MATLAB/SIMULINK

In MATLAB/SIMULINK we can describe our models in SIMULINK but also in MATLAB. In SIMULINK models we can only use static structures. A semi dynamic structure can be achieved when using MATLAB "models" as M-Files in a SIMULINK model. With this method of "dirty programming" we can make use of some of the special features in SIMULINK like the "scalar expansion". Otherwise we have no possibilities to simulate models in parallel.

Besides, we can also simulate SIMULINK models in the MATLAB environment. But here also we may only simulate the models sequentially.

MOSIS

In MOSIS the *Model Interconnection Concept* is implemented and therefore we start from a good basis. We find here features for parallel simulation.

On the runtime level we can create *instances* of the models and link them with the `connect` statement. Removing links can be done with a `disconnect` statement.

We also can assign variables in these *instances* by simply typing

```
inst.var=value
```

for assigning `value` to the variable `var` of the instance `inst`.

In addition, there are `if` statements with which we can build up models with a sophisticated structure.

On the runtime level we also can define special functions containing "C"-code or statements of the generic runtime system language. Furthermore, there are the commands for starting and stopping the simulation of the *instances*:

```
run(inst)
stop(inst)
```

These statements also stop or start all the other *instances* that follow the *instance* in the command, in other words that get data from it.

Besides, there is also a `reinit` statement that reinitialises an *instance* of a model.

Applying these possibilities we can build up dynamically linked models. Restricitions are due to the limited capabilities of the experiment level, especially the algorithmical features, that are for example the algorithms for the location of the state events, and for linearisation.

Bibliography

[1] —, *ACSL/Graphic Modeller User's Guide,* MGA Software, 1994

[2] —, *ACSL Reference Manual, Edition 11,* Mitchell and Gauthier Associates Inc., Concord MA 01742 USA, 1995

[3] —, *ACSL MATH User's Guide, Version 1 for Windows,* Mitchell and Gauthier Associates Inc., Concord MA 01742 USA, October 1996

[4] —, *ESL User Manual,* ISIM Simulation, April 1992

[5] —, *MATLAB Release Notes Version 4.2,* The Math Works Inc., Natick, Mass., USA, 1994

[6] —, *MATLAB Reference Guide,* The Math Works Inc., Natick, Mass., USA, 1992

[7] —, *MATLAB User's Guide,* The Math Works Inc., Natick, Mass., USA, 1992

[8] —, *SIMNON for Windows*, Version 1.0, May 1993, SSPA Systems Göteborg, 1993

[9] —, *SIMULINK 1.3 Release Notes,* The Math Works Inc., Natick, Mass., USA, May 1994

[10] —, *SIMULINK User's Guide,* The Math Works Inc., Natick, Mass., USA, 1992

[11] ANDERSSON, M., *Omola - An Object-Oriented Language for Model Representation,* in: 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD), Tampa, Florida, 1989

[12] ANDERSSON, M., *Object-Oriented Modeling and Simulation of Hybrid Systems,* Dept. of Automatic Control, Lund Institute of Technology, 1994

[13] BREITENECKER, F., *Klassifizierung und Umsetzung von Ereignisbeschreibungen und Ereignisbehandlungen in kontinuierlichen Simulationssprachen,* in: Kampe, G. (ed.), Simulationstechnik, 9. Symposium in Stuttgart, Oktober 1994, Vieweg, Braunschweig/Wiesbaden, 1994, pp.189-194

[14] BREITENECKER, F., *Comparison 7 – ACSL,* in: Breitenecker, F., Husinsky, I. (eds.), Comparison of Simulation Software, EUROSIM – Simulation News Europe, no. 8, July 1993, p. 30

[15] BREITENECKER, F., ECKER, H., BAUSCH-GALL, I., *Simulation mit ACSL,* Vieweg, Braunschweig/Wiesbaden, 1993

[16] BREITENECKER, F., HUSINSKY, I. (eds.), *Comparison of Simulation Software*, EUROSIM – Simulation News Europe, no. 0-17, 1990-1996

[17] BREITENECKER, F., SOLAR, D., *Models, Methods and Experiments - Modern Aspects of Simulation Languages,* in: Proceedings of the 2nd Simulation Congress, Antwerpen, Belgium, Sept 9-12, 1986, pp. 195-199

[18] CARVER, M.B., *Efficient Integration over Discontinuities in Ordinary Differential Equation Simulation,* Mathematics and Computers in Simulation XX, 1978, pp. 190-196

[19] CELLIER, F.E., *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools,* Dissertation, Swiss Federal Institute of Technology Zurich, 1979

[20] CELLIER, F.E., *Continuous System Modeling,* Springer-Verlag, New York, 1991

[21] CROSBIE, R.E., HAY, J.L., *Description and processing of discontinuities with the ESL simulation language,* in: CELLIER, F.E. (ed.), Proceedings of the Conference on Continuous System Simulation Languages, San Diego, California, USA, 23.-25.1.1986, pp.30-33

[22] CROSBIE, R.E., HAY, J.L., *Towards new Standards for Continuous System Simulation Languages,* in: Proceedings of the SCSC 1982, SCSi San Diego, pp.186-190

[23] ELMQVIST, H., ÅNGSTRÖM, K.J., SCHNTHAL, T., WITTENMARK, B., *SIMNON Simulation of Nonlinear Systems - User's Guide for MS-DOS Computers*, Version 3.2, SSPA Systems Göteborg, 1993

[24] ELMQVIST, H., CELLIER, F.E., OTTER, M., *Object-Oriented Modeling of Hybrid Systems,* in Proceedings of the ESS'93 European Simulation Symposium, Delft, 1993

[25] ELMQVIST, H., *DYMOLA - Dynamic Modeling Language - User's Manual*, Version 1.9.11, DYNASIM AB, Lund, 1992 *SIMNON Simulation of Nonlinear Systems - User's Guide for MS-DOS Computers*, Version 3.2, SSPA Systems Göteborg, 1993

[26] ENGELN-MÜLLGES, G., REUTTER, F., *Numerische Mathematik für Ingenieure,* Bibliographisches Institut, Mannheim/Wien/Zürich, 4. Aufl., 1985

[27] FASOL, K.H., DIEKMANN, K. (eds.), *Simulation in der Regelungstechnik,* Springer-Verlag, Berlin, 1990

[28] GEAR, C.W., *The Automatic Integration of Ordinary Differential Equations,* Communications of the ACM, 14, No. 3, 1971, pp.176-179

[29] GEAR, C.W., ØSTERBY, O., *Solving Ordinary Differential Equations with Discontinuities,* ACM Transactions on Mathematical Software, Vol. 10, No. 1, March 1984, pp. 23-44

[30] GLADWELL, I., SAYERS, D.K. (eds.), *Computational Techniques for Ordinary Differential Equations,* Academic Press, London, 1980

[31] GOLDYNIA, J.W., *ANA 2.x,* ftp://ftp.iert.tuwien.ac.at/ana2

[32] GOLDYNIA, J.W., *Comparison 7 – ANA 2.x,* in: Breitenecker, F., Husinsky, I. (eds.), Comparison of Simulation Software, EUROSIM – Simulation News Europe, no. 16, March 1996, p. 35

[33] HAY, J.L., CROSBIE, R.E., CHAPLIN, R.I., *Integration Routines for Systems with Discontinuities,* The Computer Journal, Vol. 17, No. 3, 1974, pp. 275-278

[34] HESSEL, E., MELZIG, M., *VHDL-A – Erste Erfahrungen mit dem neuen Standard,* in: KRUG, W. (ed.), Fortschritte in der Simulationstechnik, 10. Symposium in DRESDEN, Sept. 1996, Vieweg, Wiesbaden, 1996, pp. 437-442

[35] KORN, G.A., WAIT, J.V., *Digital Continuous-System Simulation,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978

[36] LAMBERT, J.D., *Numerical Methods for Ordinary Differential Systems - The Initial Value Problem,* Wiley, Chichester, 1991

[37] MARKOWITSCH, J., *Modellbeschreibung in Simulationssprachen – Ein Vergleich anhand von ACSL, ESL, SIMNON und SIMULINK,* Diplomarbeit, TU Wien, 1994

[38] MATTSSON, S.E., *Modelling of Power Systems in Omola for Transient Stability Studies,* CACSD'92, Dept. of Automatic Control, Lund Institute of Technology

[39] MATTSSON, S.E., ANDERSSON, M., *The Ideas Behind Omola,* CACSD'92, Dept. of Automatic Control, Lund Institute of Technology

[40] MATKO, D., ZUPANCIC, B., KARBA, R., *Simulation and Modelling of Continuous Systems - A Case Study Approach,* Prentice Hall, 1992

[41] MURRAY-SMITH, D.J., *Continuous System Simulation,* Chapman&Hall, London, 1995

[42] MURRAY-SMITH, D.J., *Comparison 7 – SLIM,* in: Breitenecker, F., Husinsky, I (eds.), Comparison of Simulation Software, EUROSIM – Simulation News Europe, no. 13, March 1995, p. 36

[43] NIKOUKHAH, R., STEER, S., *SCICOS a dynamic system builder and simulator – User's Guide – Version 0.1,* INRIA, ftp.inria.fr

[44] O'REGAN, P.G., *Step size adjustment at discontinuities for fourth order Runge-Kutta methods,* The Computer Journal, Vol. 13, No.4, Nov. 1970, pp.401-404

[45] ORTEGA, J.M., POOLE, W.G., *An Introduction to Numerical Methods for Differential Equations,* Pitman Pub. Inc., Marshfield, Mass., 1981

[46] OTTER, M., *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter,* Fortschrittberichte VDI, Reihe 20: Rechnerunterstützte Verfahren, Nr.147, VDI-Verlag, Düsseldorf, 1995

[47] OTTER, M., GRÜBEL, G., *Direct Physical Modeling and Automatic Code Generation for Mechatronics Simulation,* Second Conference on Mechatronics and Robotics, Duisburg, 27.-29.09.1993

[48] PICHLER, F., SCHWÄRTZEL, H. (eds.), *CAST Methods in Modelling,* Springer, 1992, pp.123-241

[49] PLANK, J. , *Algorithmenstruktur in Simulationssprachen – Ein Vergleich anhand von ACSL, ESL, SIMNON und SIMULINK,* Diplomarbeit, TU Wien, 1994

[50] PRÄHOFER, H., *System Theoretic Foundations for Combined Discrete-Continuous System Simulation,* Dissertation, Uni-Linz, 1991

[51] PRÄHOFER, H., AUERNIG F., REISINGER, G., *An Environment for DEVS-Based Multiformalism Simulation in Common Lisp/CLOS,* Discrete Event Dynamic Systems: Theory and Applications 3, 1993, pp.119-149

[52] PRÄHOFER, H., BICHLER, P., ZEIGLER, B., *Synthesis of Endomorphic Models for Event-Based Intelligent Control Employing Combined Discrete/Continuous Simulation,* IEEE 1993, pp. 120-126

[53] PRÄHOFER, H., JAHN, G., JACAK, W., HAIDER, G., *Supervising Manufacturing System Operation by DEVS-Based Intelligent Control,* IEEE 1994 Institute of System Science, U.Linz

[54] PRÄHOFER, H., PREE, D., *Visual Modeling of DEVS-Based Multiformalism Systems Based on Highgraphs,* Proceedings of the 1993 Winter Simulation Conference, pp. 595-603

[55] PRÄHOFER, H., REISINGER, G., *Distributed Simulation of DEVS-Based Multiformalism Models,*

[56] RECHENBERG, P., *Die Simulation kontinuierlicher Prozesse mit Digitalrechnern,* Vieweg, Braunschweig/Wiesbaden, 1972

[57] SCHUSTER, G., *Definition and Implementation of a Model Interconnection Concept in Continuous Simulation,* Dissertation, TU-Wien, 1994

[58] SCHUSTER, G., *MOSIS - The Modular Interconnection System, User's Guide, Version 1.02,* ATS/ARGESIM, Vienna, Austria, 1994

[59] SCHUSTER, G., *MOSIS - downloadable full version*, ARGESIM, ftp://argesim.tuwien.ac.at

[60] SCHUSTER, G., BREITENECKER, F., *Comparison 7 – MOSIS,* in: Breitenecker, F., Husinsky, I. (eds.), Comparison of Simulation Software, EUROSIM – Simulation News Europe, no. 12, November 1994, p. 31

[61] STRAUSS, J.C. et al., *The Sci Continuous System Simulation Language (CSSL),* Simulation, vol. 9, no. 5, Dec. 1967

[62] TROCH, I., *Gewöhnliche Differentialgleichungen,* Skriptum, TU-Wien, 1991

[63] ÜBERHUBER, C., *Computer-Numerik I+II,* Springer-Verlag, Berlin, 1995

[64] ZEIGLER, B. P., *Theory of Modelling and Simulation,* John Wiley, New York, 1976

Index

ACSL, 50, 67, 75, 136
All-in-one method, 79, 94
ANA 2.x, 72

block on a rough surface, 99, 124
  ACSL model
    Discrete Section method, 100
  SIMNON model
    Concatenated Runs method, 103
block-oriented, 12
bouncing ball, 24, 119
  macroscopic view, 25, 122
  microscopic view, 34, 92, 119

CASE, 73
classification - 2 classes, 30
classification - 4 classes, 32
COMPARTMENT, 77
component, 34, 86
Component Exchange, 86
Component Exchange method, 95
Concatenated Runs method, 84, 103
CONNECTING SYSTEM, 66, 95, 136
continuous binary search, 42, 73
CONTINUOUS SYSTEM, 66, 95
CSSL-standard, 14, 50, 59, 63, 70
CTERM, 66, 103

DERIVATIVE, 51, 70
discontinuity, 22
DISCRETE, 51, 100
Discrete Section method, 82, 100
DISCRETE SYSTEM, 66
DYMOLA, 75
DYNAMIC, 61

equation-oriented, 12
ESL, 59
event
  description, 36, 115
  detecting, 38
  handling, 37, 115
  locating, 40
  passing, 45
event condition function, 36
event-definition, 21
event-state event, 21
event-time event, 21
experiment, 17, 90, 111
experiment level, 85
Extended Experiment method, 89, 108

Hit-Crossing-Block, 54

interface model, 111

linearisation, 131

MACRO SYSTEM, 66, 96, 103, 136
MATLAB, 53, 67, 137
meta model, 111
Meta Model Concept, 111
method of false position, 45
model, 107, 111
Model Interconnection Concept, 107
ModelMaker, 77
MOSIS, 63, 137
MULTIPLEXER, 73

Newton's method, 44
numerical integration algorithms, 16

ONRISE - DO, 73

parameter, 13
pendulum, 26
  constrained pendulum, 32, 48, 124
    ACSL model, 52
    ANA 2.x model, 73
    DYMOLA model, 76
    ESL model, 62, 64
    ModelMaker model, 78
    SIMNON model, 67
    SIMULINK 1. 3c model, 55
    SIMULINK 2. 0 model, 58
    SLIM model, 70
  looping pendulum, 27, 122
pilot ejection, 127

reconditioning plant, 129
regula falsi, 45, 51, 64
RESET-Integrator, 54
restarting, 46
result model, 111
root locating, 42

SCHEDULE, 51
Sequential Model method, 87
Sequential Models method, 96
SEVENT, 63
SIMNON, 65, 75, 94, 136
simulation, 10
simulator, 11
SIMULINK, 53, 67, 75, 137
SLIM, 69
state space, 12
stiff system, 16, 133
Stop'n'Go method, 84, 103
STOP-Block, 54
structure, 12
  change of structure, 34
study, 17
SWITCH-Block, 54
system
  real system, 10

VHDL-standard, 15

WHEN, 61, 75

Über den Autor …

Dr. Johannes Plank studied Technical Mathematics at the TU Vienna. During his studies, he attended several lectures on simulation techniques, also his diploma thesis dealt with simulation techniques and comparison of simulation software. Under the supervision of Prof. Breitenecker, he started to work on this dissertation in 1995. Since his graduation to the doctoral degree in April 1997, he is now engaged in a project on "Numerical Simulation in Tunnelling" at the Technical University Graz, Austria.

Über diesen Band …

State Events in Continuous Modelling and Simulation deals with state events in continuous systems in a comprehensive way. Starting with an introduction on state events by means of simple examples a classfication of state events is given. Then the book describes systematically and methodically how state events can be formulated in mathematical models and/or simulators. Furthermore, a state-of-the-art report on handling of state events in modern simulators is given.

The second part of the book is devoted to a new approach. Based on a Model Interconnection Concept a Meta Model Concept is presented, which claims to simplify modelling and handling of state events in continuous simulation. The basic idea is to transfer state events from the model level to another level, as well the executive control of the events as well as the description of the events in separate models. As consequence, modelling and implementation of state events, re-usability of models etc. become much more simple.

Über diese Reihe …

Die Bände dieser neuen ASIM - Reihe Fortschrittsberichte Simulation konzentrieren sich auf neueste Lösungsansätze, Methoden und Anwendungen der Simulationstechnik (Ingenieurwissenschaften, Naturwissenschaften, Medizin, Ökonomie, Ökologie, Soziologie, etc.).
ASIM, die deutschsprachige Simulationsvereinigung (Fachausschuss 4.5 der GI - Gesellschaft für Informatik) hat diese Reihe ins Leben gerufen, um ein rasches und kostengünstiges Publikationsmedium für derartige neue Entwicklungen in der Simulationstechnik anbieten zu können.
Die Fortschrittsberichte Simulation veröffentlichen daher: * Monographien mit speziellem Charakter, wie z. B. Dissertationen und Habilitationen * Berichte zu Workshops (mit referierten Beiträgen) * Berichte von Forschungsprojekten * Handbücher zu Simulationswerkzeugen (User Guides, Vergleiche, Benchmarks), und Ähnliches.
Die Kooperation mit den ARGESIM Reports der ARGESIM vermittelt dabei zum europäischen Umfeld und zur internationalen Publikation.