Diss ETH No 6483

COMBINED CONTINUOUS/DISCRETE SYSTEM SIMULATION BY USE OF DIGITAL COMPUTERS: TECHNIQUES AND TOOLS

A DISSERTATION

submitted to the
SWISS FEDERAL INSTITUTE
of
TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
FRANCOIS EDOUARD CELLIER
Dipl. El.-Ing. ETHZ
born July 30, 1948
citizen of Duebendorf/ZH and La Neuveville/BE

accepted on the recommendation of
Prof. Dr. M. Mansour, referee
Prof. Dr. P. Henrici, co-referee

ADAG Administration & Druck AG

Zurich 1979

To my parents,
in appreciation for the opportunity which they provided
to enable me to reach this level of education, and

To Ursula, my wife,
in recognition of the many hours stolen from our
family life for this writing task.

ACKNOWLEDGMENTS:

The work described herein has been carried out mainly during the years 1974 to 1978 under the supervision of Prof. Dr. Mohammed Mansour, who is the Director of the Institute for Automatic Control. I wish to acknowledge my deep indebtedness to him for all he has done for me. I wish also to express my sincere thanks to Prof. Dr. Peter Henrici of the Seminar for Applied Mathematics who consented to examine this thesis.

Furthermore, I would like to express my thanks to Prof. Dr. A. Alan B. Pritsker of Purdue University (U.S.A.) for the exceedingly valuable cooperation. Many of the ideas expressed in this thesis originated from the pioneer work in combined simulation carried out by Prof. Pritsker and his group. I wish also to acknowledge my indebtedness to Prof. Dr. Tuncer Oren of the University of Ottawa (Canada) for the many good ideas he suggested in several long discussions of the topic. I am not less grateful to Prof. Dr. Bernard P. Zeigler of the Weizman Institute (Israel) for the stimulating comments obtained as a reaction to a paper [1.4] published on the subject earlier this year. I would, furthermore, like to thank Dr. Hilding Elmqvist of the Lund Institute of Technology (Sweden) for his pertinent remarks as to the MODULE concept.

Credit must also be given to my co-workers at the Institute for Automatic Control, and also to other persons outside the Institute who assisted me, either directly or indirectly, during the execution of this work.

Many of my students have contributed remarkably to the development of the software described in this thesis. Among them, special credit must be given to Mr. André E. Blitz for the implementation of the PDE software into GASP-V [1.1],

and to Mr. Antonio P. Bongulielmi who has contributed significantly to the development of COSY in two different projects dealing with the language definition [1.2] and with the compiler construction [1.3].

Finally, I should like to thank Mr. Neil J. Sullivan for his careful review of this manuscript.

I am also grateful to the OTTMAR text editing system, running on our PDP-11/60 installation, for the nice preparation of this manuscript.

TABLE OF CONTENTS:

ABSTRACT:

This thesis describes new techniques for simulating systems with complex structures by use of a digital computer, as well as the requirements of tools (simulation languages) to cope with the problem in a user-friendly way. Emphasis is given to a general applicability of the software, that is, the described software is meant to be able to handle broad classes of problems in a sub-optimal way rather than to be able to treat any specific application problem in a truly optimal manner. The increased software robustness and the highly reduced costs in coding any application problem compensate, however, for the sacrifice of efficiency in executing a particular simulation project.

Although many problems of numerical mathematics and information processing which are discussed in this thesis had to be considered and solved, the approach to them has been from the viewpoint of an engineer rather than from that of a mathematician.

Combined system simulation as it is described in this thesis is a relatively new technique for the simulation of a class of systems having properties suitable to both continuous system simulation and discrete event simulation, two techniques well known to the simulation community. This combined technique has first been proposed by Fahrland ("Simulation", vol.14, no.2, February 1970).

Major techniques and methodologies involved in this simulation approach are surveyed. Special aspects considered are numerical behaviour and information processing. It is shown that this technique is applicable to a much larger class of problems than originally suggested by Fahrland.

Simulation techniques are a very broad topic. The subject

dealt with in this thesis covers many facets -- numerical analysis, ordinary and partial differential equations, formal languages, and software design. Since most readers will have the level of familiarity required with only some of these facets, the thesis tries to provide the reader with all the information necessary for understanding the numerical and information processing aspects involved. However, a basic knowledge of continuous system modeling and discrete event modeling are a prerequisite to the understanding of this thesis. Knowledge of simulation languages (like CSMP-III and/or GASP) eases the reading of the sample programs as presented in this thesis, but is not indispensable since one of the highlights of good simulation software is its documentation value.

ZUSAMMENFASSUNG:

Diese Dissertation beschreibt neue Techniken zur digitalen Simulation von Systemen, welche eine komplexe Struktur aufweisen. Ebenfalls beschrieben werden die Werkzeuge (Simulationssprachen), welche zur Verfuegung gestellt werden muessen, damit der Benuetzer seine Probleme in moeglichst bequemer Weise formulieren kann. Der Schwerpunkt der Ueberlegungen geht dahin, allgemein verwendbare Software zu erstellen. Die beschriebene Software soll daher primaer in der Lage sein, breite Klassen von Problemen in suboptimaler Weise zu bearbeiten. Die Loesung jedes beliebigen Problems wird dadurch langsamer und somit teurer, als wenn ein optimaler, dem Problem angepasster Algorithmus zu diesem Zweck entwickelt wuerde. Der allgemeinen Verwendbarkeit und Robustheit der Software wird daher ein gewisses Mass an Effizienz bei der Behandlung eines speziellen Problems geopfert. Die dadurch erreichte erhoehte Systemsicherheit und die wesentlich reduzierten Kosten bei der Programmierung eines Anwenderproblems wiegen jedoch diesen Nachteil bei weitem auf.

Obwohl viele Probleme der numerischen Mathematik sowie der Informatik erwaegt und geloest werden mussten, zeugt die verwendete Methodik doch eher von einer ingenieurmaessigen als von einer streng mathematisch exakten Betrachtungsweise.

Der Begriff gemischte Simulationstechnik, wie er in dieser Arbeit verwendet wird, umschreibt eine relativ junge Technik zur Simulation einer Klasse von Systemen, welche Eigenschaften sowohl der kontinuierlichen Simulationstechnik wie auch der ereignisorientierten diskreten Simulationstechnik aufweist, beides Techniken, wie sie seit langem in der digitalen Simulation Verwendung finden. Diese neue gemischte Simulationstechnik wurde erstmals von Fahrland ("Simulation", Bd.14, Nr.2, Februar 1970) beschrieben.

Die in der gemischten Simulation hauptsaechlich verwendeten Techniken und Methoden werden diskutiert. Speziell stehen die Probleme der Numerik und Informatik im Vordergrund. Es wird aufgezeigt, dass die hier beschriebenen Methoden auf wesentlich breitere Problemklassen Anwendung finden koennen, als dies urspruenglich von Fahrland vorgeschlagen worden war.

Die Simulationstechnik ist ein sehr breites Gebiet. Viele Aspekte der numerischen Mathematik, der Behandlung gewoehnlicher sowie partieller Differentialgleichungen, der Theorie formaler Sprachen sowie des Softwareentwurfs finden in dieser Dissertation Beachtung. Da die meisten Leser nur mit einigen dieser Aspekte vertraut sein duerften, will diese Dissertation alle zum Verstaendnis der Numerik sowie der Informatik notwendigen Grundlagen vermitteln. Ein Grundwissen betreffend die Modellierung kontinuierlicher sowie diskreter ereignisorientierter Systeme wird allerdings vorausgesetzt. Kenntnisse in Simulationssprachen (wie CSMP-III und/oder GASP) sind nuetzlich fuer das Verstaendnis der Beispiel Programme, die in dieser Dissertation praesentiert werden. Sie sind jedoch nicht unbedingt erforderlich, liegt doch einer der wesentlichsten Vorzuege von Simulationssprachen in deren dokumentarischem Wert.

RESUME:

La présente thèse décrit de nouvelles techniques pour la simulation digitale de systèmes à structure complexe. Les outils (langages de simulation) qui doivent être à disposition pour permettre à celui qui s'en sert de formuler ses problèmes aussi aisément que possible sont également décrits. Le but poursuivi consiste à produire de la software généralement utilisable, c'est-à-dire que la software décrite doit en premier lieu plutôt permettre de résoudre de manière sous-optimale des classes étendues de problèmes que de résoudre de manière optimale certains problèmes spéciaux. C'est pourquoi on préfère renoncer à un certain degré d'efficience dans le traitement d'un problème particulier pour augmenter le champ d'application et la solidité de la software. Ce désavantage est cependant largement compensé par l'augmentation de la sécurité du système qui en résulte et le coût fortement réduit de la programmation d'un problème d'application.

Bien que de nombreux problèmes de mathématiques numériques et d'informatique aient dû être posés et résous, la méthode utilisée considère les choses plus dans l'optique d'un ingénieur que dans celle d'un calcul mathématique strictement exact.

La notion de technique de simulation mixte dont fait usage la présente étude décrit une technique relativement jeune pour la simulation d'une classe de systèmes qui ont des propriétés émanant aussi bien de la technique de simulation continuelle que de la technique de simulation d'événements discrets, techniques qui sont toutes deux utilisées déjà longtemps dans la simulation digitale. Cette nouvelle technique de simulation mixte a été décrite pour la première fois par Fahrland ("Simulation", tome 14, no 2, Février 1970).

Les techniques et méthodes principalement utilisées en simulation mixte sont décrites. Les problèmes des mathématiques numériques et de l'informatique sont en particulier placés au premier plan. Il est démontré que les méthodes décrites dans cette thèse peuvent être appliquées à des classes de problèmes beaucoup plus étendues que celles proposées à l'origine par Fahrland.

La technique de simulation est un domaine très étendu. De nombreux aspects des mathématiques numériques, de la solution d'équations différentielles ordinaires et partielles, de la théorie des langages formels et du développement de software sont pris en considération dans la présente thèse. La plupart des lecteurs ne connaissant probablement que quelques-uns de ces aspects à fond, cette thèse s'efforce d'exposer les bases nécessaires à la compréhension des mathématiques numériques et de l'informatique. Des connaissances fondamentales au sujet de la dérivation de modèles pour des systèmes continuels et d'événements discrets sont toutefois présumées. La connaissance des langages de simulation (tels que CSMP-III et/ou GASP) est utile pour comprendre les exemples de programmes que cette thèse présente. Mais elle n'est pas absolument indispensable, car un des avantages les plus importants des langages de simulation gît dans leur valeur documentaire.

I) INTRODUCTION:

The term "combined simulation" is not yet sufficiently well understood in the literature to mean one and only one specific methodology or problem class. For example, one can find references where combined simulation is used as a synonym for hybrid simulation. This term, therefore, first requires some definition to clarify how it is going to be used in this thesis.

If one speaks of simulation as a technique one usually thinks of a specific solution tool (digital simulation, analog simulation, hybrid simulation). On the other hand the term "system simulation" refers to a specific class of systems under investigation (continuous system simulation, discrete system simulation). However, as early as 1967 Kiviat [1.7, p.5] stated that it is common to find the terms "simulation" and "system simulation" used interchangeably. In this thesis we do not primarily have a specific simulation methodology in mind, but rather the simulation of one specific class of problems which we call combined systems. However, restricting ourselves to fully digital solutions only, simulation of this class of problems does suggest the use of a specific simulation methodology which we are going to discuss in detail. Although the term "combined system simulation" is thus appropriate, we will use the term "combined simulation" as well for simplicity.

It remains to define what the term "combined systems" means precisely. It can be paraphrased as follows:

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

Using this definition, the famous pilot ejection study (which is probably the best known test case for continuous system simulation) will also fall into this class of problems, since the acceleration of the ejector seat and the first time derivative of its angular position (both state derivatives in the system's definition) are discontinuous at the moment when the ejector seat is disengaged from the mounting rails.

The most comprehensive volume on combined simulation published to date [1.8] cites two examples of continuous systems -- the above mentioned pilot ejection study and an analysis of a slip clutch. Both belong to the class of combined systems according to our new definition. This shows that the definition used here is not entirely in accordance with the "common" use of this term (as a matter of fact, a proper definition for this term has never been given!). We must redefine the term "continuous system" as well to keep it consistent with our definition for combined systems.

> Continuous systems are systems described by a fixed set of differential equations with state variables and first state derivatives both being continuous over the whole simulation run.

This definition restricts the term "continuous system" to a more narrow sense than is commonly used. The motivation for redefining these terms will be given in due course.

Considering the different types of systems to be modeled,

one may find that models arising in engineering applications usually show the most complex structures of all. The reason for this comes from the following fact. Physicists and mathematicians, on one hand, often try to idealize their problems as much as possible to be able to obtain analytical results beside of the purely numerical ones (or tend to concentrate on those problems where such an idealization seems legitimate). Natural scientists and economists, on the other hand, usually have only very bad, unreliable, and irreproducible data available to represent their systems, out of which a sophisticated model may not be faithfully postulated [1.6]. In addition to this, economists face often the problems that their systems change their modes of behaviour as they find out that they are modeled ("keep-smiling" effect). They, therefore, normally use models which show a relatively simple structure. However, these are often high order models (models consisting of a large number of coupled differential equations). As examples we may mention the Lotka-Volterra type models which are frequently cited in Biology and Chemistry, or Systems Dynamics models mostly used to describe socio-economic systems. The goal of such models is to gain a qualitative insight into the system's possible modes of behaviour rather than to obtain a precise quantitative information as to how a "real" system will respond to specific control strategies.

Models in engineering, finally, are mostly used for the design of "new" systems, e.g. because constructing the system without a priori knowledge as to whether the system will behave as expected may be too expensive or too dangerous, or because some of the system's parameters are unknown and have to be evaluated by a "try-and-error" strategy which also can be profitably automated (nonlinear programming techniques). For this task, the simulation is supposed to supply a precise quantitative information about the system's behaviour under given experimental conditions. The available data for these systems are usually reliable and reproducible enough

to allow such conclusions to be drawn. The postulated models show complex structures in general. For this reason, most of the so called "continuous" engineering problems belong to the class of "combined systems" according to the definitions stated above.

Simulation packages are usually categorized into one of three classes according to the problems which can be handled by them. These are:

i) continuous-time systems described by ordinary differential equations (ODE's),

ii) continuous-time systems described by partial differential equations (PDE's) and

iii) discrete-time systems, described either by difference equations, by sequences of time-events or by mixtures of both.

It is to be stated that the partitioning of simulation packages into these three classes results from the different simulation techniques involved rather than from the physical world being dividable into three classes of dynamic systems. This can easily be shown viewing the different types of models used in traffic control [1.5]. There one can distinguish between the following three types of models.

a) Macroscopic models which are entirely continuous models, in which the single vehicle is not represented at all. State variables are traffic flow and its accompanying densities. Such models involve sets of coupled PDE's (possibly approximated by ODE's). Appropriate tools for these models can be found in classes (i) or (ii).

b) Microscopic models which are entirely discrete models with vehicles entering the considered system, traveling through it in discrete steps, e.g. from one intersection to the next with queuing situations in front of traffic lights etc.. These models will clearly be coded by use of tools out of class (iii).

c) Submicroscopic models (not often used in this context) which are continuous with specific discontinuities describing the dynamic behaviour of every single vehicle in detail. These models will require tools showing attributes of both classes (i) and (iii), that is a combined continuous/discrete approach.

The physical system, however, is identical in all three cases. The different simulation techniques have been selected in accordance with the model of the system and not with the system itself. The model of the system has been developed in a way suitable for optimal fitting the given simulation objectives. Suitability of one simulation technique is, therefore, an attribute of the selected model rather than of the underlying physical system, and we should thus better talk of "combined models" instead of "combined systems". As explained in [1.9], one should, in any event, not expect to simulate a physical system but a model derived from the physical system via an experimental frame, within which data can be collected representing the behaviour of the real system under specified experimental conditions. Hopefully, under novel experimental conditions, the constructed simulation program will produce data representative of the data to be observed when the real system is observed under these new conditions. However, since we will disregard the problem of modeling in this thesis entirely, we shall use the terms "model" and "system" interchangeably.

In a first part of this thesis, the requirements of a runtime package to carry out simulation runs of combined models

will be discussed. Of major interest in this context are the numerical requirements to guarantee correct system's response, and the structural requirements to guarantee robustness of the software. The former aspect accounts for credibility of results when executing a particular case study, whereas the latter accounts for software reliability under varying system structures and/or experimental frames. In a second part of this thesis, the required tools (simulation languages) enabling the user to code his combined models in a user-friendly manner, will be discussed. In that context, we will concentrate on the aspects of information processing.

References:

[1.1] A.E.Blitz: (1976) "Entwicklung eines universell verwendbaren Simulationspaketes". BS thesis. Internal Report: AIE-76-8263. Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[1.2] A.P.Bongulielmi: (1978) "Definition der allgemeinen Simulationssprache COSY". Senior Project. Internal Report: AIE-78-8325. Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich. To be obtained on microfiches from: The main library, ETH - Zentrum, CH-8092 Zurich, Switzerland. (Mikr. S637).

[1.3] A.P.Bongulielmi: (1979) "COSY Preprocessor --- Deklarationen Parser". BS thesis. Internal Report: AIE-79-8346. Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[1.4] F.E.Cellier: (1978) "Combined Continuous/Discrete System Simulation Languages --- Usefulness, Experiences and Future Development". Proc. of the Symposium on Modeling and Simulation Methodology, Rehovot, Israel. Published by North-Holland Publishing Company (Editors: B.P.Zeigler, M.S.Elzas, G.J.Klir, T.I.Oren); pp. 201 - 220.

[1.5] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor L.Dekker); pp. 391 - 402.

[1.6] W.J.Karplus: (1976) "The Spectrum of Mathematical Modeling and System Simulation". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 5 - 13.

[1.7] P.J.Kiviat: (1967) "Digital Computer Simulation: Modeling Concepts". Form: RM-5378-PR, The Rand Corp., Santa Monica, CA, U.S.A..

[1.8] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

[1.9] B.P.Zeigler: (1976) "Theory of Modelling and Simulation". John Wiley.

II) HISTORICAL DEVELOPMENT:

Program packages for the treatment of problems of class (iii), e.g. simulation of inventory systems or of queuing situations, have existed for more than 20 years. The most commonly used discrete simulation languages today are: GPSS-V [2.16], SIMSCRIPT-II.5 [2.5], and SIMULA-67 [2.4]. Programming languages for class (iii) problems are surveyed in [2.9].

Problems of type (i) have been solved by use of digital computers for the last 15 years. Currently used continuous simulation languages evolved from the early digital-analog-simulators which tried to imitate analog computers' capabilities under digital environment. They obtained a more or less standardized form in 1967 by the CSSL-committee [2.21]. All currently used continuous simulation languages follow more or less these recommendations. These languages are e.g. surveyed in [2.3,2.11]. Typical examples are: ACSL [2.10], CSMP-III [2.22] and DARE-P [2.8,2.19].

Packages for the numerical solution of problems of class (ii) were first announced about 9 years ago. However, an optimal package for this purpose has not been developed since these problems are numerically much more delicate than ODE problems. This can be shown by the following:

Let us define the efficiency of an integration algorithm to solve a particular problem as:

$$\eta(A,P) = \frac{\text{CPU-time}(A^*,P)}{\text{CPU-time}(A,P)}$$

where: A = algorithm in question
A* = best suited algorithm
P = problem to be solved.

For the solution of ODE problems η very rarely takes a value less than 0.01 for any application problem and any available integration algorithm except for the case of extremely stiff or highly oscillatory problems. Moreover, in considering ODE problems, possible techniques simply involve different integration algorithm. Otherwise the structure of the simulation program remains unchanged. In the PDE case, the situation is quite different. Looking, for example, only at those methods which are commonly used for the solution of elliptic PDE problems [2.14], one can see that η easily can take values on the order of 10^{-3} to 10^{-6}. Moreover, the applied methods are so much different from each other that it seems almost impossible to integrate them into one single computer program. Thus, in problem class (ii) there exists a much stronger link between the problems to be solved and the optimal algorithms to be used than in problems of classes (i) or (iii), and selection of the optimal algorithm becomes vital.

For the reasons mentioned above, it is very doubtful whether it will be possible at all to develop a package which could truly be called a "general purpose" package for the solution of PDE problems. An optimal state, however, has certainly not yet been achieved. Development goes on with the increased possibilities of new computer technologies (shorter cycle time, larger memory). Packages for certain subclasses of (ii) are e.g.: DSS [2.20] and LEANS-III [2.15]. FORSIM-VI [2.2] can be used for the solution of combined ODE and PDE problems. Surveys are given in [2.1,2.11].

Prior to 1968 it was believed that all problems to be solved could be properly classified into the three classes (i) to (iii). As recently as 1969 an eminent specialist of simu-

lation techniques wrote: "It is the considered opinion of the author that at present there is relatively little need for a combined discrete event simulation/continuous-system simulation language facility" [2.18]. The first articles proposing a combined continuous/discrete simulation approach were [2.6,2.7]. To date there already exist references to more than 30 different software packages for combined simulation. These will be discussed later in this thesis. Looking at their historical background, one may find that most of the languages and packages for combined system simulation available on the software "market" are extensions of existing "pure" discrete simulation languages/packages. As examples we may mention:

GASP-II [2.13] --> GASP-IV [2.12]
SIMSCRIPT-II.5 [2.5] --> C-SIMSCRIPT [2.5]
SIMULA-67 [2.4] --> CADSIM [2.17].

The reason for this is the following: Although a numerically well performing package for continuous system simulation is much more difficult to achieve than one for discrete system simulation, the structural concepts for the latter are much more complex than for the former. Thus, extending discrete simulation packages to encompass combined problems is a much easier task to achieve than extending a continuous simulation package for that purpose.

Extensions of discrete simulation packages have been implemented, in most cases, either by the original designers or, at least, by former users of the original software. However, these people, usually having a background in operations research, normally do not consider the requirements of systems' analysts for continuous systems from either the numerical or information point of view. For example, one may find that one specific integration algorithm has been coded into the control routine, or that no provision has been made for parallel structures, adequate run-time control pro-

cedures, etc..

We have already seen that most so-called "continuous" systems (at least in engineering) are really "combined" systems according to our definition. On the other hand there exist many systems, which may be conveniently described by purely discrete simulation elements. As a result there is a much greater impact of combined simulation on the treatment of continuous systems than of discrete systems. This allows one to conclude that the state-of-the-art of combined system simulation languages is by no means satisfactory.

References:

[2.1] M.B.Carver: (1975) "Simulation Packages for the Solution of Partial Differential Equation Systems". Proc. of the SIMULATION'75 Symposium, Zurich, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 57 - 64.

[2.2] M.B.Carver: (1978) "The FORSIM-VI Simulation Package for the Automated Solution of Arbitrarily Defined Partial and/or Ordinary Differential Equation Systems". Form: AECL-5821. Atomic Energy of Canada, Ltd.; Chalk River Nuclear Laboratories, Mathematics & Computation Branch, Chalk River, Ontario, Canada K0J 1J0.

[2.3] F.E.Cellier: (1975) "Continuous-System Simulation by Use of Digital Computers: A State-of-the-Art Survey and Prospectives for Development". Proc. of the SIMULATION'75 Symposium, Zurich, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 18 - 25.

[2.4] O.J.Dahl: Nygaard K.: (1966) "SIMULA; A Language for Programming and Description of Discrete Event Systems". Oslo, Norwegian Computing Center.

[2.5] C.M.Delfosse: (1976) "Continuous Simulation and Combined Simulation in SIMSCRIPT-II.5". To be ordered from: C.A.C.I., Inc., 1815 North Fort Myer Drive, Arlington VA 22209, U.S.A..

[2.6] D.A.Fahrland: (1968) "Combined Discrete Event / Continuous System Simulation". MS Thesis, Systems Research Center Report SRC-68-16, Case Western Reserve University, Cleveland, Ohio, U.S.A..

[2.7] D.A.Fahrland: (1970) "Combined Discrete-Event Continuous System Simulation". Simulation vol.14 no. 2 : February 1970; pp. 61 - 72.

[2.8] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice Hall.

[2.9] W.Kreutzer: (1976) "Comparison and Evaluation of Discrete Event Simulation Programming Languages for Management Decision Making". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 429 - 438.

[2.10] E.E.L.Mitchell, Gauthier J.S.: (1976) "ACSL: Advanced Continuous Simulation Language - User/Guide Reference Manual". To be ordered from: Mitchell and Gauthier, Assoc., 1337 Old Marlboro Road, Concord MA 01742, U.S.A..

[2.11] R.N.Nilsen, Karplus W.J.: (1974) "Continuous-System Simulation Languages - A State-of-the-Art Survey". Annales de l'Association Internationale pour le Calcul Analogique (AICA), no. 1, January 1974; pp. 17 - 25.

[2.12] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

[2.13] A.A.B.Pritsker, Kiviat P.J.: (1969) "Simulation with GASP-II". Prentice Hall.

[2.14] J.R.Rice: (1976) "Algorithmic Progress in Solving Partial Differential Equations". SIGNUM-Journal, vol. 11, no. 4, December 1976, (Special Interest Group on Numerical Mathematics of ACM); pp. 6 - 10.

[2.15] W.E.Schiesser: (1971) "LEANS-III: Introductory Programming Manual". To be ordered from: Computing Center, Lehigh University, Bethlehem PA 18015, U.S.A..

[2.16] T.J.Schriber: (1974) "Simulation using GPSS". John Wiley.

[2.17] R.J.W.Sim: (1975) "CADSIM - User's Guide and Reference Manual". To be ordered from: Department of Computing and Control, Imperial College, London SW7 2BZ, England.

[2.18] J.C.Strauss: (1969) "Discrete Event and Continuous System Simulation Languages: A Critical Comparison". In E.Sevin, Editor, Computational Approaches in Applied Mechanics (American Society of Mechanical Engineers), New York; pp. 50 - 59.

[2.19] J.V.Wait, DeFrance Clarke III: (1976) "DARE-P User's Manual". (Version 4.1). To be ordered from: Department of Electrical Engineering, University of Arizona at Tucson, Tucson AZ 85721, U.S.A..

[2.20] M.G.Zellner: (1970) "DSS: Distributed System Simulation". Ph.D. Thesis, Department of Chemical Engineering, Lehigh University, Bethlehem PA 18015, U.S.A..

[2.21] (1967) "The SCi Continuous System Simulation Language (CSSL)". Simulation, vol. 9, no. 6 : December 1967; pp. 281 - 303.

[2.22] (1972) "Continuous System Modeling Program III (CSMP-III) - Program Reference Manual". Program number: 5734-XS9, Form: SH19-7001-2. To be ordered from: IBM Canada Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, Canada.

III) USEFULNESS OF COMBINED SYSTEM SIMULATION:

In references [3.5,3.6] it has been shown, that there exist problems which cannot be modeled in a proper way by either purely discrete or purely continuous simulation elements. Examples given in the above references include a steel soaking pit and slabbing mill, and also a chemical batch process. The arguments given in these references to justify the new combined approach to these systems are certainly correct. However, we shall show that the needs for combined simulation languages are even more evident and elementary than explained in these references.

References [3.13,3.14] describe control of the motion of trains by SCR's (silicon controlled rectifiers). For this task, a current has to follow a sine wave within a prespecified tolerance range. This was achieved by controlling the SCR's by a chopper in a way that the simulation always switches back and forth between basically two different models where the condition for switching from one to the other model is expressed by the current reaching its upper or lower limit resp.. When the sine wave crosses through zero, the commutation of the diode rectifier must be taken into account. The full set of equations and state-conditions as used in [3.13,3.14] to model this situation involves a linear, time-variant 5th order system and 12 state-conditions (describing all possible discontinuities). As long as we are only interested in the time-solution for the first half period of the sine wave, we may neglect commutation effects and use a reduced linear time-variant model of 4th order with one state-condition only, describing the conditions of the chopper when to switch from non-conducting to conducting status and vice-versa.

The following set of equations models the system:

$$IL' = (1/X)*(ULD*\sin(\omega t) - AZ*UZ)$$
$$ISP' = (1/XSP)*(UZ - UC)$$
$$UC' = XCSP*ISP$$
$$UZ' = XCS*(AZ*IL - ISP - C1)$$

where: IL = line current
UL = line voltage (= ULD*sin(ωt)
UZ = DC link voltage
ISP = draining circuit current
UC = voltage at draining circuit capacitor

and:
X	=	16.25 mΩ
XSP	=	96.0 mΩ
XCSP	=	384.0 mΩ
XCS	=	318.3 mΩ
ULD	=	1060.66 V
C1	=	6294.2 A

AZ : determines the number of non-conducting choppers which is always either 0.0 or 1.0.

All quantities are related to the secondary winding of the transformer.

Derivatives are taken with respect to ωt, therefore, all X-values have their dimensions specified in Ω (at 16 2/3 Hz). To obtain standardized results which are interpretable also at line frequencies used in other countries ω takes a value of 1.0 (sec) in all computations.

The following circuit diagram illustrates the system.

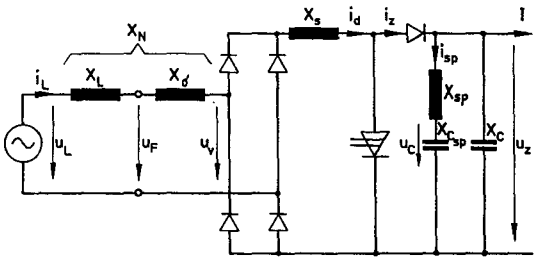

Fig. 3.1: Diagram of the SCR control problem

The line current (IL) is controlled in such a way that it remains always in the neighbourhood of the curve:

$$Y(t) = (15000000/ULD)*\sin(\omega t)$$

For AZ = 0.0 the line current (IL) grows rapidly until it crosses (Y+BT) in the positive direction. At this moment, AZ takes a new value of 1.0, and IL now approaches (Y-BT), where AZ takes a value of 0.0 as before.

$$BT = 200.0 \quad A$$

determines the allowed tolerance bound around Y, within which IL is supposed to operate.

AZ can be modeled in the following way. First we compute an auxiliary variable (H) by the back-lash function depicted in Fig. 3.2.

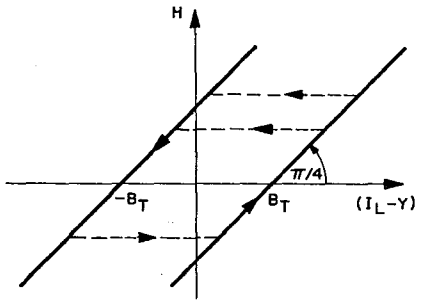

Fig. 3.2: Graph of back-lash function

The auxiliary variable (H) remains always constant as long as this does not force H to leave the area between the two bounds. If the right bound is reached, H moves upward along this border line. If the left bound is reached, H moves downward along this border line. A physical interpretation of this multivalued function could be the loose of a gear. Some simulation languages (CSMP-III, GASP-V) call this a hysteresis function

$$H = \text{hstrs}\ (IL-Y,\ -BT,\ BT,\ H_{initial})$$

AZ can be computed from H with the help of an input-switch function

$$AZ = \text{insw}\ (H,\ x1,\ x2)$$

which is defined as:

$$AZ = \begin{cases} x1\ , & H < 0.0 \\ x2\ , & H \geq 0.0 \end{cases}$$

where x1 = 0.0 and x2 = 1.0.

This control problem may be used as a good bench mark for the ability of an integration algorithm to "digest" nasty discontinuities as illustrated in Fig. 3.7a which depicts some of the state variables of this system plotted versus time. Fig. 3.3 shows the listing of a GASP-V program [3.2,3.3] to simulate the situation outlined above. This program required 75.9 sec of CPU-time for execution on a CDC 6000 series installation. In this example, we used the functions HSTRS and INSW as explained above. Modeling is, thus, done in the same way as if one would use a purely continuous simulation language [3.2]. This is called continuous modeling technique, and the discontinuous functions are called GASP-functions in GASP-V. There exists nevertheless an important difference in the way these functions are handled in GASP-V as compared to a continuous simulation language. GASP-V interprets discontinuities internally as discrete events, and resolves them by iteration. This procedure is further explained in chapter IV of this thesis. Since GASP-V locates discontinuities within GASP-functions through bi-section, this modeling technique does not prove very efficient for solving the stated problem. Fig. 3.4 shows the listing of a modified GASP-V program in which the discontinuities are modeled directly as state-events, and in which an inverse Hermite' interpolation formula (cf. chapter IV) is used to locate the unknown event times. Here the required CPU-time reduces to 10.5 sec. Fig. 3.5 shows the listing of a CSMP-S/360 program [3.16] for the same problem. As one can see, the coded model looks very similar to the GASP-V program of Fig. 3.3. In this case the required CPU-time was 408.0 sec (on the same installation as above) and the obtained results were rubbish (Fig. 3.9a). They were, by the way, produced without any notification of the (credulous) user that they might be incorrect. We will come back to this point later when discussing the aspect of software robustness.

The CSMP program was coded using only standard features offered by the CSMP language, and in all three programs we used the same Runge-Kutta-Simpson integration algorithm of 4th order. This variable step length integration algorithm compares a 4th order Runge-Kutta algorithm to a second order Simpson rule for error estimation. This is the default algorithm offered in CSMP-III (METHOD RKS). A mathematical description of this algorithm can be found in [3.16]. The reduction factor in computing time for GASP-V compared to CSMP was 5.4, when the same continuous modeling technique was used (Fig. 3.3), and 38.9, when the most efficient state-event iteration mechanism available in GASP-V was accessed (Fig. 3.4). The reason for this reduction in CPU-time arises from the fact that CSMP utilizes the step-size control mechanism of the integration algorithm for event location which is a rather inefficient method for location of state-events and even less efficient for the location of time-events. Time-events are scheduled events whose realization time is known in advance. They may, however, be either exogenous time-events, if the event time is known throughout the program, or endogenous time-events, if the event time is computed only during execution of the simulation run as a result of a previous time- or state-event. State-events, on the other hand, are events triggered by the system status fulfilling certain conditions (so called state-conditions) e.g. crossing of a state variable through a prescribed threshold. These events cannot be scheduled in advance, and their unknown realization time must be iterated [3.11].

With respect to the inability of CSMP to solve the posed problem efficiently and correctly, it should be stated that any other continuous simulation language would behave in the same way as CSMP. This is not a shortcoming of the particular language CSMP, but a problem of the inadequate solution technique applied.

```
FACTOR(,2.0,2.0)
* SCR - CONTROL CIRCUIT (CONTINUOUS MODELING TECHNIQUE)
GRAPHY,TIME,IL,UL,UF,UZ
* STEP SIZE (CONTINUOUS MODELING TECHNIQUE)
GRAPHY,TIME,STEP
END


      PROGRAM MAIN (INPUT,OUTPUT,MONITR,TIME,CROSS,SAVE,TAPE1=MONITR,TAP
     1E2=TIME,TAPE3=CROSS,TAPE4=SAVE,TAPE5=INPUT,TAPE6=OUTPUT)
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      NCRDR = 5
      NPRNT = 6
      CALL  GASP
      CALL  BYE
      END


      SUBROUTINE  INTLC
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /UCOM1/ BT,C1,PI,ULD,W,X,XCS,XCSP,XL,XSP,YP
      REAL  IL, ISP
      EQUIVALENCE (SS(1),IL),(SS(2),ISP),(SS(3),UC),(SS(4),UZ)
C
C*****DEFINITION OF MATHEMATICAL NUMBER PI
C
      PI = 4.0*ATAN (1.0)
C
C*****DEFINITION OF SYSTEMS PARAMETERS
C
      UE = 20.0
      ECTR = 5.0
      ECXS = 10.0
      XL1 = 2.0
      ZN2 = 0.075
      VE = 1.0
      US = 1.0
      BT = 200.0
      W = 1.0
      UZ0 = 1200.0
C
C*****DEFINITION OF AUXILIARY VARIABLES
C
      WF = 100.0*PI/3.0
      XL = XL1/(UE*UE)
      XSI = ZN2*ECTR/100.0
      XN = XL + XSI
      XS = ZN2*ECXS/100.0
      X = XN + XS
      XCSP = 1.0/(0.024868*WF)
      XSP = XCSP/(4.0*VE)
      XCS = 1.0/(0.03*WF)
      ULD = 750.0*US*SQRT (2.0)
      YP = 7.5E6*2.0/ULD
      C1 = (YP + BT/2.0)*ULD/(2.0*UZ0)
C
C*****DEFINITION OF INITIAL CONDITIONS
C
      IL = 0.0
      ISP = -C1
      UC = 1400.0
      UZ = UZ0
      RETURN
      END
```



```
      SUBROUTINE  STATE
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /OCOM1/ STEP,T,UFS,ULS,UZS
      COMMON /UCOM1/ BT,C1,PI,ULD,W,X,XCS,XCSP,XL,XSP,YP
      REAL  IL, ISP, ILDOT, ISPDOT
      EQUIVALENCE (SS(1),IL),(SS(2),ISP),(SS(3),UC),(SS(4),UZ),
     1(DD(1),ILDOT),(DD(2),ISPDOT),(DD(3),UCDOT),(DD(4),UZDOT)
C
C*****COMPUTATION OF LINE VOLTAGE
C
      H1 = SIN (W*TNOW)
      UL = ULD*H1
C
C*****COMPUTATION OF NUMBER OF NON-CONDUCTING CHOPPERS (AZ)
C*****BY CONTINUOUS MODELING TECHNIQUES
C
      Y = YP*H1
      SS(5) = IL - Y
      SS(6) = HSTRS (5, -BT/2.0, -BT, BT, 1.0E-3, 1)
      SS(7) = 0.0
      SS(8) = 1.0
      AZ = GINSW (6, 7, 8, 1.0E-3, 2)
C
C*****STATE SPACE DESCRIPTION OF THE SYSTEM
C
      ILDOT = (1.0/X)*(UL - AZ*UZ)
      ISPDOT = (1.0/XSP)*(UZ - UC)
      UCDOT = XCSP*ISP
      UZDOT = XCS*(AZ*IL - ISP - C1)
C
C*****STORE DATA FOR OUTPUT
C
      T = 30.0*TNOW/PI
      ULS = 10.0*UL
      UF = UL - ILDOT*XL
      UFS = 10.0*UF
      UZS = 10.0*UZ
      STEP = DTFUL
      RETURN
      END


      SUBROUTINE  INTEG
C
C*****SELECT INTEGRATION METHOD
C
      CALL  RKS
      RETURN
      END
```

```
                        SIMULATION PROJECT NUMBER   1  BY       GELLIER

                        DATE 11/ 16/ 1978       RUN NUMBER    1 OF    1
                        LLSUP=0000000000000000  GASP V  VERSION 25MAY78

NCRDR=    5    NPRNT=    6    NMONT=    1    IICRS=    3    IISAV=    4    IITIM=    2
NMOTP=    5    NMXKS=    1

NNAMS=STEP          TIME            UF              UL              UZ              IL

NNCLT=    0    NNSTA=    0    NNHIS=    0    NNPRM=    0    NMPLT=    0    NNSTR=    1    NNTRY=    0
NNATR=    0    NNFIL=    0    NNSET=    0    NNEQD=    4    NNEQS=    4    NFLAG=    0    NMPDE=   -0

IIEVT=    1    LLERR=    0    AAERR=      .1000E-02          RRERR=      .1000E-02
DTMIN=     .1000E-06          DTMAX=      .6000E-02          DTSAV=      .6000E-02

NSTOP=    1    JJCLR=    0    JJOEG=    1    IICRO=    0    TTBEG=    0.          TTFIN=      .3600E+01
JJFIL=    0
IISED=   -0
```

Fig. 3.3: Listing of a GASP-V program for the SCR control problem involving GASP-functions (continuous modeling technique)

```
FACTOR(,2.0,2.0)
* SCR - CONTROL CIRCUIT (COMBINED MODELING TECHNIQUE)
GRAPHY,TIME,IL,UL,UF,UZ
* STEP SIZE (COMBINED MODELING TECHNIQUE)
GRAPHY,TIME,STEP
END


      PROGRAM MAIN (INPUT,OUTPUT,MONITR,TIME,GROSS,SAVE,TAPE1=MONITR,TAP
     1E2=TIME,TAPE3=GROSS,TAPE4=SAVE,TAPE5=INPUT,TAPE6=OUTPUT)
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      NCRDR = 5
      NPRNT = 6
      CALL  GASP
      CALL  BYE
      END


      SUBROUTINE  INTLC
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /UCOM1/ BT,C1,PI,ULD,W,X,XCS,XCSP,XL,XSP,YP
      COMMON /UCOM2/ AZ
      REAL  IL, ISP
      EQUIVALENCE (SS(1),IL),(SS(2),ISP),(SS(3),UC),(SS(4),UZ)
C
C*****DEFINITION OF MATHEMATICAL NUMBER PI
C
      PI = 4.0*ATAN (1.0)
C
C*****DEFINITION OF SYSTEMS PARAMETERS
C
      UE = 20.0
      ECTR = 5.0
      ECXS = 10.0
      XL1 = 2.0
      ZN2 = 0.075
      VE = 1.0
      US = 1.0
      BT = 200.0
      W = 1.0
      UZ0 = 1200.0
C
C*****DEFINITION OF AUXILIARY VARIABLES
C
      WF = 100.0*PI/3.0
      XL = XL1/(UE*UE)
      XSI = ZN2*ECTR/100.0
      XN = XL + XSI
      XS = ZN2*ECXS/100.0
      X = XN + XS
      XCSP = 1.0/(0.024868*WF)
      XSP = XCSP/(4.0*VE)
      XCS = 1.0/(0.03*WF)
      ULO = 750.0*US*SQRT (2.0)
      YP = 7.5E6*2.0/ULO
      C1 = (YP + BT/2.0)*ULO/(2.0*UZ0)
C
C*****DEFINITION OF INITIAL CONDITIONS
C
      IL = 0.0
      ISP = -C1
      UC = 1400.0
      UZ = UZ0
      SS(5) = IL
C
C*****SELECT INITIAL MODEL
C
      AZ = 0.0
      RETURN
      END


      SUBROUTINE  EVNTS(IX)
      COMMON /UCOM2/ AZ
C
C*****EVENT HANDLING: THE ONLY REQUIRED ACTIVITY IS TO SET AZ TO 1.0
C*****WHEN IT IS 0.0 AND VICE-VERSA
C
      AZ = 1.0 - AZ
      CALL  EFCRE
      RETURN
      END


      SUBROUTINE  INTEG
C
C*****SELECT INTEGRATION METHOD
C
      CALL  RKS
      RETURN
      END
```



```
      SUBROUTINE  STATE
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /UCOM1/ BT,C1,PI,ULD,W,X,XCS,XCSP,XL,XSP,YP
      COMMON /UCOM2/ AZ
      REAL  IL, ISP, ILDOT, ISPDOT
      EQUIVALENCE (SS(1),IL),(SS(2),ISP),(SS(3),UC),(SS(4),UZ),
     1(DD(1),ILDOT),(DD(2),ISPDOT),(DD(3),UCDOT),(DD(4),UZDOT)
C
C*****COMPUTATION OF LINE VOLTAGE
C
      UL = ULD*SIN (W*TNOW)
C
C*****STATE SPACE DESCRIPTION OF THE SYSTEM
C
      ILDOT = (1.0/X)*(UL - AZ*UZ)
      ISPDOT = (1.0/XSP)*(UZ - UC)
      UCDOT = XCS*ISP
      UZDOT = XCS*(AZ*IL - ISP - C1)
C
C*****ADD ADDITIONAL STATE EQUATION TO ENABLE FOR
C*****INVERSE HERMITE' INTERPOLATION
C
      DD(5) = ILDOT - YP*COS (W*TNOW)
C
C*****STORE DATA FOR OUTPUT
C
      T = 30.0*TNOW/PI
      ULS = 10.0*UL
      UF = UL - ILDOT*XL
      UFS = 10.0*UF
      UZS = 10.0*UZ
      STEP = DTFUL
      RETURN
      END


      SUBROUTINE  SCOND
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /GCOM14/ AAZ(10),IIZ(10),JJZ(10),KKZ(10),NNZ,ZZ(10)
      COMMON /UCOM1/ BT,C1,PI,ULD,W,X,XCS,XCSP,XL,XSP,YP
      COMMON /UCOM2/ AZ
C
C*****AZ MAY BE USED TO DETERMINE THE ACTIVE MODEL
C
      IF (AZ.EQ.1.0)  GO TO 1
C
C*****AZ = 0.0 : CHOPPER CONDUCTING
C
      ZZ(1) = SS(5) - BT
      IIZ(1) = 5
      JJZ(1) = 1
      RETURN
C
C*****AZ = 1.0 : CHOPPER NON-CONDUCTING
C
    1 ZZ(1) = SS(5) + BT
      IIZ(1) = 5
      JJZ(1) = -1
      RETURN
      END
```

```
                    SIMULATION PROJECT NUMBER   2  BY       CELLIER

                    DATE 11/ 16/ 1978          RUN NUMBER    1 OF    1
                    LLSUP=0000000000000000   GASP V  VERSION 25MAY78

NCRDR=     5    NPRNT=     6    NMONT=     1    IICRS=     3    IISAV=     4    IITIM=     2
NNOTP=     5    NNMXS=     1

NNAMS=STEP           TIME             UF               UL               UZ               IL

NNCLT=     0    NNSTA=     0    NNMIS=     0    NNPRM=     0    NNFLT=     0    NNSTR=     1    NNTRY=     0
NNATR=     0    NNFIL=     0    NNSET=     0    NNEQD=     5    NNEQS=     0    NFLAG=     0    NNPDE=    -0

IIEVT=     1    LLERR=     0    AAERR=      .1000E-02           RRERR=      .1000E-02
DTMIN=      .1000E-06           DTMAX=      .6000E-02           DTSAV=      .6000E-02

NNZ  =     1    IIDIS=     0    EEPS =      .1000E-13           NNINT=  5000    NNITR=   100
AAZ( 1)=   .1000E-13    AAZ( 2)=          -R    AAZ( 3)=          -R    AAZ( 4)=          -R
AAZ( 5)=          -R    AAZ( 6)=          -R    AAZ( 7)=          -R    AAZ( 8)=          -R
AAZ( 9)=          -R    AAZ(10)=          -R

MSTOP=     1    JJCLR=     0    JJBEG=     1    IICRD=     0    TTBEG=     0.          TTFIN=      .3000E+01
JJFIL=     0
IISEO=    -0
```

Fig. 3.4: Listing of a GASP-V program for the SCR control problem involving state-events to be iterated by inverse Hermite' interpolation (combined modeling technique)

```
TITLE  SCR - CONTROL CIRCUIT (CSMP-S/360)
FIXED  K, IC, IN, IP
/      DIMENSION  TAB(1000,6), X1(1002), Y1(1002), IC(6), IN(6)
/      DATA  (IN(I),I=1,6)/2HIL,2HUL,2HUF,2HUZ,8HTIMEIMS),4HSTEP/
CONSTANT W = 1.0
PARAMETER UE = 20.0, ECTR = 5.0, ECXS = 10.0, XL1 = 2.0, ...
          ZN2 = 0.075, VE = 1.0, US = 1.0, BT = 200.0
INCON IL0 = 0.0, UC0 = 1400.0, UZ0 = 1200.0
*
*
INITIAL
*******
*******
*
*****INITIALIZATION OF PLOTTER
*
NOSORT
      CALL  PLTFIN
SORT
*
******DEFINITION OF MATHEMATICAL NUMBER PI
*
PI = 4.0*ATAN (1.0)
*
******DEFINITION OF AUXILIARY VARIABLES
*
WF = 100.0*PI/3.0
XL = XL1/(UE*UE)
XSI = ZN2*ECTR/100.0
XN = XL + XSI
XS = ZN2*ECXS/100.0
X = XN + XS
XCSP = 1.0/(0.024868*WF)
XSP = XCSP/(4.0*VE)
XCS = 1.0/(0.03*WF)
ULD = 750.0*US*SQRT (2.0)
YP = 7.5E6*2.0/ULD
C1 = (YP + BT/2.0)*ULD/(2.0*UZ0)
*
******DEFINITION OF COMPUTED INITIAL CONDITION FOR ISP
*
ISP0 = -C1
*
******DEFINITION OF AUXILIARY COUNTER K
******AS WELL AS OF THE COMMUNICATION INTERVAL (TPLOT)
******AND OF A FLAG FOR DYNAMIC RUN TERMINATION
*
K = 0
FLAG = 0.0
TPLOT = 0.006
*
*
DYNAMIC
*******
*******
*
******COMPUTATION OF THE LINE VOLTAGE
*
H1 = SIN (W*TIME)
UL = ULD*H1
*
******COMPUTATION OF THE NUMBER OF NON-CONDUCTING CHOPPERS (AZ)
*
Y = YP*H1
SS5 = IL - Y
SS6 = HSTRSS (-BT/2.0, -BT, BT, SS5)
AZ = INSW (SS6, 0.0, 1.0)
*
******STATE SPACE DESCRIPTION OF THE SYSTEM
*
ILDOT = (1.0/X)*(UL - AZ*UZ)
IL = INTGRL (IL0, ILDOT)
ISP = INTGRL (ISP0, (1.0/XSP)*(UZ - UC))
UC = INTGRL (UC0, XCSP*ISP)
UZ = INTGRL (UZ0, XCS*(AZ*IL - ISP - C1))
```



```
NOSORT
*
******STORE DATA FOR LATER PLOT
*
      SW = IMPULS (0.0, TPLOT)
      IF ((SW.NE.1.0).OR.(KEEP.NE.1))  GO TO 1
      IF (K.GE.1000)  FLAG = 1.0
      K = K + 1
      TAB(K,1) = IL
      TAB(K,2) = 10.0*UL
      UF = UL - ILDOT*XL
      TAB(K,3) = 10.0*UF
      TAB(K,4) = 10.0*UZ
      TAB(K,5) = 30.0*TIME/PI
      TAB(K,6) = DELT
    1 CONTINUE
*
*
TERMINAL
********
********
*
NOSORT
*
******REDUCE TABLE IF NOT FULL
*
      IF (K.LT.1000)  CALL  RED (TAB, 1000, 6, K)
*
******FIRST PLOT
*
      DO 1000  I=1,4
      IC(I) = -1
 1000 CONTINUE
      IC(5) = 1
      IC(6) = 0
      IP = 0
      CALL  FIGUR (TAB, K, 6, IC, IN, 1, 1, IP, X1, Y1)
*
******SECOND PLOT
*
      DO 1001  I=1,4
      IC(I) = 0
 1001 CONTINUE
      IC(6) = -1
      CALL  FIGUR (TAB, K, 6, IC, IN, 1, 1, IP, X1, Y1)
*
******CLOSE PLOT FILE PROPERLY
*
      CALL  PLOT (0.0, 0.0, 999)
*
*
TIMER  FINTIM = 3.0, PRDEL = 0.006
FINISH  FLAG = 0.5
PRINT  IL, ISP, UC, UZ
END
STOP
ENDJOB
```

Fig. 3.5: Listing of a CSMP-S/360 program for the SCR control problem

Fig. 3.6 depicts schematically what happens to the integration step-size at event times.

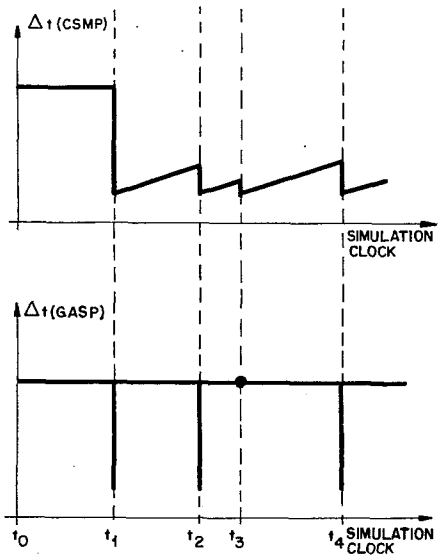

Fig. 3.6: Integration step-sizes of continuous and combined languages versus the simulation clock.

Using a continuous simulation language, the step-size will be considerably reduced when a discontinuity is encountered since the integration algorithm is unable to compute a step properly over discontinuities. This fact has been used in these languages to "localize" event times. However, since the program cannot know that a discontinuity is occuring (no language element is provided to explain this to the system), the algorithm will "think" that the set of equations suddenly became extremely stiff, and reduce the step-size to cope with the new situation. (A discontinuity can be considered as a local point of infinite stiffness.) For this reason

originators of continuous simulation packages always claim that their software is able to handle discontinuities.

Having located the event, the algorithm will carefully explore the possibility to make the step-size larger again, but is not allowed to enlarge the step-size immediately since this would lead to instability behaviour in an actual stiff case.

A combined simulation language, on the other hand, will provide for a language element to describe discontinuities. Now the step-size will be reduced by an event iteration procedure inherent in the program to locate the unknown event time in case of state-events (events 1, 2 and 4 in Fig. 3.6) taking place, whereas the step-size will simply be reset to the known event time in case of a time-event (event 3 in Fig. 3.6) taking place. After accomplishment of the event the integration algorithm will be restarted and will use the internally provided (and hopefully efficient) algorithm to obtain a good guess for the new "first" step-size to be used. Note that the independent axis on the graph denotes the simulation clock and not CPU-time.

This technique has first been described in [3.12]. In this reference, it has been applied to a very specific application problem. [3.7,3.8] describe this technique from a somewhat more general point of view. A detailed discussion of this technique shall be presented in chapter IV of this thesis.

Fig. 3.7a depicts the behaviour of some variables of the above outlined control problem versus simulation clock, and Fig. 3.7b shows the integration step-size (dt) versus simulation clock for the first GASP-V program of Fig. 3.3. Fig. 3.8a and Fig. 3.8b show the same quantities output by the second GASP-V program of Fig. 3.4, and Fig. 3.9a and Fig. 3.9b resp. are produced by the CSMP program of Fig. 3.5.

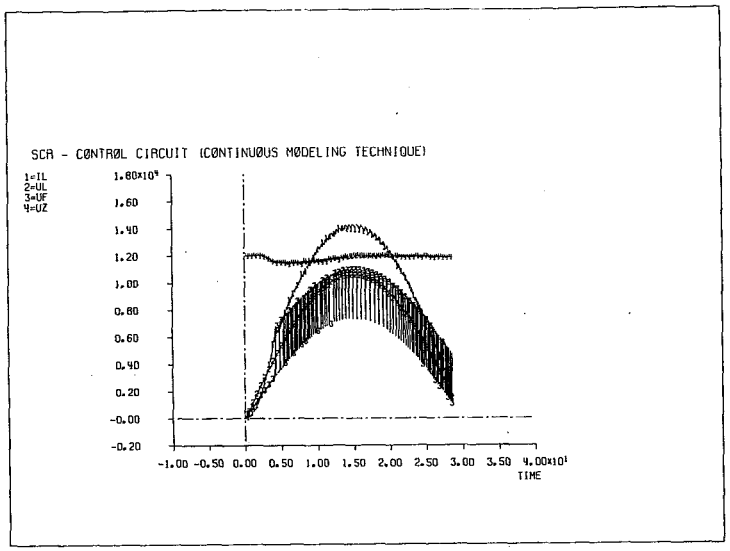

Fig. 3.7a: Time response of the GASP-V program of Fig. 3.3.

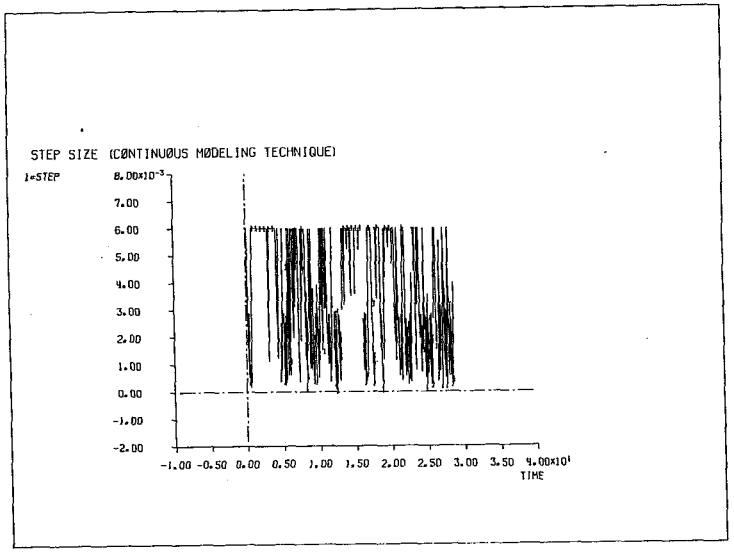

Fig. 3.7b: Step-size versus simulation clock
for the GASP-V program of Fig. 3.3.

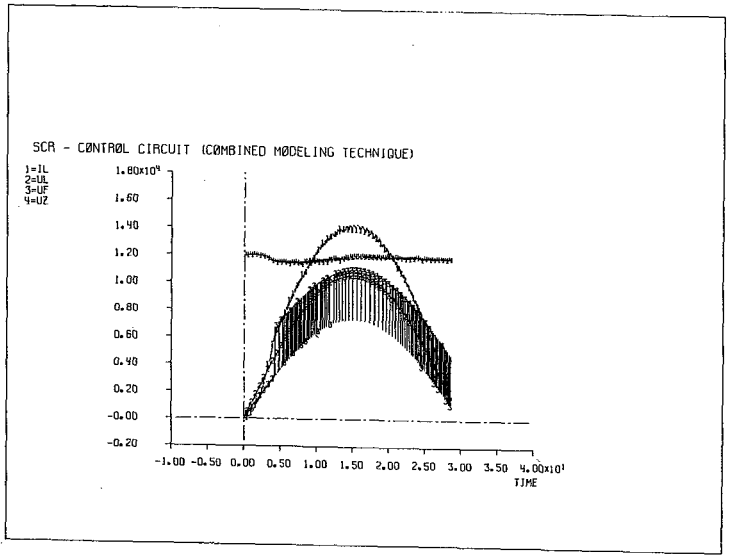

Fig. 3.8a: Time response of the GASP-V program of Fig. 3.4.

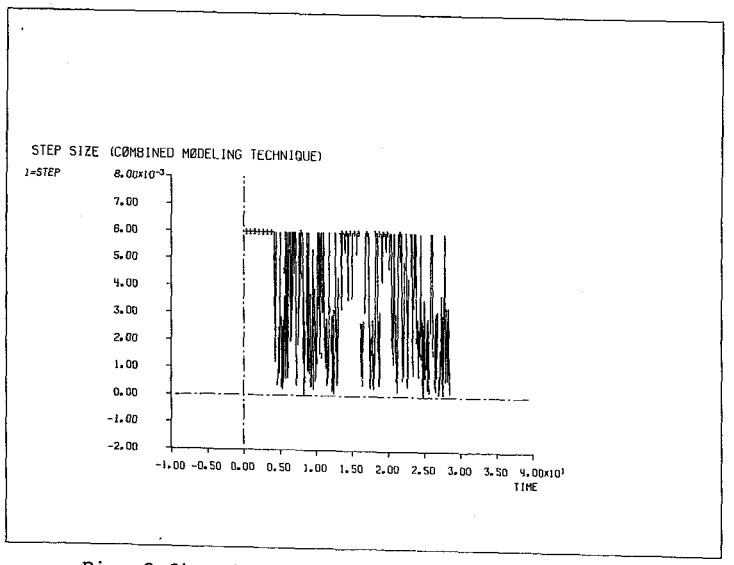

Fig. 3.8b: Step-size versus simulation clock
for the GASP-V program of Fig. 3.4.

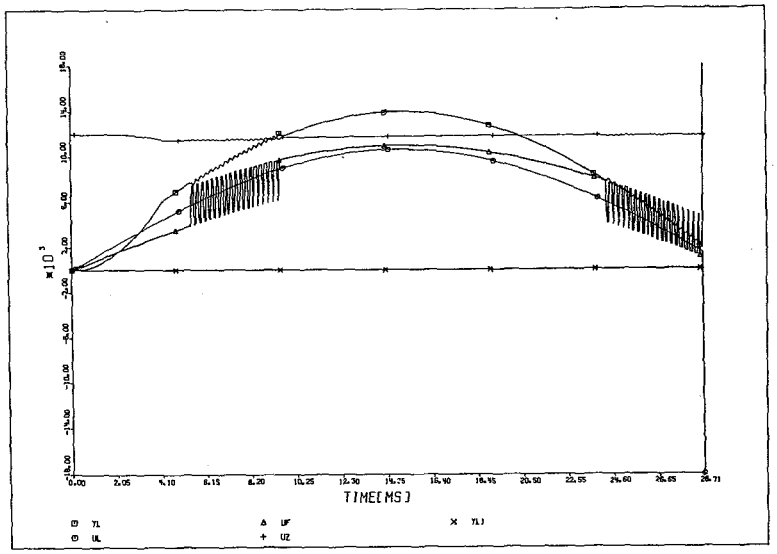

Fig. 3.9a: Time response of the CSMP-S/360 program of Fig. 3.5.

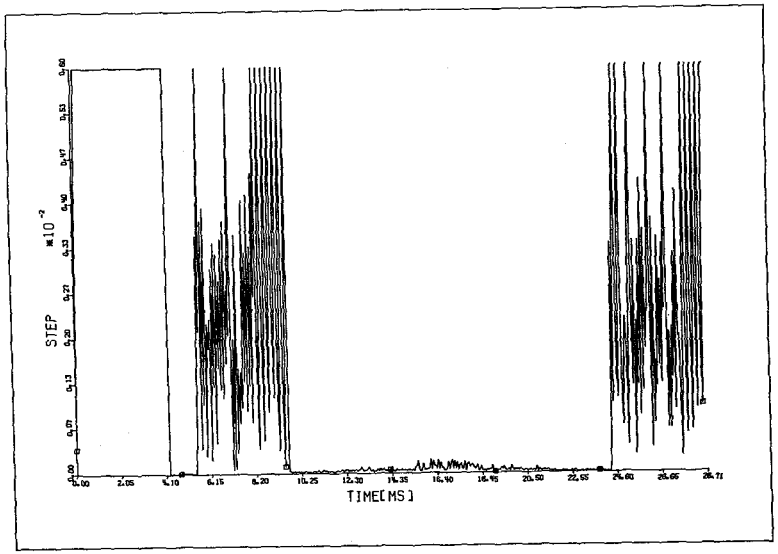

Fig. 3.9b: Step-size versus simulation clock for the CSMP-S/360 program of Fig. 3.5.

The incorrect results obtained using the CSMP program arise from the following:

Let us assume that at time (t*) the auxiliary variable (H) has a negative value and is approaching zero. The crossing will usually take place at any intermediate computation of the state derivatives within an integration step. AZ will now immediately change its value from 0.0 to 1.0. Since the integration algorithm cannot compute values over discontinuities, its inherent error estimation procedure will detect that something went wrong, and will reject the current integration step. IL and Y will resume their values at the beginning of the integration step and the step is repeated with a smaller dt.

During execution of one integration step, it is even possible that AZ changes its value several times since the first time derivative of the line current (IL') will immediately change its sign as soon as AZ switches from 0.0 to 1.0 or vice-versa. Since the memory of the HSTRSS-function may only be updated at the end of successfully accomplished integration steps (KEEP = 1), the output of the HSTRSS-function (H) will also cross back and forth through zero, and thus AZ will also change its value from 0.0 to 1.0 repeatedly. The step-size (DELT) will be reduced until the error estimation algorithm is satisfied. This will be the case as soon as the two integration algorithms, which are compared with each other to obtain an estimate for the local integration error (RK4 and Simpson in this example), show similar results. That is, either when the analysis is correct, or when both algorithms agree to the same incorrect result (!). Since the analysis can never be correct over this discontinuity, the step-size reduction will go on until both algorithms, by chance, produce a similar (incorrect) result. (For sufficiently small step-sizes any integration algorithm will behave like Euler and, thus, produce comparable results.)

The future trajectory will now depend heavily upon whether AZ takes (by chance) a value of 0.0 or 1.0 at the end of the finally accepted (but nevertheless incorrect) integration step. If the threshold has been passed, the HSTRSS-function can update its memory correctly, and integration proceeds as it should. If the threshold has not been passed, the next integration step will face the same problem as the one before, namely, that the trajectory will again try to surmount the threshold without success, and the program starts creeping. This is a well known phenomenon when using continuous simulation languages to simulate discontinuous models.

Fig. 3.9a shows that, during computation, AZ sometimes takes the correct value and sometimes the wrong value at the end of an integration step. Initially there exists a time interval in which the program gives incorrect results. This is followed by an interval during which the program proceeds correctly and produces valid results. Following this interval, one can again observe a long time interval during which the program, using an incredibly small step-size (Fig. 3.9b), creeps along producing erroneous results. Then another interval follows during which the correct time response is produced.

As shown in [3.4] the only proper way to surmount this difficulty is to inhibit discontinuities to take place within integration steps.

Discontinuous functions are composed of continuous branches. Instead of allowing a discontinuity to take place within an integration step, we only describe the conditions when to switch from one continuous branch to another (state conditions), but remain on the first (prolongated) continuous branch throughout the whole iteration procedure. At event time only, after the iteration procedure has successfully converged, we will switch from one branch to the other.

After event handling, before we continue to integrate the system further over time, we will then reinitialize the integration algorithm. With this technique we can guarantee purely continuous trajectories throughout integration.

In using GASP-functions (continuous modeling technique) this splitting-up procedure is hidden to the user, but nevertheless takes place precisely as described above resulting in series of hidden state- and time-events which are taking place and are properly handled by the system without demanding any further consideration by the user of the software [3.3].

Considering the numerical aspects of the problem one should describe the above mentioned control system by combined simulation techniques rather than by purely continuous simulation. However, this problem would definitely belong to the class of continuous systems according to the "common" use of the term. This gives the required motivation for our redefinition of the terms combined and continuous simulation as stated in the introduction.

Considering the aspect of information processing CSMP obviously offers modeling elements (such as a switch function) which it is unable to preprocess into properly executable code. Again this is not a problem of the language CSMP, but holds for all CSSL-type languages. Some of the languages (like DARE-P [3.9]) are somewhat more modest in the facilities they offer in this respect, for which they are criticized by many users. We, however, believe that it is more desirable to offer few facilities than to write a beautiful manual, offering many nice features, which effectively cannot be properly used. On the other hand, the reaction of these users proves that the facilities are useful and needed. For this reason, we feel that in a future revision of the CSSL specifications [3.15], combined simulation facilities should be taken into account, opposing herein to

the opinion expressed in [3.1]. A revision of the CSSL specifications will be necessary anyway, if for no other reason than the original definition contains over 40 syntactical errors as shown in [3.10].

As can be seen from the above discussion: The problem of combined simulation can be subdivided into the numerical aspects (executability of the run-time system) and the aspects of information processing (definition of the descriptive input language). These two problems shall subsequently be considered in greater detail.

References:

[3.1] P.R.Benyon: (1976) "Improving and Standardizing Continuous Simulation Languages". Proc. of the SIMSIG Simulation Conference, Melbourne, Australia, May 17-19, 1976; pp. 130 - 140.

[3.2] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[3.3] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 391 - 402.

[3.4] F.E.Cellier, Rufer D.F.: (1975) "Algorithm Suited for the Solution of Initial Value Problems in Engineering Applications". Proc. of the SIMULATION'75 Symposium, Zurich, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 160 - 165.

[3.5] D.A.Fahrland: (1970) "Combined Discrete-Event Continuous System Simulation". Simulation vol. 14 no. 2 : February 1970; pp. 61 - 72.

[3.6] D.G.Golden, Schoeffler J.D.: (1973) "GSL - A Combined Continuous and Discrete Simulation Language". Simulation vol. 20 no. 1 : January 1973; pp. 1 - 8.

[3.7] J.L.Hay, Crosbie R.E., Chaplin R.I.: (1974) "Integration Routines for Systems with Discontinuities". Computer Journal, Vol. 17, No. 3; pp. 275 - 278.

[3.8] J.L.Hay, Griffin A.W.J.: (1979) "Simulation of Discontinuous Dynamical Systems". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company (Editors: L.Dekker, G.Savastano, G.C.Vansteenkiste); pp. 79 - 87.

[3.9] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice Hall.

[3.10] T.I.Oren: (1975) "Syntactic Errors of the Original Formal Definition of CSSL 1967". Technical Report TR75-01 (IEEE Computer Society Repository no. R75-78), Computer Science Department, University of Ottawa, Ottawa, Canada.

[3.11] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language. John Wiley.

[3.12] B.Ramer, Ramer U.: (1969) "Digitale Untersuchung einer halbgesteuerten Dreiphasenbrueckenschaltung". Senior Project. Internal Report: AIE-69-8041. Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[3.13] H.Schlunegger: (1977) "Untersuchung eines netzrueckwirkungsarmen, zwangskommutierten Triebfahrzeug-Stromrichters zur Einspeisung eines Gleichstromzwischenkreises aus dem Einphasennetz". Ph.D. Thesis, no. DISS.ETH.5867: The Swiss Federal Institute of Technology Zurich, Switzerland.

[3.14] H.Schlunegger: (1977) "Digital Simulation of a Forced-Commutated Converter for Single-Phase for AC Locomotives". Proc. of the IFAC - Symposium on Control in Power Electronics and Electrical Drives, Duesseldorf, FRG. Published by Pergamon Press (Editor: M.A.Kaaz); pp. 759 - 767.

[3.15] (1967) "The SCi Continuous System Simulation Language (CSSL)". Simulation, vol. 9 no. 6 : December 1967; pp. 281 - 303.

[3.16] (1972) "Continuous System Modeling Program III (CSMP-III) - Program Reference Manual". Program number: 5734-XS9, Form: SH19-7001-2. To be ordered from: IBM Canada Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, Canada.

IV) NUMERICAL ASPECTS:

IV.1) Structure of the Run-Time Package:

The previous example shows that for the execution of combined system simulation, the following concept is to be used. A combined model must be subdivided into the following parts:

a) a discrete part consisting of elements known from discrete event simulation

b) a continuous part consisting of elements known from continuous system simulation

c) an interface part describing the conditions when to switch from (a) to (b) and vice-versa

During the execution of a combined system simulation we are, therefore, performing either entirely discrete event simulation (with its well known properties) or entirely continuous system simulation (with its also well known properties), whereas execution of simultaneously combined continuous and discrete simulation does not exist. Thus a combined simulation run-time package must be composed of:

a) a discrete event simulation run-time package

b) a continuous system simulation run-time package

c) some algorithms describing the activities to be taken, when branching from (a) to (b) or vice-versa is required.

The numerical requirements for the subsystems (a) and (b) are both well known and discussed on many occasions

[4.1,4.3,4.6,4.8] and, thus, need not be considered here again. Once this structure has been understood, we can restrict ourselves merely to combining previously developed software for discrete and continuous simulation to obtain a good run-time package for combined simulation as well.

In the following, we will restrict our view on subsystem (c).

IV.1.1) Conditions for Changing to Continuous Simulation when Executing Discrete Simulation:

Let the simulation clock be advanced to event time tl. Executing discrete simulation means that the system is about to perform event handling at time tl. We have to execute discrete simulation until all events scheduled for time tl have been performed. We have then to switch to continuous simulation if there are differential equations currently involved in the combined simulation (for some intervals of time there may be none). Otherwise we advance the simulation clock to the next event time, and continue executing discrete simulation until there are no events left to be performed for this new event time. Therefore, no special algorithms need to be developed for this case. After event handling has been performed, the integration algorithm needs to be restarted. This is especially important in case multistep methods are being used.

IV.1.2) Conditions for Changing to Discrete Simulation when Executing Continuous Simulation:

Continuous simulation has to be performed either up to the next scheduled event time (for time-events), or until a state-condition is met which triggers execution of a state-event, whichever comes first. In both cases, the step-size

control mechanism of the integration algorithm has to be disabled. In the former case (handling of a time-event), the step-size simply has to be reduced to the scheduled event time, in the latter case (handling of a state-event) a new step-size control algorithm must be activated for iteration of the solution to the unknown event-time. Again, these algorithms are not really new. Any good iteration procedure (like Newton-Raphson) can solve the problem [4.2,4.5,4.7]. On one hand the iteration procedure should converge as fast as possible to minimize computing time. This calls for an iteration procedure with a quadratic or cubic convergence ratio. On the other hand, our requirement of software robustness (as discussed later) call for an iteration procedure with an infinite convergence range. For this reason, we recommend a combination of the inverse Hermite' interpolation (fast convergence) with Regula-Falsi (unlimited convergence range). This scheme is discussed in the following.

IV.1.2.1) The Regula-Falsi Iteration Scheme:

Fig. 4.1 shows how the Regula-Falsi algorithm may be used to detect the crossing of a state-condition (any function of state and time) through zero.

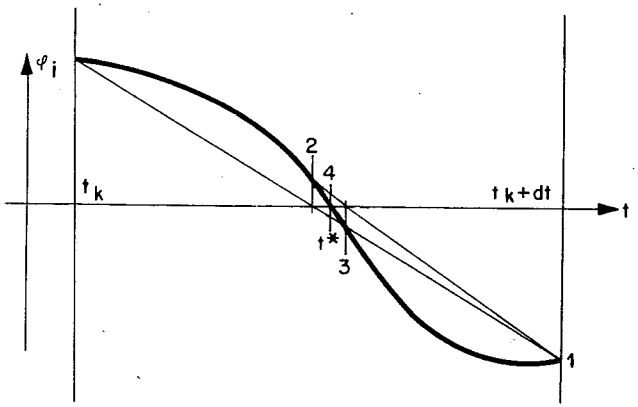

Fig. 4.1: Iteration of a zero crossing with Regula-Falsi

We can formulate all types of state-conditions as zero crossings without restriction in generality, since any state-condition can be brought into this form by simple subtraction of the two sides. In this special form the state-condition will be called a discontinuity function (d.f.) [4.2].

Integration goes on until a d.f. changes its sign for the first time. At this moment, we know that a state-event is supposed to take place within the last executed integration step. The step-size control mechanism of the integration algorithm is now disabled and the step is repeated with a new step-size computed by the iteration scheme. The interval is reduced by always keeping the d.f. at one interval bound above and, at the other, below the zero level. The advantage of this iteration scheme is that it will converge under all circumstances (unlimited convergence range). The disadvantage is that it only provides a linear convergence speed.

A bi-section law (repeated halving of the step size until

the zero crossing is located) has been used in [4.10] to iterate state-conditions, since Pritsker felt that the demands of software robustness would be jeopardized by any other means of iteration (A.A.B.Pritsker: Private communication). When using the Regula-Falsi scheme, we, indeed, face problems which do not exist in bi-section. These are illustrated in Fig. 4.2a and Fig. 4.2b resp..

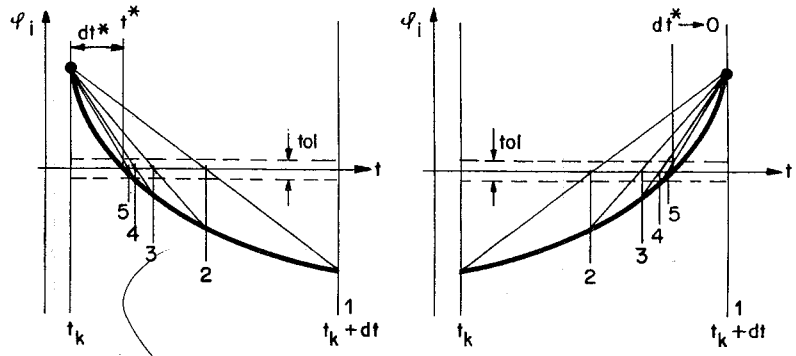

Fig. 4.2: Difficulties of iterating zero crossings with Regula-Falsi

a) As can be seen in Fig. 4.2a, the left interval bound remains here always the same. The step-size will, therefore, not necessarily converge to zero in this algorithm. As a matter of fact it is here the difference between step-sizes, following each other during iteration, which converges to zero. Bi-section on the other hand halves the step-size during each iteration step. This must, therefore, converge to zero. In using an algorithm like Regula-falsi special care is, thus, required to terminate the iteration correctly.

b) In Fig. 4.2b the right interval bound remains always unchanged. At this interval bound the allowed tolerance

range of the d.f. is not met. The left interval bound approaches the zero crossing instant rapidly, but the d.f. can never cross through zero there. When we continue to integrate the system over time after event handling has been accomplished, it may well happen that the previously iterated d.f. is again "discovered" during the first new integration step, and the algorithm finds itself iterating the same zero crossing again and again. Unfortunately, the package has no means to discover what happened since, at event time, the whole system structure may change. For this reason, the iteration algorithm cannot decide whether this is really "the same" d.f. as before or not. The meaning of the i-th d.f., which led to the iteration, may (but need not) have changed completely after event handling. The package must, therefore, ask the user to supply information as to whether a possible zero crossing of the i-th d.f., within the first integration step after event handling, is to be regarded or ignored.

IV.1.2.2) The Generalized Regula-Falsi Iteration Scheme:

So far we have discussed how zero crossings of a d.f. can be iterated. The first zero crossing of any d.f. will be found by these means. It may, however, happen that several d.f.'s cross through zero within the same integration step. The iteration scheme must, therefore, be generalized in such a way that the first zero crossing can be found. (There is, however, no a priori information available on which of the candidates is crossing first.) The iteration scheme applied is very simple. We compute a "new" step-size for all crossing d.f.'s independently of each other, and use their arithmetic mean value as the next step-size. Most probably, some of the d.f.'s still have a crossing within the reduced interval whereas others have not. We continue with this algorithm until just one crossing is left within the considered interval. From then on, we may proceed as in section IV.1.2.1.

Problems may arise if the user (unconsciously) has specified the same d.f. twice. In this case, the generalized scheme has to be used until "both" d.f.'s lie within the specified tolerance range. The user is, therefore, required to test at event time for possible realizations of all d.f.'s sequentially.

A further rule which must be remembered when formulating d.f.'s is that no d.f. may be equal to zero over a finite time interval. This is obvious from the definition of the d.f., since zero crossings determine event times.

Fig. 4.3 depicts the generalized Regula-Falsi iteration scheme.

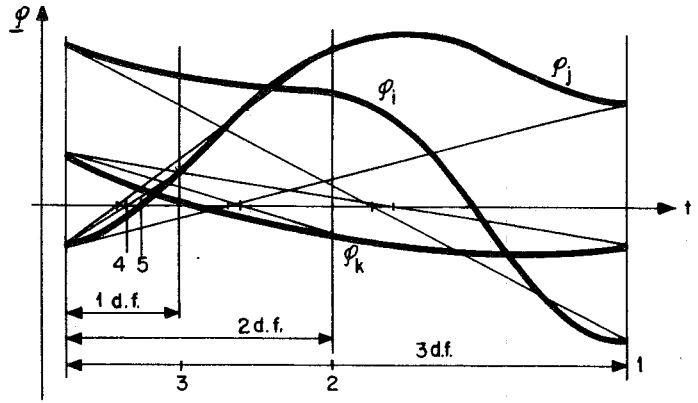

Fig. 4.3: Iteration of zero crossings using generalized Regula-Falsi

Let the end points of all d.f.'s be stored in two arrays ZL(i) for the left (last) end point and ZN(i) for the right

(new) end point. The array KKZ(i) is supposed to contain information whether a crossing of the i-th d.f. took place within the current integration step (KKZ(i) ≠ 0) or not (KKZ(i) = 0). NNZ denotes the number of active d.f.'s. The next step-size (DTNEX) can be computed by the algorithm coded in Fig. 4.4.

```
      S = 0.0
      IA = 0
      DO 1000  I=1,NNZ
      IF (KKZ(I))  1, 2, 1
    1 HA = ABS (ZL(I))
      HB = ABS (ZN(I))
      S = S + HA/(HA + HB)
      IA = IA + 1
    2 CONTINUE
 1000 CONTINUE
      DTNEX = S*DTFUL/FLOAT (IA)
```

Fig. 4.4: Program segment for the generalized Regula-Falsi rule

If only one crossing takes place, this algorithm degenerates to the normal Regula-Falsi rule.

IV.1.2.3) The Inverse Hermite' Interpolation:

This iteration scheme generates a cubic polynomial through the two boundary values of a d.f. and their first time derivatives. For the generation of the polynomial, the two axes (d.f. and time) are exchanged. This has two advantages:

a) There always exists exactly one time instant (t*) for which the interpolation polynomial is zero, whereas the direct (not inverse) polynomial has either one or three real solutions, and

b) it is not necessary to determine the root of a cubic polynomial to find the solution, that is the inverse interpolation will be faster.

Fig. 4.5 illustrates this iteration scheme.

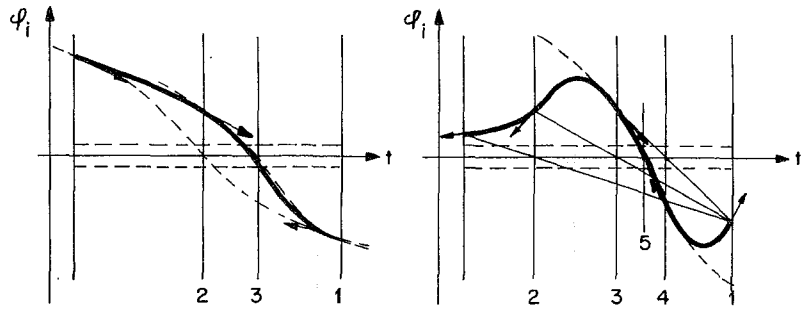

Fig. 4.5: Iteration of a discontinuity function using inverse Hermite' interpolation

A very elegant and simple algorithm can be determined for this iteration procedure. This algorithm is subsequently described.

Let the same two quantities ZL and ZN be given as before. This time, however, they are not indexed since we assume that not more than one crossing takes place within one iteration step. This is no restriction in generality since we may utilize the generalized Regula-Falsi algorithm until the interval is sufficiently reduced to contain one crossing

only. Furthermore we use the two quantities DZL and DZR to denote dz/dt at the left and at the right end of the interval resp.. Also the two time instances TL and TN of the two interval ends are known.

The interpolation polynomial can be written as:

$$t(z) = a*z^3 + b*z^2 + c*z + d$$

The four unknown parameters a, b, c, and d of the polynomial can be computed from the four equations:

$$\begin{aligned} TL &= a*ZL^3 + b*ZL^2 + c*ZL + d \\ TN &= a*ZN^3 + b*ZN^2 + c*ZN + d \\ TL' &= 3a*ZL^2 + 2b*ZL + c \\ TN' &= 3a*ZN^2 + 2b*ZN + c \end{aligned}$$

where TL' and TN' denote the derivatives of t with respect to z at the left and at the right end of the interval resp..

The four equations are to be solved for the unknown parameters a, b, c, and d. These can then be substituted back into the interpolation polynomial. Finally, we set z to zero to find the time of the zero crossing (t*).

To simplify this computation, let us apply a linear transformation to the polynomial:

$$\zeta = v*z + w$$

where v and w are determined in such a way that ζ is equal to 0.0 where z is equal to ZL, and ζ is equal to 1.0 where z is equal to ZN.

t	I	z	I	ζ
TL	I	ZL	I	0.0
TN	I	ZN	I	1.0
t*	I	0.0	I	$\hat{\zeta}$

For v and w we find the values:

$$v = 1.0/(ZN-ZL) \quad ; \quad w = -ZL/(ZN-ZL) = \hat{\zeta} \; .$$

Now we can construct four auxiliary polynomials in ζ:

$$\begin{aligned} p_i(\zeta) &= \alpha_i*\zeta^3 + \beta_i*\zeta^2 + \gamma_i*\zeta + \delta_i \\ p_i'(\zeta) &= 3\alpha_i*\zeta^2 + 2\beta_i*\zeta + \gamma_i \end{aligned}$$

such that

$$\begin{aligned} &p_1(0) = 1 \; ; \; p_1(1) = 0 \; ; \; p_1'(0) = 0 \; ; \; p_1'(1) = 0 \\ &p_2(0) = 0 \; ; \; p_2(1) = 1 \; ; \; p_2'(0) = 0 \; ; \; p_2'(1) = 0 \\ &p_3(0) = 0 \; ; \; p_3(1) = 0 \; ; \; p_3'(0) = 1 \; ; \; p_3'(1) = 0 \\ &p_4(0) = 0 \; ; \; p_4(1) = 0 \; ; \; p_4'(0) = 0 \; ; \; p_4'(1) = 1 \end{aligned}$$

These polynomials are illustrated in Fig. 4.6.

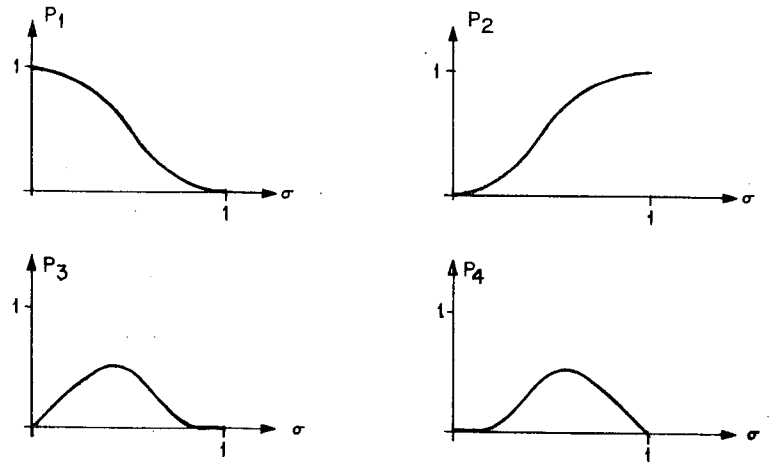

Fig. 4.6: Graph of the four auxiliary polynomials

Each of these auxiliary polynomials has four unknown parameters which can be determined in a unique manner out of the four imposed conditions. The resulting polynomials are:

$$p_1(\zeta) = 2*\zeta^3 - 3*\zeta^2 + 1$$
$$p_2(\zeta) = -2*\zeta^3 + 3*\zeta^2$$
$$p_3(\zeta) = \zeta^3 - 2*\zeta^2 + \zeta$$
$$p_4(\zeta) = \zeta^3 - \zeta^2$$

as it is easy to check.

The inverse Hermite' interpolation polynomial:

$$t(\zeta) = e*\zeta^3 + f*\zeta^2 + g*\zeta + h ,$$

which we are looking for, can be rewritten in terms of the four auxiliary polynomials as:

$$t(\zeta) = TL*p_1(\zeta) + TN*p_2(\zeta) + DTL*p_3(\zeta) + DTN*p_4(\zeta) .$$

DTL and DTN denote the derivatives of t with respect to ζ at the left and at the right end of the interval resp.. They are computed as:

$$DT_i = \frac{dt}{d\zeta} = 1.0/\frac{dt}{d\zeta} = 1.0/(v*\frac{dz}{dt}) = \frac{ZN-ZL}{dz/dt} .$$

This is easily verified by evaluating this polynomial and its derivative $dt/d\zeta$ at $\zeta = 0.0$ and $\zeta = 1.0$ resp.. Due to the uniqueness of this representation (the four $p(\zeta)$ form a base), this is the polynomial we were looking for.

To obtain the crossing time (t*), we simply evaluate this polynomial at $\zeta = \hat{\zeta}$.

This iteration scheme has a cubic convergence ratio but <u>finite</u> convergence range. To be sure that the solution of the interpolation polynomial lies within the considered interval, we require that the first time derivative of the d.f. at both interval bounds points "into" the interval (as illustrated by the arrows on Fig. 4.5). Mathematically, this rule can be expressed as:

$$ZL*DZL < 0.0$$
$$ZN*DZN > 0.0 .$$

If one of these two conditions is not fulfilled, it is safer to perform one step using Regula-Falsi and then again check as to whether the conditions are fulfilled within the newly obtained reduced interval. The two algorithms are perfectly compatible with each other, and there is no problem in switching back and forth between them. (This is one of the reasons why inverse Hermite' interpolation seems more favourable than the previously used Newton-Raphson algorithm [4.5], which has also a finite convergence range and shows quadratic convergence speed, but is much less compatible to

Regula-Falsi than the inverse Hermite' interpolation scheme.)

The combination of the two algorithms (inverse Hermite' interpolation and Regula-Falsi), thus, guarantees fast convergence within an unlimited convergence region. This is illustrated in Fig. 4.5b.

A prerequisite for applying this scheme is that the first time derivative of the d.f. is available. If the derivative is not analytically computable, it would theoretically be possible to approximate it numerically by computing test-steps. This, however, usually does not pay off in terms of computer time, and, under these circumstances, it is, in most cases, faster to apply the Regula-Falsi algorithm. We require, therefore, that the d.f. takes a special form when the inverse Hermite' interpolation is to be used:

$$\phi(\underline{x},t) = x[i] \pm \text{const}$$

where x[i] is the i-th state variable, and $\phi(\underline{x},t)$ denotes the discontinuity function. A state-event takes place when x[i] takes a value of $\mp$const.

In this case the first time derivative can be written as:

$$\dot{\phi}(\underline{x},t) = \dot{x}[i]$$

x[i] denotes the left side of the i-th state equation and is computed anyhow. This type of d.f. is called special discontinuity function. Fig. 4.7 shows how an inverse Hermite' interpolation algorithm can be coded.


```
C*****CHECK WHETHER SIGN OF DERIVATIVES ALLOWS
C*****FOR COMPUTATION, ELSE DO REGULA-FALSI
      IF ((ZL*DZL).GT.(-EEPS))  GO TO 1
      IF ((ZN*DZN).LT.EEPS)  GO TO 1
C*****INVERSE HERMITE' INTERPOLATION ALGORITHM
      HA = ZN - ZL
      DTN = HA/DZN
      DTL = HA/DZL
      HA = -ZL/HA
      HB = HA*HA
      HC = HB*HA
      PB = 3.0*HB - 2.0*HC
      PA = 1.0 - PB
      PC = HC - 2.0*HB + HA
      PD = HC - HB
      TE = TL*PA + TN*PB + DTL*PC + DTN*PD
C*****CHECK WHETHER THE COMPUTED VALUE OF TIME (TE)
C*****LIES WITHIN THE INTERVAL [TL,TN], ELSE DO RF
      HA = TN - TL
      HB = TE - TL
      HC = TN - TE
      IF ((HA*HB).LT.EEPS)  GO TO 1
      IF ((HA*HC).LT.EEPS)  GO TO 1
C*****EVERYTHING IS OKAY. COMPUTE STEP SIZE
      DTNEX = TE - TL
      RETURN
C*****SOMETHING WENT WRONG. DO REGULA-FALSI
    1 CALL  RGFLS (ZN, ZL)
      RETURN
```

Fig. 4.7: Inverse Hermite' Interpolation

This algorithm has been coded in a straight forward manner. However, there may be derived numerically better behaving codes for this iteration scheme. It is feasible (and would be more suitable) to code the inverse Hermite' interpolation scheme as a Regula-Falsi with a correction term (P.Henrici:

private communication). This can be easily achieved by analytically substituting the expressions for P and H into the expression for TE, and finally separating the Regula-Falsi rule out of the resulting expression for TE.

The iteration procedures, as presented so far, are not the only means to solve the problem. They constitute, however, the most general solution to it since they operate independently of the integration algorithm in use. The same piece of code can, thus, be modularly combined with any numerical integration software in question.

However, for some special integration methods there exists another solution to this problem which is more elegant. This concerns modern multi-step integration algorithms, and is subsequently described.

The classical explicit multi-step algorithm extrapolates the next state of the system out of the values of the state variables and their first time derivatives at previous meshpoints

$$x(t+dt) = h(x(t), x(t-dt), \ldots, x(t-k*dt), \dot{x}(t), \dot{x}(t-dt), \ldots, \dot{x}(t-k*dt))$$

The classical multi-step method usually operates with a fixed step-size throughout the integration since, by changing dt, one obviously misses the required information at the appropriate meshpoints (which makes these algorithms inattractive for use in a robust general purpose simulation software). To overcome this difficulty, modern multi-step integration algorithms transform the values of the state variables collected over k meshpoints into values of the state variables and their k time derivatives at one meshpoint (time t). This method has been described by Nordsieck [4.9]. These values are composed to form the vector:

$$\underline{N}' = (x, \dot{x}, \ddot{x}, \ldots, x^{(k)})$$

which is called the Nordsieck-vector. The information contained in the Nordsieck vector is equivalent to the previously used information.

Integration to the new time (t+dt) is done by a simple Taylor series expansion around time t, an algorithm which is obviously applicable for any step size dt. For this reason, the Nordsieck vector grants an easy mean to modify the step size of multi-step integration algorithms. If, at the end of one integration step, we detect that a special d.f. has crossed through zero, we may interpolate the affected state variable to its threshold by using the Nordsieck vector. This has been shown in [4.2.] By these means, the interpolation algorithm is characterized by the same order of accuracy as the numerical integration itself. A more accurate interpolation does not make much sense anyhow. In this way we can replace the (expensive) iteration by a simple interpolation. The only disadvantage of this method is its dependency on the integration algorithm.

IV.1.2.4) Transformation of General Discontinuity Functions into Special Discontinuity Functions:

For d.f.'s which cross through zero often during a simulation run, it is important to use the fastest iteration procedure available. For this reason we want to show how a large group of general d.f.'s can be transformed into special d.f.'s.

Let a system description be given as:

$$\dot{x}[i] = f[i](\underline{x},t) \; ;$$
$$x[i](t=t0) = x0[i] \; ; \; i=1,\ldots,n$$

together with a set of general d.f.'s:

$$\phi[j](\underline{x},t) = g[j](\underline{x},t) \; ; \; j=1,\ldots,m$$

for which $\dot{g}(\underline{x},t)$ shall exist.

This problem can be transformed into the following equivalent one:

$$\dot{x}[i] = f[i](\underline{x},t) \; ;$$
$$x[i](t=t0) = x0[i] \; ; \; i=1,\ldots,n$$
$$\dot{x}[n+j] = \dot{g}[j](\underline{x},t) \; ;$$
$$x[n+j](t=t0) = g[j](\underline{x}=\underline{x0},t=t0) \; ; \; j=1,\ldots,m$$

together with the new set of special d.f.'s:

$$\phi[j](\underline{x},t) = x[n+j] \; ; \; j=1,\ldots,m$$

The price for this transformation is an enlargement of the state space by m additional state equations. Whether this transformation pays off in terms of computing time depends on the problem. For d.f.'s crossing through zero only occasionally, it will certainly not be justified to integrate additional state variables to save iteration time. For other d.f.'s which have zero crossings at short time intervals, this transformation can save a remarkable amount of CPU-time. This technique has been applied in the GASP-V program coded in Fig. 3.4.

Warning:
This transformation can be numerically harmful as illustrated in Fig. 4.8.

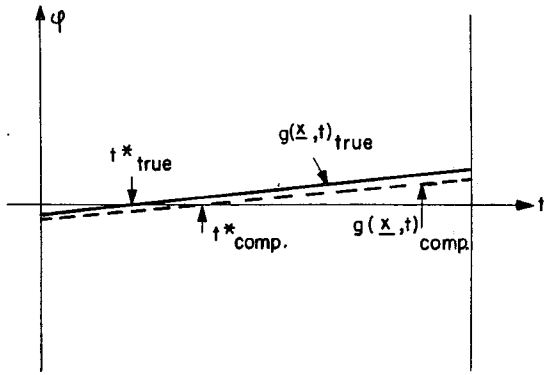

Fig. 4.8: Danger of transforming a general d.f. into a special d.f.

After the transformation, $g[j](\underline{x},t)$ is no longer explicitely specified by the user, but is obtained through numerical integration out of $\dot{g}[j](\underline{x},t)$. In this way, numerical integration errors which are unavoidable and which cannot be compensated are introduced. If the slope of $g[j](\underline{x},t)$ in the neighbourhood of the zero crossing is small, the resulting error in t* is much larger than the error in $g[j](\underline{x},t)$ as illustrated in Fig. 4.8, and it does not help at all to iterate the new $\phi[j](\underline{x},t)$ down to a very small value (!).

Using a one step integration algorithm, this problem can be overcome by resynchronizing x[n+j] with $g[j](\underline{x},t)$ after each integration step or at least once per communication interval (constant interval for data storage). Using a multi-step integration algorithm, this procedure is not recommended since we would have to restart the integration algorithm thereafter. Use of this transformation has been demonstrated in the SCR control problem (Fig. 3.4). In this example, the concerns, as expressed above, are irrelevant since all cros-

sings have a sufficiently large slope.

IV.1.2.5) Location of Short-Living Discontinuities:

So far we have assumed that an interval in which the d.f. crosses through zero has already been found. To obtain this interval, we compute the value of the d.f. once after each successfully accomplished integration step.

However, it may happen that a d.f. crosses through zero several times within one integration step. If the number of crossings is even, the algorithm has no chance to detect the crossings at all, otherwise one crossing is detected but all others will most probably be lost. Such d.f.'s are called short-living discontinuity functions.

If we assume, that, for example, no d.f. crosses through zero more than twice during any integration step, we can use the following procedure to guarantee detection of all crossings:

Given a set of general d.f.'s:

$$\phi[j](\underline{x},t) = g[j](\underline{x},t) \; ; \; j=1,\ldots,m$$

Assuming that $\dot{g}[j](\underline{x},t)$ exists, we can enlarge this set by additional m d.f.'s of the form:

$$\phi[m+j](\underline{x},t) = \dot{g}[j](\underline{x},t) \; ; \; j=1,\ldots,m$$

The actions to be taken when one of the additional (artificial) d.f.'s crosses through zero is none. However, if the k-th original d.f. $\phi[k]$ crosses through zero twice within an integration step, the partner-d.f. $\phi[m+k]$ crosses through zero precisely once in between as illustrated in Fig. 4.9.

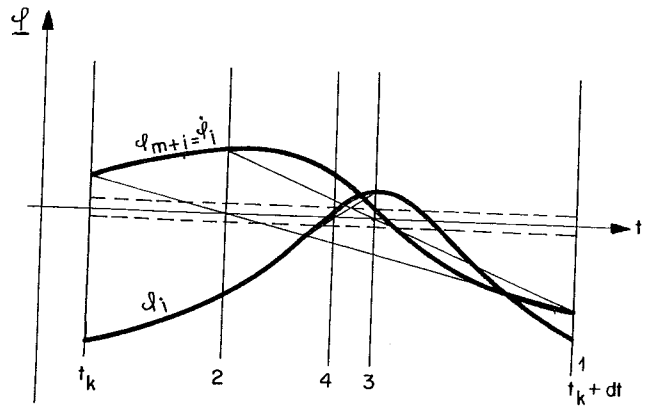

Fig. 4.9: Detection of short-living discontinuity functions

The algorithm which checks for possible realizations of d.f.'s will notice the crossing of the additional d.f. and enable iteration. During the step-size reduction procedure, the "true" d.f. must be located, since the crossing of the partner-d.f. lies between the two crossings of the original one.

When three crossings are allowed within one integration step, the second time derivative must also be added to the set of d.f.'s, etc..

When detection of short-living d.f.'s is to be guaranteed, the user must be able to make an assumption concerning the highest frequency present in the Fourier analysis of each d.f..

The price for properly detecting short-living d.f.'s are additional d.f.'s which must be watched and possibly iterated for nothing.

IV.1.3) Selection of the Initial Subsystem:

Having discussed the conditions for branching from the discrete to the continuous subsystem and vice-versa, it remains to determine the subsystem to be used first at initialization time, t0. The rule is simply to start with the discrete subsystem. This will then check whether there are any events to be treated at time t0 and if not transfer control to the continuous simulation package in which case there would be no action taken (this under the assumption that differential equations form part of the system's description at time (t0+dt), otherwise proceed as described in section IV.1.1).

IV.2) Program Flow:

From the previous statements, it can be concluded that the only new structural elements in combined continuous/discrete system simulation are the state-events and their associated state-conditions. Concerning the numerical aspects these structural elements result in additional step-size control algorithms which must replace the ordinary step-size control mechanism (for error control) during location of the (unknown) event time of a state-event. Once this event time has been located, a state-event is an event as any other and no longer requires any special treatment.

Fig. 4.10 illustrates the general program flow in a combined simulation package. Two dashed squares denote the discrete event simulation package and the continuous system simulation package.

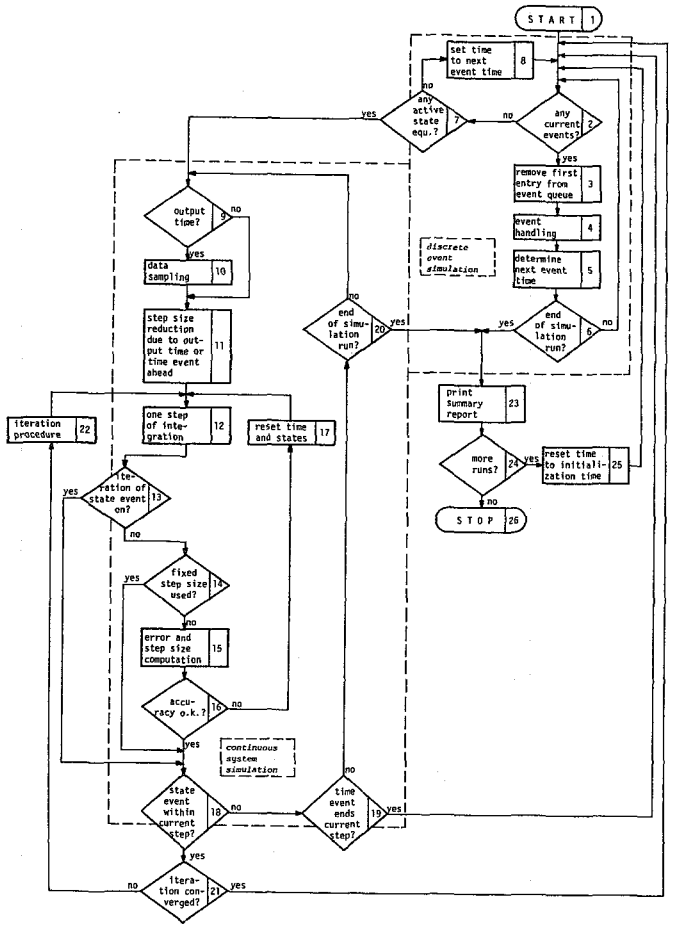

Fig. 4.10: Program flow in a combined system simulation package

First the discrete simulation square is entered, and the question is posed whether there are events scheduled for initialization time. If yes, the first event will be immediately executed. Event execution involves:

a) removing of the event from the event queue,

b) event handling (which is user coded and may include anything from scheduling of new future time-events to collection of statistics and modification of the structure of the ODE set for the next integration period) and

c) determination of the next event time (which may be the current time again if more than one time-event is scheduled to take place at the same time instant).

As soon as there are no events left for handling at the current (initialization) time, the question is asked whether there are any ODE's currently involved in the simulation. If this is not the case, the current time is set to the next event time and a normal discrete event simulation is carried out. If there are currently active ODE's in the system's description, control is transferred to the continuous system simulation package.

The continuous system simulation package will first determine whether there are currently data to be stored for later printout. After that, it will check whether the next integration step can be performed with the proposed step-size or whether the step-size has to be reduced due to a scheduled event time or a data sampling instant or final time of the simulation run being shortly ahead. This is necessary since such time instants are not to be bypassed. Now one step of integration can be performed. If no state-event iteration is in progress, and if a variable step-size integration algorithm is used, the program will now compute an estimate for the local integration error, and propose a new step-size for

the next step to be taken. Now the program must check whether the current step is acceptable or has to be repeated with a smaller step-size because of accuracy requirements not being met. If the step can be accepted, the program will determine whether a state-event has been bypassed during the last completed integration step. If so, the iteration procedure will be activated to locate the state-event precisely, and the ordinary step-size control is disabled during iteration. As soon as the event has been found, control is returned to the discrete event simulation package for event handling. If no state-event took place during the actually completed integration step, the program has to determine whether a time-event ends the current integration step. If this should be the case, control is also immediately returned to the discrete event simulation package for event handling, whereas otherwise the next integration step can be performed.

One additional test is required to determine when the actual simulation run is to be terminated. Once this has been detected, the simulation run will be interrupted, and the question is asked whether additional runs are to be performed or not. Accordingly, the system will be reinitialized or the program will be terminated.

In the following chapter we shall describe a software product (GASP-V [4.4]) in which all the algorithms described in the current chapter have been successfully implemented and tested. GASP-V has been released in May 1978 and has been implemented meanwhile on about a dozen installations of different computer makes throughout the world.

References:

[4.1] P.R.Benyon: (1976) "Improving and Standardizing Continuous System Simulation Languages". Proc. of the SIMSIG Simulation Conference, Melbourne, Australia, May 17-19, 1976; pp. 130 - 140.

[4.2] M.B.Carver: (1977) "Efficient Handling of Discontinuities and Time Delays in Ordinary Differential Equation Simulations". Proc. of the SIMULATION'77 Symposium, Montreux, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 153 - 158.

[4.3] F.E.Cellier: (1975) "Continuous-System Simulation by Use of Digital Computers: A State-of-the-Art Survey and Prospectives for Development". Proc. of the SIMULATION'75 Symposium, Zurich. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 18 - 25.

[4.4] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[4.5] F.E.Cellier, Rufer D.F.: (1975) "Algorithm Suited for the Solution of Initial Value Problems in Engineering Applications". Proc. of the SIMULATION'75 Symposium, Zurich, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 160 - 165.

[4.6] W.Kreutzer: (1976) "Comparison and Evaluation of Discrete Event Simulation Programming Languages for Management Decision Making". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 429 - 438.

[4.7] R.Mannshardt: (1977) "Simulation of Discontinuous Systems by Use of Runge-Kutta Methods Combined with Newton Iteration". Proc. of the SIMULATION'77 Symposium, Montreux, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 163 - 167.

[4.8] R.N.Nilsen, Karplus W.J.: (1974) "Continuous-System Simulation Languages - A State-of-the-Art Survey". Annales de l'Association Internationale pour le Calcul Analogique (AICA), no. 1, January 1974; pp. 17 - 25.

[4.9] A.Nordsieck: (1962) "on the Numerical Integration of Ordinary Differential Equations". Journal of Mathematics and Computers, Vol. 16; pp. 22 - 49.

[4.10] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

V) GASP-V:

V.1) The GASP Program Family:

Fig. 5.1 illustrates the major members of the GASP program family tree:

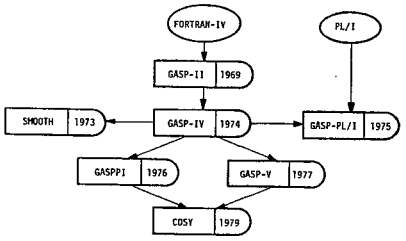

Fig. 5.1: The GASP program family tree

The first widely used program package in the GASP family of programs was GASP-II [5.15] which was a purely discrete event simulation package released in 1969 by Pritsker and Kiviat. This program was later augmented by some elements from continuous system simulation to form GASP-IV.

GASP-IV [5.14] was released in 1974 and was the first combined continuous/discrete system simulation package which reached a sufficiently high status of software reliability and documentation to allow unlimited external distribution. Previously announced programs (e.g. GSL [5.9]) never reached this level, and consequently have not been used at many installations other than at the site where they originated. GASP-IV soon became very popular and is the widest used program for combined simulation to date. This is probably because of to its excellent documentation and maintenance.

Out of GASP-IV, several programs have emerged.

GASP-PL/I [5.16] is a PL/I coded version of GASP-IV which takes advantage of the far better data structuring capabilities of PL/I as compared to FORTRAN-IV, the "mother tongue" of GASP-IV.

SMOOTH [5.18] has a topological input description language (network simulator), and will be discussed later. This is one of the programs which has never really become widely used (as it is the case with most of the programs on the software or, at least, software manual(!) "market").

GASPPI [5.20] is a true descendant of GASP-IV. Its discrete simulation capabilities have been augmented by a process description facility which may be used in addition to the event description philosophy formerly used by GASP-IV. Its continuous part remained unchanged. These additional elements bring the level of modeling comfort closer to the standards reached by discrete event simulation languages like GPSS-V [5.17] or SIMULA-67 [5.6]. A disadvantage of using these modeling facilities, however, lies in the fact that the transparency of the program becomes greatly reduced. If something "goes wrong", the user will usually find it much more difficult to determine the bug. GASPPI has seen some external implementations, but also never reached an "adult" status so far.

Although GASP-IV has been widely accepted and used, it is certainly not optimal from the continuous system analysis point of view. One easily notes that GASP-IV was born out of a discrete event simulation family, and that the package obtained its continuous system simulation capabilities only through marriage. For this reason we tried to overcome these shortcomings by developing another descendant of GASP-IV which is called GASP-V [5.4,5.5]. This package is entirely upwards compatible with GASP-IV. In this package, the dis-

crete simulation part has been left as it was defined in GASP-IV, whereas the continuous part has been remarkably improved and enhanced. GASP-V is, furthermore, the first simulation package which is able to simulate systems out of all three problem classes (ODE's, PDE's and discrete events). This will be illustrated subsequently in an example. The PDE facilities offered have merely been adopted for use in GASP-V from another program package, FORSIM-V (which was the predecessor of FORSIM-VI [5.2]).

In the following section we will present the major highlights of GASP-V.

V.2) Improvements of GASP-V as Compared to GASP-IV:

A) GASP-V offers many new run-time control facilities. Due to its input/output philosophy GASP-IV is somewhat unwieldy when used for optimization studies.

In GASP-IV, it is possible to carry out a dynamic number of simulation runs (as used e.g. in an optimization study) by reinitializing the system by hand from within the terminal section of the application program (that is from subroutine OTPUT). This has been illustrated in an example in [5.14]. To do this, one requires, however, a profound insight into the structure of the simulation package itself -- an understanding which lies far beyond the capabilities of the average user. It seems, therefore, more favourable to outwit the package by incrementing the number of remaining runs by one (NNRNS = NNRNS + 1) within subroutine OTPUT each time an additional run is to be caused. In this way, the reinitialization can be left to the package. This is, however, a trick, and application programs making use of it will lack readability. GASP-V, therefore, offers additional subroutines (RERUN, CONTN and FINIS) for

run-time control. These subroutines have similar meaning as subroutines RERUN, CONTIN and FINISH offered in CSMP-III [5.23]. (For reasons of software portability, no GASP variable takes more than five letters.)

In most cases, however, the user might find it bothersome to code the optimization strategy by himself. He would prefer to fall back upon a precoded nonlinear programming package (which, in itself, may be of similar size as the simulation package!). Fig. 5.2 illustrates how a program to carry out a dynamic parameter optimization study should be structured.

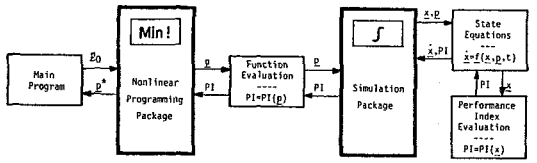

Fig. 5.2: Structure of a program for dynamic parameter optimization studies.

The user first specifies an initial guess for the parameter vector ($\underline{p0}$), and calls the optimization package. This program will call another user-supplied subprogram in which the performance index (PI) is to be computed as a function of a given parameter set ($\underline{p}$).

$$PI = PI(\underline{p})$$

Each function evaluation involves one whole simulation run, since in reality PI is not directly specified as a function of $\underline{p}$, but as a function of the state vector $\underline{x}$:

$$PI = PI(\underline{x})$$

where $\underline{x}$ is determined by a set of first order ODE's:

$$\dot{\underline{x}} = \underline{f}(\underline{x},\underline{p},t) \; ; \; \underline{x}(t=t0) = \underline{x0}(\underline{p})$$

The optimization package will contain one or several strategies to modify $\underline{p}$ in such a way as to minimize (or maximize) the performance index (PI). Finally the optimal parameter set ($\underline{p^*}$) is returned to the calling program.

As can be seen, the user is supposed to call the nonlinear programming package from the main program, whereas the simulation package must be called (and thus callable) from within a subroutine.

This can be realized in GASP-IV by calling the nonlinear programming package from the main program and by calling subroutine GASP from within the function evaluation routine of the nonlinear programming package. Unfortunately GASP-IV prompts the user to supply the whole set of GASP data cards each time subroutine GASP is called, which is certainly inconvenient during an optimization study. These reflections led to a slight modification of the effect of calling subroutine GASP in GASP-V. Here the user will be asked to supply only those data cards of which he had declared in the previous run that they would change, as if he would have initiated the new run from within subroutine OTPUT. (This, as a matter of fact, comprises the only incompatibility between GASP-IV and GASP-V.)

Comparing the two possible ways of initiating an optimization study, the only advantage of using subroutine RERUN lies in the fact that GASP-V in that case will automatically suppress all intermediate output until the final run is executed, whereas this remains the responsibility of the user when calling subroutine GASP

from within a loop. (In that case the package has no means to find out whether an additional run will be due or not.) On the other hand, the optimization strategy will have to be user coded when subroutine RERUN is utilized, whereas any good nonlinear programming package may eventually solve the problem in the former case.

B) In GASP-IV, integration is performed by a Runge-Kutta-England algorithm of 4th order [5.14] built directly into the execution subroutine GASP. This restricts the use of GASP-IV to such systems for which a Runge-Kutta integration is adequate. In GASP-V, integration has been separated from the simulation control mechanisms (for this purpose the control routine GASP has been entirely recoded), and a library of good integration algorithms is now at the user's disposal (including, among others, a Runge-Kutta-Fehlberg algorithm of 5th and one of 8th order [5.7] as well as the Gear-Kahaner package for the solution of stiff systems [5.8,5.10,5.11]). This facility is demonstrated in the programs of Fig. 3.3 and Fig. 3.4 (Runge-Kutta-Simpson of 4th order being used), and in Fig. 5.6 (where the Gear-Kahaner package is called). A further motivation for providing this facility in GASP-V will be given in the discussion of the heat diffusion example (section V.3).

C) GASP-V allows for forward and backward integration (which may be, for example, useful for the solution of Riccati equations).

D) GASP-V contains additional subroutines for comfortably simulating systems described by sets of partial differential equations eventually coupled to ODE's as well.

PDE's are reduced to sets of ODE's by the method-of-lines approach [5.1,5.2,5.3]. This will be demonstrated in an example in section V.3.

E) As described previously, GASP-V provides for more sophisticated techniques for the location of state-events than GASP-IV, which always uses bi-section. GASP-V leaves it up to the user to decide whether a particular state-condition is to be iterated by inverse Hermite' interpolation, by Regula-Falsi, or by bi-section. This feature has been outlined in detail in section IV.1, and is demonstrated in Fig. 5.4b and Fig. 5.4c.

F) GASP-V offers a comprehensive library of run-time functions including hysteresis-function, pulse generator, delays, algebraic loop solver etc. (very much like the run-time library offered by CSMP-III). This enables the user to model his system by continuous modeling techniques as he would in utilizing a language like CSMP. In GASP-V, however, all discontinuities of built-in run-time functions (so called "GASP-functions") will be resolved by "hidden" time and/or state-events, although the user need not to be aware of this fact. By these means all creeping effects, which are well known from CSSL-type software, are automatically avoided. Examples of the use of GASP-functions are presented in the SCR control problem coded in Fig. 3.3.

Although it is as easy to make use of this continuous modeling technique in GASP-V as in any CSSL-type language, the library is not as easily extendable by the user. Though provisions have been made to expand the library, and the manner in which such GASP-functions are to be coded is fully documented [5.4,5.5], the user requires a profound understanding of the software to make proper use of this possibility.

G) GASP-IV possesses very limited output representation facilities for continuous systems. The reason is simply that comfortable plot routines consume a quite remarkable amount of core memory.

For this reason it seems more advisable to store data in a file at run-time and let them be transformed into graphical representations later by a separate output program. Since such output packages already exist in the literature, it seems unproductive to "reinvent the wheel" by coding a new one. In GASP-V, we have chosen to use the DARE-P postprocessor [5.12,5.19] for output representation since the facilities offered by this software are very generous, and because the programming style of DARE-P is as strict and cautious as in the GASP software. For this purpose, a DARE-P compatibility mode has been created in which data are written onto a file at sampling instances in such a way that the DARE-P postprocessor must "think" that they have been generated by the DARE-P run-time system. This postprocessor interprets commands coded in a very simple command language, and includes modules for cross-plots, over-plots, high quality plots for (x,y) plotting devices, etc..

H) Since the DARE-P language has no provisions for PDE's, the original DARE-P postprocessor does not contain modules for three dimensional graphical representations. A new version has, therefore, been coded for use with GASP-V which contains some additional modules (and commands). These include print plots similar to the PAGE SHADE and PAGE CONTOUR options offered in CSMP-III [5.23] as well as 3-dim. plots with hidden lines removed

and moveable viewing position (central projection) for (x,y) plotting devices. Also these output facilities are illustrated in the heating example which follows.

V.3) Example - Simulation of a Heating System:

V.3.1) Statement of the Problem:

To illustrate the capabilities of GASP-V let us consider as an example the central heating system of a building. The building is modeled by a stick of length unity. The left end of it represents the center of the building with the heating source, whereas its right end denotes the walls. The temperature distribution in the building is modeled using the diffusion equation:

$$\frac{\partial u}{\partial t} = 0.5 * \frac{\partial^2 u}{\partial x^2}$$

The heating of the central room is modeled by a first order ODE of the following form:

$$\dot{z} = 4.0*(35.0 - z) \; ; \; z(t=t0) = 0.0$$

z represents the temperature in the center of the building where the heating system is situated. The ODE constitutes at the same time one of the two required boundary conditions for the PDE specified at the left end of the stick:

$$u(x=0.0,t) = z(t)$$

The second boundary condition may be specified by the radiation at the walls, which is valid at the right end of the stick:

$$\frac{\partial u}{\partial x}(x=1.0,t) = 4.0E\text{-}9*(UA(t)**4 - u(x=1.0,t)**4)$$

where UA(t) denotes the outside temperature. In this equation, UA(t) and u(x,t) must be expressed in absolute temperature (degrees Kelvin), whereas centigrades (degrees Celsius) are used otherwise as a measurement unit for all temperatures throughout the program.

The model, as it has been described so far, is, however, applicable only when the heating is "on". During times where the heating is "off", the ODE no longer holds, and the left boundary condition of the PDE is replaced by a symmetry condition of the form:

$$\frac{\partial u}{\partial x}(x=0.0,t) = 0.0$$

Each time the heating is turned "on", the number of differential equations (NNEQD) is incremented by one (to model the additional ODE z) and z obtains as initial condition the actual value of the temperature in the heating room:

$$z(t=t^*) = u(x=0.0,t=t^*)$$

When the heating is turned "off", the number of differential equations must be decremented again by one.

Conditions for heating are the following: At night time (between 7 p.m. and 7 a.m.) the heating is always "off". This determines two different alternating time-events.

During day time the heating is turned "on", as soon as the temperature at the walls falls below 19.5C and is turned "off" when the temperature at the walls raises above 22.5C. This is specified by two state-conditions. The simulation is said to start at 6 a.m. with a temperature of 0.0C throughout the building.

Fig. 5.3 illustrates the model of the building.

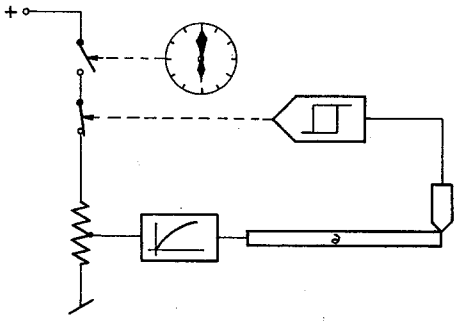

Fig. 5.3: Model of the central heating system of a building

This concludes the description of the problem.

V.3.2) Simulation Objectives:

This example shows how, by use of the GASP modeling philosophy, a rather complicated model can be partitioned into small functional blocks which are much easier to understand and to code. All different modeling elements (ODE's, PDE's, time-events and state-events) are present in this example and operate together.

The model, although quite realistic, is not meant to be a valid model of a real heating system. It is artificially

constructed to illustrate the capabilities of the GASP-V software to tackle combined problems.

V.3.3) Special Features:

This example demonstrates the modeling of PDE's in GASP-V, and how these may coexist beside of ODE's and discrete events.

Furthermore, it is shown how data can be stored in the DARE-P compatibility mode to obtain two and three dimensional high quality graphs on (x,y) plotting devices.

An additional feature which is demonstrated is the use of different integration algorithms.

Finally, the number of differential equations (NNEQD) involved in this simulation is time-dependent.

V.3.4) The Method-of-Lines Approach to PDE Problems:

Let a system of n PDE's of the following type be given:

$$\frac{\partial \underline{u}(x,t)}{\partial t} = \underline{f}\{\underline{u}(x,t), \frac{\partial \underline{u}(x,t)}{\partial x}, \frac{\partial^2 \underline{u}(x,t)}{\partial x^2}, x, t\}$$

$$\underline{u} \in \mathbb{R}^n$$
$$t \in [0.0, \infty)$$
$$x \in [xl, xu]$$

with boundary conditions of the form:

$$B1(t)*\frac{\partial \underline{u}(x,t)}{\partial x} + B2(t)*\underline{u}(x,t) = B3(t)$$

$$t \in [0.0,\infty)$$
$$x = \{xl\},\{xu\}$$

and initial conditions:

$$\underline{u}(x,t=0.0) = \underline{u0}(x) \ .$$

The system of n PDE's can now be converted to a system of n*m ODE's by means of a differential quadrature, representing the solutions $\underline{u}$ at m grid points by approximating polynomials:

$$x \dashrightarrow \underline{\xi} = \{1,2,\ldots,m\} \quad \text{(indices)}$$

leading to:

$$U_t(t) = F\{U(t),U_x(t),U_{xx}(t),\underline{\xi},t\}$$

$$U \in \mathbb{R}^{n\times m}$$
$$t \in [0.0,\infty)$$
$$\underline{\xi} = \{1,2,\ldots,m\}$$

In this new formulation, the problem can be divided into three easier-to-solve sub-problems:

a) computation of the spatial derivatives, $U_x(t)$ and $U_{xx}(t)$, from U(t) by difference schemes for any given values of time t,

b) computation of the state derivatives, $U_t(t)$, from the state space description of the problem and,

c) computation of U(t+dt) from U(t) by means of numerical integration.

Using this formulation, it is evident that there is no methodological difference between simulating only one PDE or systems of several coupled PDE's eventually accompanied by ODE's. (There is, of course, a large influence on core memory requirements, computing time, etc.!)

An advantage of this methodology lies in the fact that we do not have just one specific algorithm available for the solution of PDE problems, but may select an appropriate scheme:

a) by choosing the best suited integration algorithm from the available library,

b) by choosing the optimal differentiation scheme for the computation of the spatial derivatives and,

c) by optimizing the grid width of the spatial discretization.

V.3.5) Simulation Procedure:

For detailed understanding of the GASP-V program presented in this section, knowledge of the GASP software as described in [5.4,5.14] will be useful. Most of the statements are, however, properly explained, to make this thesis as self-documentary as possible.

In our heat diffusion problem, the PDE

$$\frac{\partial u}{\partial t} = 0.5*\frac{\partial^2 u}{\partial x^2}$$

is transformed into a set of NNDIV first order ODE's

$$\dot{U}[i] = 0.5*UXX[i] \; ; \; i=1,...,NNDIV$$

At initialization time (t0), all U[i] are known from initial conditions. By use of these values we can compute the second spatial derivatives using any valid differentiation scheme. This is done in GASP-V by a call to subroutine PRSET:

```
CALL PRSET (MSET, NNDIV, U, UT, UX, UXX)
```

where MSET denotes the number of active PDE's involved in the simulation (which is one in our example), and NNDIV determines the number of grid points used in space (corresponding to the number of ODE's obtained per PDE).

Through GASP data cards (variable NNDIF), the user can specify the difference scheme to be used by subroutine PRSET (like 3, 5 or 7 point central formulae).

The boundary conditions (b.c.'s) are specified through a master scheme as presented for the general case above, where BBL(i,k) represents the boundary function Bi(t) for the k-th PDE specified at the lower bound, and BBU(i,k) denoting the same quantity at the upper bound (i=1,2,3). NNL(k) and NNU(k) determine the form the associated b.c. takes, where NNi(k)=0 means that the b.c. takes precisely the form as specified in the general case above, and NNi(k)=-2 means that no b.c. is imposed.

When the spatial derivatives at time t0 have been computed, we may use the state space description of the problem to next compute the first time derivatives:

```
      DO 1000  I=1,NNDIV
      UT(I) = 0.5*UXX(I)
1000 CONTINUE
```


Now we can integrate the state equations over time to obtain new values for U[i] at time t=t0+dt. The integration algorithm will adapt its step size according to the largest gradient in the system. Since this is often found to be at the interval bounds where integration really is not needed (the newly obtained value for u(x=xl,t) or u(x=xu,t) will be immediately overwritten again by the specified boundary value), we want to avoid controlling the step-size by this gradient. This is accomplished by calling subroutine PRFIN:

```
CALL PRFIN (MSET, NNDIV, U, UT, UX, UXX)
```

at the end of the description which will nullify these gradients. This subroutine call has no other effect than (hopefully) to reduce the execution time. It could as well have been omitted. As for the computation of the spatial derivatives, any of the available integration algorithms may be used.

It can be seen from the description that the method-of-lines is not really one particular algorithm for the solution of PDE problems, but a specific solution methodology describing a whole class of different algorithms out of which the best appropriate may be chosen (which is not a trivial problem at all).

The method-of-lines approach to the solution of PDE problems can be thought of as a philosophy in which a n-dimensional PDE is solved for the highest derivative in one of the n dimensions (usually time). The other (n-1) dimensions (usually space) will then be discretized, and all associated derivatives will be computed by means of numerical differentiation. The description of the PDE can then be used to compute the highest derivative in the "last" dimension. This is left pseudo-continuous, and the solution vector is obtained by means of numerical integration in this "last" dimension.

From this procedure results either an initial value ODE problem or a boundary value ODE problem depending on the type of b.c.'s imposed on the original PDE problem, and depending on the dimension the PDE is solved for.

The subroutines PRSET and PRFIN (as well as some further internally used subroutines) have been adopted from FORSIM-V as has the whole PDE description philosophy. More details can be found in [5.2,5.4].

Subroutine STATE is shown in Fig. 5.4a.

```
      SUBROUTINE  STATE
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNO(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /GCOM7/ AAUX,AAUXX,DDX,NNDIF,NNDIV,NNEQX,XX(300),XXL,XXU
      LOGICAL  AAUX, AAUXX, NNEQX
      COMMON /GCOM8/ BBL(3,10),BBU(3,10),NNL(10),NNU(10)
      COMMON /OCOM1/ SWS
      COMMON /UCOM1/ PI,SW,STATEV,TIMEV,UZERO
      DIMENSION  U(11), UT(11), UX(11), UXX(11)
      EQUIVALENCE (SS(1),U(1)),(SS(12),Z),(DD(1),UT(1)),(DD(12),ZT)
C
C*****SELECT APPROPRIATE MODEL FOR LEFT BOUNDARY CONDITION OF PDE
C*****AS A FUNCTION OF SW=TIMEV*STATEV
C
      IF (SW.EQ.1.0)  GO TO 1
C
C*****SW=0.0 :  BUILDING IS COOLING DOWN
C
      BBL(3,1) = 0.0
      GO TO 2
C
C*****SW=1.0 : BUILDING IS HEATED
C
C*****COMPUTE FIRST ACCOMPANYING ODE
C
    1 ZT = 4.0*(35.0*SW - Z)
C
C*****COMPUTE NOW THE BOUNDARY CONDITION AS A FUNCTION OF Z
C
      BBL(3,1) = Z
C
C*****COMPUTE OUTSIDE TEMPERATURE AS A FUNCTION OF TIME
C
    2 UA = 5.0*SIN (PI*TNOW/12.0)
C
C*****COMPUTE NOW THE OTHER BOUNDARY CONDITION, WHICH IS IDENTICAL
C*****FOR BOTH MODELS
C
      UK = U(NNDIV) + UZERO
      UAK = UA + UZERO
      BBU(3,1) = 1.0E-9*(UAK**4 - UK**4)
C
C*****COMPUTATION OF SPATIAL DERIVATIVES
C
      CALL  PRSET (1, NNDIV, U, UT, UX, UXX)
C
C*****STATE SPACE DESCRIPTION OF THE PDE
C
      DO 1000  I=1,NNDIV
      UT(I) = 0.5*UXX(I)
 1000 CONTINUE
C
C*****CORRECTION OF THE TIME DERIVATIVES
C
      CALL  PRFIN (1, NNDIV, U, UT, UX, UXX)
C
C*****STORE DATA FOR LATER OUTPUT
C
      SWS = 10.0*SW
      RETURN
      END
```

Fig. 5.4a: FORTRAN listing of subroutine STATE for the heat diffusion problem.

This subroutine is used to code the state space description of the model. State variables are automatically stored once per communication interval for later use by the DARE-P post-processor. This has been specified on GASP data cards

(Fig. 5.5). Additional quantities (as SWS in our example) must be collected in a special COMMON block (/OCOM1/).

Fig. 5.4b shows the listing of subroutine SCOND for the specification of the state-conditions.

```
      SUBROUTINE SCOND
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /GCOM7/ AAUX,AAUXX,D9X,NNDIF,NNDIV,NNEQX,XX(300),XXL,XXU
      LOGICAL  AAUX, AAUXX, NNEQX
      COMMON /GCOM14/ AAZ(10),IIZ(10),JJZ(10),KKZ(10),NNZ,ZZ(10)
      COMMON /UCOM1/ STATEV,TIMEV,UA
      DIMENSION  U(11)
      EQUIVALENCE  (SS(1),U(1))
      ZZ(1) = U(NNDIV) - 19.5
      IIZ(1) = NNDIV
      JJZ(1) = -1
      ZZ(2) = U(NNDIV) - 22.5
      IIZ(2) = NNDIV
      JJZ(2) = 1
      RETURN
      END
```

Fig. 5.4b: FORTRAN listing of subroutine SCOND

Here we demonstrate how the new iteration schemes as presented in section IV.1.2 can be used. ZZ(I) determines the i-th discontinuity function (d.f.) which is to be iterated to zero. ZZ(I) is accompanied by some additional quantities:

IIZ(I) determines the iteration scheme to be used. If IIZ(I) = 0, the Regula-falsi rule is to be used. If IIZ(I) = k where: $1 < k \leq$ NNEQD, the inverse Hermite' interpolation algorithm will be used to iterate the i-th special d.f., which takes the form:

$$\phi[I] = x[k] \pm \text{const}$$

or in terms of GASP-V:

$$ZZ(I) = SS(K) \pm CONST$$

to zero.

JJZ(I) determines the direction of crossings to be detected and iterated. It is possible to detect positive

or negative crossings or both. This is equivalent to the LDIR parameter of the KROSS function offered in GASP-IV (which is, of course, still accessible in GASP-V as well).

AAZ(I) corresponds to the TOL parameter of the KROSS function to denote the accuracy requirements to be met by the iteration. Since these iteration schemes tend to converge far faster than the bi-section rule as it is used in the KROSS function of GASP-IV, the required tolerance range may be specified much more stringently without leading to unduly high computational costs.

KKZ(I) is a flag corresponding to LFLAG(I) in GASP-IV to tell the user, at event time, whether a particular crossing took place.

NNZ finally determines the number of active d.f.'s (corresponding to NFLAG).

Fig. 5.4c shows the listing of the EVNTS subroutine for event handling.

```
      SUBROUTINE  EVNTS (IX)
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NHATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ QQ(100),QQL(100),QTFUL,QTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQQ,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /GCOM7/ AAUX,AAUXX,QQX,NNDIF,NNDIV,NNEQX,XX(300),XXL,XXU
      LOGICAL  AAUX,AAUXX, NNEQX
      COMMON /GCOM8/ BBL(3,10),RBU(3,10),NNL(10),NNU(10)
      COMMON /GCOM14/ AAZ(10),IIZ(10),JJZ(10),KKZ(10),NNZ,ZZ(10)
      COMMON /UCOM1/ PI,SW,STATEV,TIMEV,UZERO
      DIMENSION  U(11)
      EQUIVALENCE  (SS(1),U(1)),(SS(12),Z)
C
C*****BRANCH TO APPROPRIATE EVENT
C
      GO TO (1,2,3), IX
C
C*****EVENT CODE 1: GOOD MORNING
C*****SET TIMEV TO BUSY AND SCHEDULE NEXT 'GO-TO-BED' EVENT TO TAKE
C*****PLACE IN 12 HOURS FROM NOW
C
    1 ATRIB(1) = ATRIB(1) + 12.0
      ATRIB(2) = 2.0
      CALL  SCHED
      TIMEV = 1.0
      IF (STATEV.EQ.0.0)  GO TO 3
      BBL(1,1) = 0.0
      BBL(2,1) = 1.0
      SW = 1.0
      Z = U(1)
      NNEQQ = NNDIV + 1
      RETURN
C
C*****EVENT CODE 2: GOOD EVENING
C*****SET TIMEV TO IDLE AND SCHEDULE NEXT 'WAKE-UP' EVENT TO TAKE
C*****PLACE IN 12 HOURS FROM NOW
C
    2 ATRIB(1) = ATRIB(1) + 12.0
      ATRIB(2) = 1.0
      CALL  SCHED
      TIMEV = 0.0
      BBL(1,1) = 1.0
      BBL(2,1) = 0.0
      SW = 0.0
      NNEQQ = NNDIV
      RETURN
C
C*****EVENT CODE 3: STATE EVENT CODE
C*****CHECK WHICH OF THE DISCONTINUITY FUNCTIONS HAS ACTUALLY CROSSED
C*****THROUGH ZERO AND SET STATEV ACCORDINGLY.
C
    3 CONTINUE
      IF (KKZ(1).GE.0)  GO TO 4
      AAZ(1) = -AAZ(1)
      STATEV = 1.0
      IF (TIMEV.EQ.0.0)  RETURN
      BBL(1,1) = 0.0
      BBL(2,1) = 1.0
      SW = 1.0
      Z = U(1)
      NNEQQ = NNDIV + 1
      RETURN
    4 CONTINUE
      IF (KKZ(2).LE.0)  RETURN
      AAZ(2) = -AAZ(2)
      STATEV = 0.0
      BBL(1,1) = 1.0
      BBL(2,1) = 0.0
      SW = 0.0
      NNEQQ = NNDIV
      RETURN
      END
```

Fig. 5.4c: FORTRAN listing of subroutine EVNTS

Three event codes are used in this example:

event code	event type	flag indicator	definition and name
1	time-event	----	switching from night to day service
2	time-event	----	switching from day to night service
3	state-event	KKZ(1)<0	temperature at the walls has decreased below 19.5C.
		KKZ(2)>0	temperature at the walls has increased above 22.5C.

Event activities are the following for:

event 1 ("morning" event):

A) Schedule a new event to take place in 12 hours from now (ATRIB(1) = TNOW + 12.0) with event code 2 (ATRIB(2) = 2.0) to switch to night service again.

B) Turn the time switch (TIMEV) "on".

C) Detect whether the model must be changed. If the state switch (STATEV) is in "off" position, nothing really happens. Otherwise the heating must now be turned "on", that is:

a) the new b.c.'s must be specified,

b) the model switch (SW) must be reset to 1.0,

c) the initial condition for the accompanying ODE (Z) is to be established and,

d) the number of differential equations is to be incremented by one for the additional ODE.

event 2 ("evening" event):

A) schedule a new "morning" event to happen in 12 hours from now.

B) Turn the time switch "off".

C) Use, in any event, the "heating = off" model, that is:

a) reset the b.c.'s,

b) set the model number (SW) to zero and

c) set the number of ODE's accordingly.

event 3 (state-event):

A) If the first state-event is realized (KKZ(1) < 0), the temperature at the walls has fallen below 19°.5C, and we must:

a) turn the state switch (STATEV) "on".

b) Since the same state-condition is modeled to be valid for the next integration period as well, and since we cannot guarantee that the temperature is really below 19°.5C when using inverse Hermite'

interpolation or Regula-Falsi (cf. section IV.1.2.1), there is a risk that the iteration procedure "detects" the same d.f. immediately again. The statement:

AAZ(1) = -AAZ(1)

tells the package to close its eyes during the first integration step for possible crossings of the d.f. number 1.

c) As in event 1 (C) we must now detect whether the heating is to be enabled. This is the case if the time switch (TIMEV) is currently in "on" position.

B) If the temperature has risen above 22°.5C (KKZ(2) > 0),

a) the state switch has to be turned "off".

b) The package must now close its eyes for possible crossings of the d.f. number 2 during one integration step.

c) The heating must be in "off" position.

It is not necessary to test the two state-events consecutively here, since they are disjoint (the temperature cannot be less than 19°.5C and greater than 22°.5C simultaneously).

Fig. 5.4d shows the listing of subroutine INTLC for establishment of the initial conditions.

```
      SUBROUTINE  INTLC
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1MAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /GCOM7/ AAUX,AAUXX,DDX,NNDIF,NNDIV,NNEQX,XX(300),XXL,XXU
      LOGICAL   AAUX,AAUXX,NNEQX
      COMMON /GCOM8/ BBL(3,10),BBU(3,10),NNL(10),NNU(10)
      COMMON /UCOM1/ PI,SW,STATEV,TIMEV,UZERO
      DIMENSION  U(11)
      EQUIVALENCE (SS(1),U(1)),(SS(12),Z)
C
C*****COMPUTE MATHEMATICAL NUMBER PI
*
      PI = 4.0*ATAN (1.0)
C
C*****SCHEDULING OF FIRST TIME EVENT TO TAKE PLACE AT TNOW=1.0
C*****WITH EVENT CODE 1 (TURNING ON DAY SERVICE)
C
      ATRIB(1) = 1.0
      ATRIB(2) = 1.0
      CALL  SCHED
C
C*****UPSETTING OF BOUNDARY CONDITIONS FOR THE PDE, AS FAR AS THESE ARE
C*****TIME INVARIANT
C
      NNL(1) = 0
      BBL(1,1) = 1.0
      BBL(2,1) = 0.0
      NNU(1) = 0
      BBU(1,1) = 1.0
      BBU(2,1) = 0.0
C
C*****UPSETTING OF INITIAL CONDITIONS FOR PDE
C
      DO 1000  I=1,NNDIV
      U(I) = 0.0
 1000 CONTINUE
C
C*****SINCE ONLY THE SECOND SPATIAL DERIVATIVE IS NEEDED, AAUX IS SET
C*****FALSE TO SAVE COMPUTING TIME
C
      AAUX = .FALSE.
C
C*****SETTING OF FLAGS (SWITCHES) AND OF OUTSIDE TEMPERATURE
C*****WHICH IS 0.0 DEGREES CENTIGRADE, BUT IS TO BE COMPUTED IN
C*****DEGREES KELVIN FOR THE RADIATION
C
      TIMEV = 0.0
      STATEV = 1.0
      SW = 0.0
      UZERO = 273.3
      RETURN
      END
```

Fig. 5.4d: FORTRAN listing of subroutine INTLC

This subroutine is self explanatory. No initial conditions are specified for the ODE, since the heating is always "off" at 6 a.m., and since z, for this reason, has no meaning at initialization.

Fig. 5.4e shows the listing of the main program.

```
 PROGRAM MAIN (INPUT,OUTPUT,MONITR,TIME,CROSS,SAVE,TAPE1=MONITR,TAP
1E2=TIME,TAPE3=CROSS,TAPE4=SAVE,TAPE5=INPUT,TAPE6=OUTPUT)
 COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
1MAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
2,TTCLR,TTFIN,TTRIB(25),TTSET
 COMMON  QSET(8)
 NCRDR = 5
 NPRNT = 6
 CALL  GASP
 CALL  BYE
 END
```

Fig. 5.4e: FORTRAN listing of the main program

As is usual in GASP, this program consists of three executable statements only,

a) to set the input device to logical file number 5,

b) to set the output device to logical file number 6 and,

c) to call the simulation (CALL GASP).

This will be true as long as no optimization is performed by the program.

Subroutine BYE is used instead of the ordinary FORTRAN "STOP" statement for proper closing of the output storage files used in the DARE-P compatibility mode (MONITR, TIME, CROSS and SAVE). This cannot be automated since the package has no means to determine whether another simulation run will follow when returning control to the calling program.

Fig. 5.5 shows the GASP-V echo check for the input data to this problem.

```
                    SIMULATION PROJECT NUMBER   3  BY       GELLIER

                    DATE 11/ 15/ 1978       RUN NUMBER    1 OF    1
                    LLSUP=0000000000000000  GASP V  VERSION 25MAY78

 NCRDR=    5   NPRNT=    6   MMONT=    1   IICRS=    3   IISAV=    4   IITIM=    2
 NNOTP=    1   NNMXS=   12

 NNAMS=SN            U             U2            U3            U4            U5
 NNAMS=U6            U7            U8            U9            U10           U11
 NNAMS=Z

 NNCLT=    0   NNSTA=    0   NNHIS=    0   NNPRM=    0   NNPLT=    0   NNSTR=    1   NNTRY=    2
 NNATR=    2   NNFIL=    1   NNSET=    8   NNEQD=   11   NNEQS=    0   NFLAG=    0   NNPDE=    1

 NNDIF=    3   NNDIV=   11   NNX  =    0
 XXL  =   0.                 XXU  =     .1000E+01

 KKRNK= (   1)

 IINN = (   1)

 IIEVT=    3   LLERR=    0   AAERR=     .1000E-02          RRERR=     .1000E-02
 DTMIN=     .1000E-03        DTMAX=     .2500E+00          DTSAV=     .2500E+00

 NNZ  =    2   IIDIS=    0   EEPS =     .1000E-13          NNINT=  5000   NNITR=  100
 AAZ( 1)=   .1000E-05   AAZ( 2)=    .1000E-05   AAZ( 3)=  -0.          AAZ( 4)=  -0.
 AAZ( 5)=  -0.          AAZ( 6)=          -R    AAZ( 7)=         -R          AAZ( 8)=          -R
 AAZ( 9)=          -R   AAZ(10)=          -R

 MSTOP=    1   JJGLR=    0   JJBEG=    1   IICRD=    0   TTBEG=   0.          TTFIN=     .3600E+02
 JJFIL=    1
 IISED=   -8

 PARTIAL DIFFERENTIAL EQUATION  1 WILL BE DISCRETISED USING  3 POINT FORMULAE AND 11 EQU. SPATIAL DIVISIONS
```

Fig. 5.5: GASP-V echo check of input data for the heating system

In addition to the normal data cards used in GASP-IV programs, there are additional data cards found:

a) for the specification of the DARE-P compatibility mode,

b) for proper initialization of PDE's and,

c) for the specification of d.f.'s.

This concludes the description of the GASP-V program to simulate the heat diffusion problem.

V.3.6) Results:

When this program is computed as formulated above, a Runge-Kutta-Fehlberg integration algorithm of 5th order will be

used (default method). Using a 3 point central formula for computation of the spatial derivatives, one run of the heat diffusion problem consumes 39.7 sec of CPU-time.

Fig. 5.6 shows the listing of an additionally supplied subroutine INTEG which is added to replace the default integration method by the Gear-Kahaner package [5.11].

```
      SUBROUTINE  INTEG
      DIMENSION  WORK(204)
C
C*****THIS SUBROUTINE IS USED TO SELECT THE APPROPRIATE
C*****INTEGRATION ALGORITHM
C
      CALL  GEAR (J, 264, 5, 2, 12, WORK)
      RETURN
      END
```

Fig. 5.6: FORTRAN listing of subroutine INTEG

Using this algorithm the same problem requires only 8.3 sec of CPU-time. A gain of 4.8 can, thus, be achieved by these means.

When 33 discretization points are used and a 7 point central formula is applied for computation of the spatial derivatives, the Runge-Kutta algorithm requires 620.0 sec of CPU-time, whereas the Gear algorithm can solve the problem now within 26.9 sec of CPU-time. In this case there results a gain of 23.0 in favour of the Gear algorithm.

This shows that the problem is obviously stiff, and becomes even stiffer for smaller values of the grid width (dx).

How can this result be explained? Considering the scaled PDE problem

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

and using a 3 point central formula for the computation of the spatial derivatives, the resulting ODE system will have a Jacobian of the form

$$J = (1,-2,1)*(dx^{-2})$$

This is a time-invariant tri-diagonal matrix. The eigenvalues of this matrix are known to be all negative and real. For dx sufficiently small, the largest of the eigenvalues, (λ_1), approaches $-\pi^2$ and the smallest eigenvalue, (λ_n), approaches minus infinity as $-dx^{-2}$ which is in accordance with the obtained results [5.13].

How can this special result be of general interest? It is obviously difficult to predict, in a mathematically proper way, how stiff an ODE system resulting from conversion of any PDE using any particular differentiation scheme will be. We want to show, however, that any PDE problem will usually be transformed into a stiff ODE system by the method-of-lines approach.

Let us consider any PDE of the type presented in section V.3.4. It is certainly legitimate to express neighbouring solutions (that are solutions at neighbouring grid points) as

$$U[k] = 0.5*(U[k-1] + U[k+1]) + o(dx) \ .$$

That is, for dx sufficiently small, an almost linear relationship results for neighbouring solutions. This will then lead to an almost singular Jacobian which is equivalent to a stiff problem formulation. For the solution of almost any PDE problem, Runge-Kutta algorithms will, therefore, not be suitable. This is another reason why we have made the decision to separate integration from simulation control.

Many PDE problems have already been coded in GASP-V and this

intuitively formulated statement has been found to be true in almost all of the cases. In the heat diffusion problem, for instance, the resulting ODE system is as stiff for the 7 point formula as for the 3 point formula.

Fig. 5.7 shows the listing of DARE-P output commands specified for the heat diffusion problem.

```
* HEAT DIFFUSION PROBLEM  ( GEAR  -  NNDIV=11  -  NNDIF=3 )
DPLOT,U(C,0.0,0.0,11)
NULLIFY
FACTOR(,2.0,2.0)
GRAPH,U,U11,SW
ALTERNATE(,15.0,-10.0,50.0)
CALCOMP,U(C,0.0,0.0,11)
END
```

Fig. 5.7: DARE-P output commands for the heating system

Fig. 5.8a to Fig. 5.8c show the output produced by these commands.

Fig. 5.8a: Output obtained by the DPLOT command

DPLOT will produce output on a line printer similar to that obtained by using the PAGE CONTOUR or PAGE SHADE option offered in CSMP-III [5.23].

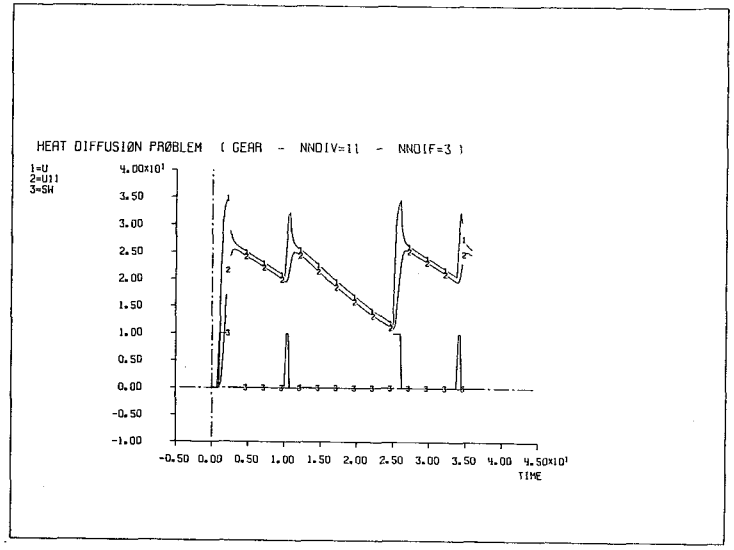

Fig. 5.8b: Output obtained by the GRAPH command

GRAPH is a command to obtain 2-dim. plots. In this example, the temperatures in the heating room (U) and at the walls (U11), as well as the switch function (SW) denoting "on" and "off" status of the heating system, are plotted versus the simulation clock.

The heating system seems to be somewhat overdimensioned for this building and reacts very hectically (no wonder, if we tell the program that the heating system is able to heat the central room from 0°.0C to 35°.0C within 15 min!). When the heating is turned "off", the temperature will quickly equalize throughout the building. There is obviously hardly any thermal insulation between the rooms. Later on, the temperature decreases smoothly due to the radiation at the (much better insulated) walls.

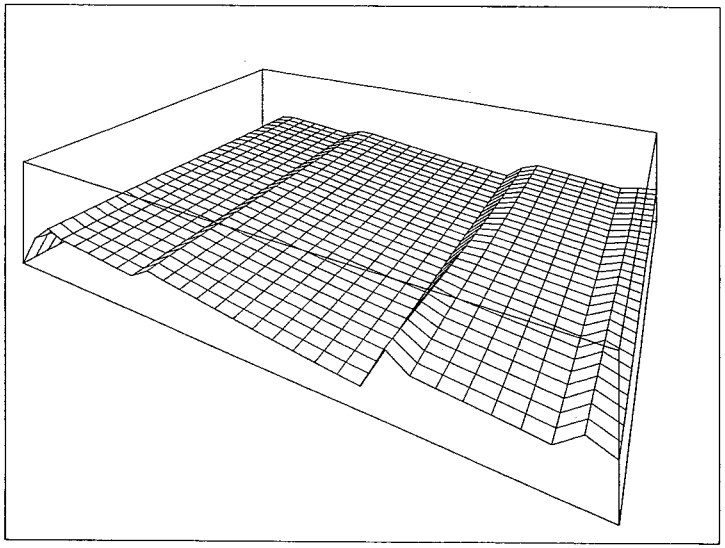

Fig. 5.8c: Output obtained by the CALCOMP command

Fig. 5.8c shows a 3-dim. plot with hidden lines removed as obtained by the CALCOMP command. The viewing position is determined by the previously specified ALTERNATE command. The FACTOR command determines the size of the graphs and the commands starting with '*' denote titles.

This concludes the analysis of the heating system.

V.4) Unsolved Problems:

Many problems have been tested using the GASP-V software and the results were quite promising. There are still two unsolved problems:

A) Taking the definition of Pritsker [5.14] for event times:

> "An event occurs at any point in time beyond which the status of a system cannot be projected with certainty"

it is clear that an infinite density of events must not occur. This may, however, happen in at least the following two cases:

a) A system is modeled by a set of PDE's and discontinuities exist. In this case the discontinuity may "walk" through space with time and can no longer be localized in the way proposed in section IV.1. As an example, let us consider a long electric wire where a current is imposed at one end which suddenly (at time t1) changes its value. This discontinuity will remain in the system for some time and "flow" through the wire. If the effects of reflection at opposite ends are taken into account, the disturbance caused by the introduction of the discontinuity may even remain in the system "forever". Thus, in this example, we will find that for any instant of time, t>t1, the system will be discontinuous at one particular point in space (x1) which is variable with time.

b) The behaviour of the continuous subsystem is stochastic in nature. The spectrum of a random number stream has infinite frequencies which has the effect that it is nowhere differentiable. If such a random number is superposed to the input of an integrator, we face the problem mentioned above. This holds, of course, only for stochastic behaviour of the continuous subsystem and not for the discrete subsystem. Stochastic interarrival times of cus-

tomers to a queue, for instance, will not effect the numerical behaviour of the system, since new samples for the random numbers are only computed at event times. Between event times these variables are constant.

Zeigler [5.21, Chapter 9] has shown that the existence of an infinite density of events always results in an illegitimate model. In the case of (a), it is, theoretically, always possible to respecify the model so that the new equivalent model is no longer illegitimate. In this new formulation, the propagation of discontinuities will follow the axes of the coordinate system. This is well known as the "method-of-characteristics". In the case of the linear wave equation, we know that the characteristics are straight parallel lines and the required variable transformation is easy to achieve. For complex situations (nonlinear cases), however, to find the characteristics of the problem (which are now curves bended in time and space) is almost equivalent to solving the entire problem. Thus while we can solve the problem (a) theoretically, in practice, the required computations for obtaining the variable transformations are extremely tedious and may prevent us from doing so.

Therefore, we usually find another solution for this problem. In using the method-of-lines approach [5.2,5.3], we found that the integration over time is not notably effected in most applications by these discontinuities, whereas the computation of the spatial derivatives is greatly disturbed. Therefore, for each step, we first try to identify the discretization interval in which the discontinuity is situated at that particular instant in time. We then split up the region, and compute the spatial derivatives independently for the two parts lying to the left and to the right of the discontinuity. This procedure can easily be expanded for

several space dimensions as well.

The case of (b) is, in principle, more difficult to treat. Zeigler's characterization of illegitimate models was developed only for discrete event models. However, his discussion of the intrinsic limitation of the class of continuous systems which can be simulated by digital computers [5.21, Chapter 5] and [5.22] may be applied to the present problem. According to this analysis, there must always be a non-zero interval separating computer updates of the model's state. Thus, the computer must guess what the behaviour of the model is in the interval separating computational instants -- the problem of "bridging the gap". Since the computer is given a description of the model components and their coupling, it can guess correctly only if certain conditions enabling perfect interpolation in the gap hold. Polynomial trajectories, commonly assumed in integrating differential equation models, serve this purpose.

In the case of stochastic continuous models, it is not easy to justify the assumption of polynomial trajectories. For example, if the model contains a white noise component then no means of bridging the gap exist in principle. This is because, by definition, the correlation between sample values, however closely spaced in time, is zero. Even if the noise is not white, current numerical methods are not geared to exploiting autocorrelations specified by the model for optimum choice of the integration step. As a result, most step-size control algorithms will produce extremely pessimistic guesses for the step-sizes to be used, resulting in high computational costs.

We found the following approach useful in many applications: First we compute one run by setting the noise to zero using variable step integration (the continuous

subsystem is now deterministic). In this run, we collect statistics (histogram) of the utilized step-sizes dt. From the cumulative frequency curve we select the 0.1 level point (10% of the step-sizes fall below this point). Now we compute a new run, this time with inclusion of the noise, where we keep the step-size dt fixed at this 0.1 level point. A disadvantage of this solution is, of course, that we now have no measurement for the quality of the approximation. Thus, we must be very careful in the interpretation of results obtained in this way. Furthermore, the proposed method can be applied only if the signal/noise ratio is high. For a low signal/noise ratio, we do not know any good numerical technique to get around this problem.

B) The user of a simulation package wishes either to obtain reliable results or have a "bell" ring when an algorithm is unable to perform proper work. Under no circumstances does he want to obtain results which are incorrect (problem of software robustness). To date, this can be guaranteed with a high confidence in the case of ODE problems only. For PDE problems, numerical difficulties need not necessarily be detected by the package, resulting in inaccurate or even entirely incorrect results. The problem arises from the fact that we always use a fixed grid width for the spatial approximation, and thus have no control on the error resulting from this discretization.

The following chapter will discuss this problem in more detail.

References:

[5.1] M.B.Carver: (1975) "Simulation Packages for the Solution of Partial Differential Equation Systems". Proc. of the SIMULATION'75 Symposium, Zurich, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 57 - 64.

[5.2] M.B.Carver: (1978) "The FORSIM-VI Simulation Package for the Automated Solution of Arbitrarily Defined Partial and/or Ordinary Differential Equation Systems". Form: AECL-5821. Atomic Energy of Canada, Ltd.; Chalk River Nuclear Laboratories, Mathematics & Computation Branch, Chalk River, Ontario, Canada K0J 1J0.

[5.3] F.E.Cellier: (1977) "On the Solution of Parabolic and/or Hyperbolic PDE's by the Method-of-Lines Approach". Proc. of the SIMULATION'77 Symposium, Montreux, Switzerland. To be ordered from: ACTA Press, P.O.Box 354, CH-8053 Zurich, Switzerland; pp. 144 - 148.

[5.4] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[5.5] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 391 - 402.

[5.6] O.J.Dahl, Nygaard K.: (1966) "SIMULA: A Language for Programming and Description of Discrete Event Systems". Oslo, Norwegian Computing Center.

[5.7] E.Fehlberg: (1968) "Classical 5th-, 6th-, 7th-, and 8th-order Runge-Kutta Formulas". Report: NASA TR R-287.

[5.8] C.W.Gear: (1971) "Numerical Initial Value Problems in Ordinary Differential Equations". Prentice Hall, Series in Automatic Computation.

[5.9] D.G.Golden, Schoeffler J.D.: (1973) "GSL - A Combined Continuous and Discrete Simulation Language". Simulation vol. 20 no. 1 : January 1973; pp. 1 - 8.

[5.10] A.C.Hindmarsh, Gear C.W.: (1972) "Ordinary Differential Equation System Solver". To be ordered from: Lawrence Livermore Laboratory; Report: UCID 30001, Rev.2.

[5.11] D.Kahaner: (1977) "A New Implementation of the Gear Algorithm for Stiff Systems". This is a new version of the well known Gear-Hindmarsh program [5.10]. Unpublished private communication. For further detail contact: Dr. David Kahaner, University of California, Los Alamos Scientific Research Laboratory, Contract W-7405-ENG-36, P.O.Box 1663, Los Alamos NM 87545, U.S.A..

[5.12] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice Hall.

[5.13] N.K.Madsen: (1975) "The Method of Lines for the Numerical Solution of Partial Differential Equations". SIGNUM-Journal, vol. 10, no. 4 : December 1975, (Special Interest Group on Numerical Mathematics of ACM); pp. 5 - 7.

[5.14] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

[5.15] A.A.B.Pritsker, Kiviat P.J.: (1969) "Simulation with GASP-II". Prentice Hall.

[5.16] A.A.B.Pritsker, Young R.E.: (1975) "Simulation with GASP-PL/I. A PL/I Based Continuous/Discrete Simulation Language". John Wiley.

[5.17] T.J.Schriber: (1974) "Simulation Using GPSS". John Wiley.

[5.18] C.E.Sigal, Pritsker A.A.B.: (1973) "SMOOTH: A Combined Continuous/Discrete Network Simulation Language". Proc. of the 4th Annual Pittsburgh Conference on Modeling and Simulation. Pittsburgh, Penn., U.S.A., April 23-24, 1973; pp. 324 - 329.

[5.19] J.V.Wait, DeFrance Clarke III: (1976) "DARE-P User's Manual". (Version 4.1). To be ordered from: Department of Electrical Engineering, University of Arizona at Tucson, Tucson AZ 85721, U.S.A..

[5.20] W.B.Washam, Pritsker A.A.B.: (1976) "Introduction to GASPPI". Unpublished private communication. For further detail contact: A.A.B.Pritsker, Ph.D., Professor, Pritsker & Assoc., Inc., Consultants in Systems Engineering, P.O.Box 2413, West Lafayette IN 47906, U.S.A..

[5.21] B.P.Zeigler: (1976) "Theory of Modeling and Simulation". John Wiley.

[5.22] B.P.Zeigler: (1977) "Systems Simulateable by the Digital Computer". Logic of Computers Group Report, University of Michigan, Ann Arbor, U.S.A..

[5.23] (1972) "Continuous System Modeling Program III (CSMP-III) - Program Reference Manual". Program number: 5734-XS9, Form: SH19-7001-2. To be ordered from: IBM Canada Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, Canada.

VI) SOFTWARE ROBUSTNESS:

VI.1) Definition:

The term "robust" has, in the past, often been used and misused as an attribute to almost everything because it is considered elegant to call something "robust". The term (as well as others like "automated" or "adaptive") is, therefore, a slogan which very often has been used for no other reason than for marketing, since it seems to increase the number of items sold remarkably, if one assigns this attribute to a product. Consequently, this term is not too well defined, and we must first specify in which context we are going to use it.

In connection with simulation software, this attribute can be assigned to language definitions, to compilers and to run-time software [6.5,6.6].

A) One can talk of a robust simulation language definition in two senses:

   a) It can be robust with respect to modeling, in that it provides for a general scheme for partitioning any application problem in such a way that the resulting submodels are easier to formulate. By these means, the risk for formulating erroneous models can be notably reduced.

      In combined simulation, the concept of subdividing a combined system into a discrete and a continuous subsystem, as described in chapter IV of this thesis, will definitely increase modeling safety. As an example for this let us cite a solar energy heating system. This is a typical combined problem with time-events (sunrise, sunset, good weather, bad

weather) and state-events (the pump for the circulation of the liquid is either "on" or "off" depending on the temperature at the collector, the additional oil heating can be in "on" or "off" status depending on the temperature in the building). We guided several groups of students who were supposed to model such a system during one semester term of 16 weeks length. One student used ordinary FORTRAN programming for the task, and called subroutines for numerical integration and output representation from a library of FORTRAN routines. After sixteen weeks, this student ended up with a tremendous program for which he was unable to draw a proper flow chart. He had entirely lost the overvue of his program, and it never worked. The program was very badly structured from the beginning. Other groups used the GASP software for this modeling task, and found it much easier to construct running (although not necessarily valid (!)) simulation programs.

Another possibility to improve robustness in this respect is to provide facilities for hierarchical constructs.

A language definition can, furthermore, be robust with respect to modeling in that it contains additional redundancy. If the user, for instance, is required to supply dimensions for all variables in a declaration block of his application program, the software can perform an automated dimensional analysis for all equations. If the user is requested to supply ranges for all variables, the software can check whether trajectories during simulation behave as expected.

b) The language definition can be <u>robust with respect to programming</u>. For this purpose, the language

definition must contain enough redundancy so that the software is able to detect as many programming (e.g. typing) errors as possible. The user can, for example, be asked to declare all variables in a declaration block of his program. This will enable the compiler to detect most of the typing errors (like misspelled variables or keywords). The danger of "programming by exception" has been noticed years ago, and most of the modern computer languages take this into account. This knowledge, however, has not yet reached most of the simulation software designers, because in todays simulation languages such features are hardly ever offered. This is probably because most software designers stick too closely to the CSSL-definition [6.21] which was defined before one paid too much attention to questions of software robustness.

By such measures, the user code will become longer than necessary, and user programs will be somewhat "verbose". This may bother some of the inveterate CSSL programmers in the beginning, but modeling safety can, by these means, be remarkable improved.

B) On a second level, one can talk of <u>robust simulation compilers</u> in three senses.

a) A compiler can be <u>robust with respect to programming</u>. This aspect is closely related to the previous one. The simulation compiler should perform extensive error testing while parsing the application program.

This <u>must</u> be done because simulation software is ever increasingly being used by non-specialists in computing, and because the complexity of the application program is dictated by the complexity of

the system under investigation rather than by programming experience and sophistication of the simulation user.

It can be done, because the underlying simulation package (run-time system) is a large program anyhow, consuming quite a lot of core memory for its execution. GASP-V, for example, uses 100.000_8 core memory locations on a CDC 6000 series installation. It, therefore, does no harm to allow the same size for the simulation compiler as well, whereas this cannot be tolerated in a general task language like FORTRAN or PASCAL. Moreover, a "small" student's job in simulation, involving 10 to 20 statements, will cost for its execution at least 10 times as much as a comparable FORTRAN student's program (e.g. to determine the largest element in an array). For this reason we can also allow the simulation compiler to execute about 10 times slower compared to a general task compiler to grant more extensive error checking during compilation. Finally. the possible structures in a dedicated task language are much more rigid than in a general application language. For this reason, additional tests in the compilation phase are feasible.

b) A compiler can be robust with respect to implementation. This aspect of robustness involves insensitivity to alterations in the operating system, the underlying computer hardware, and peripheral equipment.

c) A compiler can be robust with respect to maintenance. If a compiler failure has been detected, or if a person wants to improve the language definition by adding additional features to it, this should be implementable as easily as possible in the simulation compiler, and it should result in as few "dirty" side-effects as possible with respect to the compilation of previously implemented language features.

All these requirements for compiler construction have their implications in how the simulation language must be defined.

The best way to guarantee that a compiler is easily maintainable, for instance, is to define the language as a deterministic left-to-right language for which a one-pass compiler without needs for backtracing can be coded (so called LL(1) language). This has been outlined in [6.3]. It is unavoidable that a complex program like a compiler has some "bugs" in it which are not detected until somebody stumbles upon them by chance. At that time, it is most likely that the programmer of the compiler has left the place already, and is no longer accessible. In such a situation, it is extremely important that somebody else is able to read and understand the compiler to be able to remove the bug. It is then very cumbersome if the software engineer is forced to read and understand the compiler as a whole. In most cases it is not too difficult to identify and isolate the bug within the compiler. A local patch, however, bears the risk of unexpected side effects creating new bugs which are often worse than the removed one (!). Such side effects result mostly from GOTO-statements pointing backward from below to beyond the patch position. If the effect of such a GOTO-statement is not taken into account, the patch creates often troubles which are difficult to explain and to correct. Since LL(1) grammars allow compilers to be written in an almost linear top-down structure, the robustness of such a compiler with respect to its maintainability is remarkably better than in the case of other types of grammars being used.

Robustness with respect to implementation can only be guaranteed if the simulation compiler is realized as a preprocessor. The target language, as well as the language in which the preprocessor is coded, must be high-level languages for which there exist compilers for many different types of computers. This, again, has its implications in the simulation language definition, in that only such features can be offered in that language which are realizable in the target language as well. If, for example, the target language is FORTRAN (as this is the case for most simulation languages at present), the newly defined simulation language must be somewhat restrictive in the data structuring capabilities it offers.

If a compiler should, finally, be robust with respect to programming, the language definition must contain enough redundancies to allow for proper error testing during the parsing procedure. Beside of provisions for a sufficiently high redundancy, also the use of LL(1) grammars can again improve the compiler robustness with respect to programming. Since user programs, if coded in a LL(1) language, can be parsed from left to right by looking only one symbol ahead, illegal symbols must be recognized immediately as such, and may be reported to the user. Bu using LL(1) languages, we can, thus, guarantee that no syntactically incorrect user program is accepted by the compiler.

It cannot be expected that a designer of simulation software produces optimal solutions to all problems all at a time. It is quite common that language definitions are revised after some time. They are modified because of users complaining that some of the formerly offered language structures or simulation features are inconsistent or awkward. They are extended because of the software being exposed to problems it was originally not

designed for.

For this reason, it makes sense to discuss the robustness of a simulation system with respect to its updatability (which includes maintainability as a subset). A simulation system is a new term denoting the union of simulation language, simulation compiler, simulation run-time system, and (last not least!) the documentation volume. Experience has shown that program code is much faster and easier updated than the documentation material. For this reason, it is extremely important to make the documentation as easily updatable as possible. For this task, it is very convenient if the documentation is also developed by use of the computer. The text itself should (as this thesis) be composed by use of a powerful text editing system. It is moreover very useful if the syntactical rules of the language are described by syntax diagrams [6.20] which must form an intrinsic part of the documentation. In [6.3] we have described a general purpose table driven parser program which can process any context-free grammar specified in an extended Backus-Naur form (EBNF) notation, and which can check for LL(1) parsibility. Another program [6.4] can then access the same input file, and can produce syntax diagrams of the language definition on any (x,y) plotting device. By use of the parser program, we can check that the suggested modifications of the language are correct (that is consistent with the rest of the language definition), and that the modified language definition is still LL(1) parsible (and thus deterministic and unambiguous). This can be done, before the compiler is touched. With the help of the syntax diagram drawing program, we can automatically draw new syntax diagrams of the modified language definition which can replace the previous diagrams in the documentation volume. By these means, we can guarantee that the documentation material is as easily updated as possible.

C) On a third level, one can talk of a robust simulation run-time system in two different senses.

a) A run-time system can be robust with respect to procedures. The user should never be required to provide any kind of information which he does not have at his disposal. He should be able to concentrate on those factors which have to do with the statement of his problem, and should be relieved, as much as possible, of all aspects which have to do with the way his problem is executed on the machine. He should be able to describe his system as easily as possible in terms which are closely related to his common language, but must not be required to provide a step-size for the numerical integration or to specify the integration algorithm to be used.

b) A run-time system can be robust with respect to algorithms. The run-time software itself must be able to check whether the produced time responses are "correct" (within a prescribed tolerance range). The user, normally, has a more or less precise (although often not mathematically formulated) knowledge of the system he is investigating. He has, however, hardly any "insight information" into the tool he is using for that task. He is, usually, very credulous (the obtained results must be correct because the computer displays 14 digits!), and he has no means to judge the correctness of the produced results. For this reason, it is vital that each algorithm in the system has its own "bell" which rings as soon as it is unable to properly proceed. Under no circumstances are incorrect results allowed to be displayed to the user.

The aim of this chapter is to focus on such aspects of how to improve run-time system robustness.

VI.2) Automated Selection of Integration Algorithm:

A huge step towards robust simulation software has been taken in the development of step-size controlled integration algorithms. Before these algorithms existed, the user of digital simulation software was required to supply information concerning the step-size to be used -- an information item which he clearly did not have at his disposal. Now, the user can simply provide a tolerance range for the accuracy of the results. This is identical to requesting the user to identify the smallest number in his problem which can be distinguished from zero. This question can certainly be answered by any user, independently of whether he is an expert in numerical mathematics or not, since it is closely related to the physics of the problem, and not to the numerical behaviour of the algorithm.

Available simulation software, up to now, usually offers a comprehensive selection of different integration algorithms. It does, however, not tell the user which would be the most appropriate one for his particular application. In this way, the user is again confronted with making a decision on something he does not really understand. Experience has shown that the majority of the average users always operate with the default integration method implemented in the package which, in most cases, is a Runge-Kutta algorithm of 4th order. Since he does not know what to specify, he simply ignores that question, and after some time of using the software he has even forgotten that the language provides him with the facility to select among different integration algorithms. So far, no integration algorithm could be found which would be able to handle all kinds of problems equally well, and it is more than doubtful whether such an algorithm could be found at all. The user, who does not make use of the facility to select among different integration rules, will, consequently, often waste a lot of computing power. Although much research has been devoted to the development

of different integration methods for the different classes of application problems [6.8,6.10,6.15], the user has, however, no means to easily determine, from the state space description, the problem class to which his particular application belongs. For this reason, the selection of the appropriate integration algorithm should also be automated.

For this purpose, we try to extract features from an application problem during its execution which are supposed to characterize the numerical behaviour of that particular problem as completely as possible. These features are then combined in a feature space in which we can identify specific clusters for which a particular integration method is optimally suited. The proposed methodology for the solution to this problem originates from pattern recognition.

What features may be used for this purpose? A first feature can be associated with the accuracy requirements for the problem. It can be found that the CPU-cost to execute a particular problem can be graphed versus the required relative accuracy as shown in Fig. 6.1.

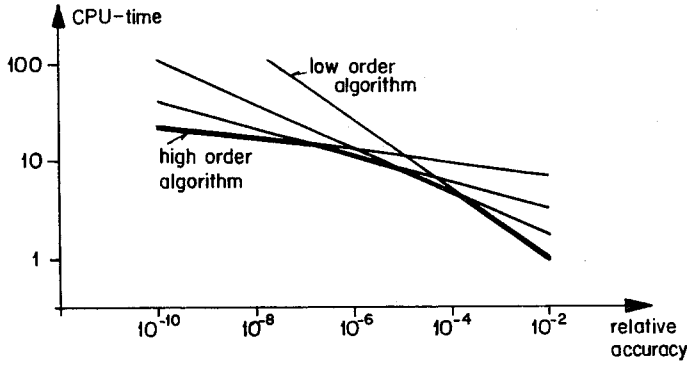

Fig. 6.1: CPU-time versus relative accuracy requirements for different integration algorithms

According to Fig. 6.1, low order algorithms are appropriate for the treatment of systems from the "gray-" and "black box" area where the available data and models are so vague that a precise numerical integration does not make much sense, whereas higher order algorithms are appropriate for the handling of systems from the "white box" area, e.g. from celestrial mechanics. Since the user is requested to specify the wanted relative accuracy, this feature can be extracted from the input data.

A commonly cited "rule of thumb" states that, 10^{-k} being the relative accuracy required, a k-th order algorithm would be close to optimal.

Fig. 6.2 shows the accumulated CPU-cost graphed versus the simulation clock for one-step and multi-step integration.

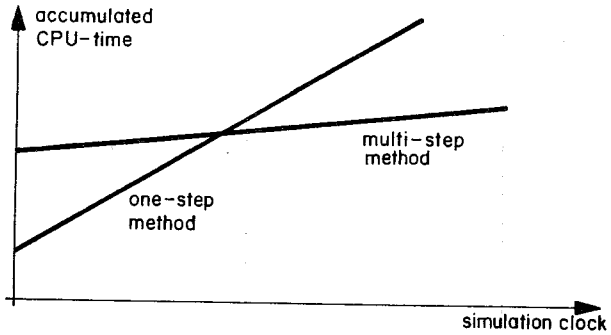

Fig. 6.2: Accumulated CPU-time versus simulation clock for one-step and multi-step integration

As can be seen from Fig. 6.2, multi-step methods require more CPU-time during their initial phase, but are more economic than one-step methods if integration goes on over a longer interval of simulated time. This can be explained by the fact that one-step methods are self-starting whereas

multi-step methods need to be initialized.

This leads to a second feature. Since the integration has to be restarted after event times, multi-step integration is in favour for purely continuous problems or for problems with few event times (smooth problems), whereas one-step integration is appropriate for combined problems where events occur with a high density.

Each integration algorithm has associated with it a domain of numerical stability. This is outlined in Fig. 6.3 for a Runge-Kutta algorithm.

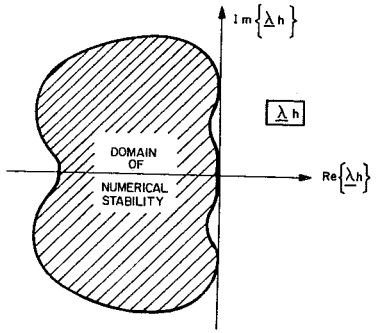

Fig. 6.3: Domain of numerical stability
for a Runge-Kutta algorithm

The stiffer a particular problem is (the more the different eigenvalues (λ_i) of the Jacobian are separated), the smaller the step-size (h) must be in order to keep all (λ_i*h) within the stability region of the algorithm. For this reason, Fig. 6.1 is an idealization, since the step-size to be used (and with it the required CPU-time to execute the problem) depends not only upon the required relative accuracy for the problem, but also upon the boundary of stability of the integration method. Fig. 6.4 is a refinement of Fig. 6.1 de-

picting one particular integration algorithm, where now the stiffness of the problem is the parameter of the graph.

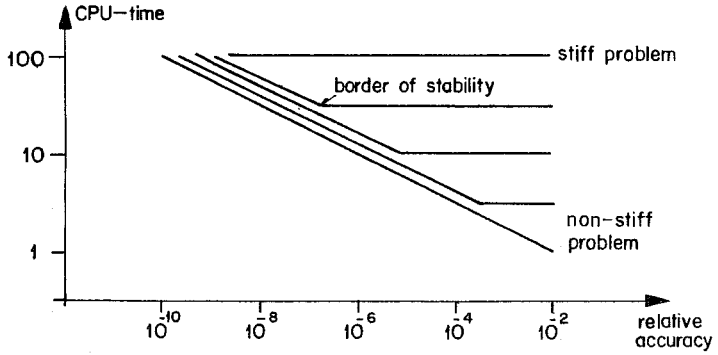

Fig. 6.4: CPU-time versus relative accuracy requirements for varying problem stiffness

Fortunately, special integration algorithms could be found for which this restriction no longer holds, since they show a stability region as outlined in Fig. 6.5 [6.8].

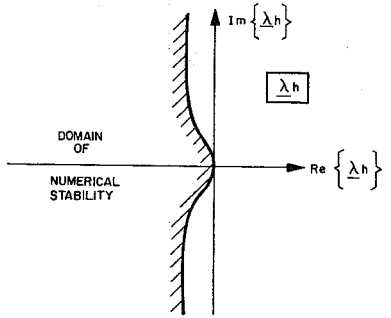

Fig. 6.5: Stability region for a
stiffly-stable integration rule

If such an algorithm is used, the step-size need not be reduced due to restrictions imposed by stability demands, but is determined exclusively by the requirements of accuracy.

For this reason the eigenvalue distribution of the Jacobian determines a third feature which must influence our decision as to which integration algorithm to use for the execution of a particular problem.

These three features can now be combined to a feature space as depicted in Fig. 6.6.

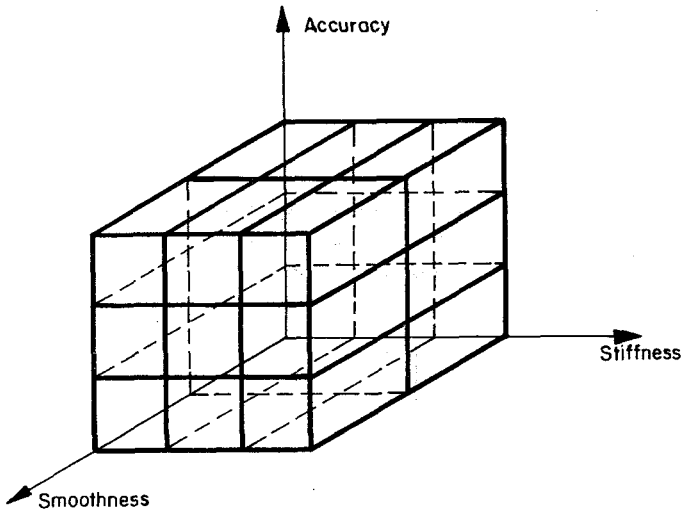

Fig. 6.6: Feature space for selection of integration algorithm

18 clusters have been distinguished in Fig. 6.6, and Fig. 6.7a and Fig. 6.7b show integration rules which can be associated with them.

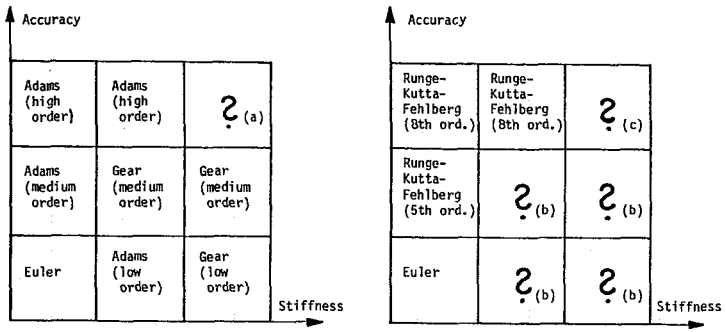

Fig. 6.7: Integration algorithms to be used for smooth (left) and non-smooth (right) problems

As can be seen from Fig. 6.7, some of the assignments are still open.

Concerning (a): A Gear algorithm [6.8] would be appropriate, but it should be of about 8th order (at least for a CDC 6000 series installation -- this depends on the length of the mantissa), whereas, in the Kahaner implementation [6.12] we use, we have only up to 5th order available.

Concerning (b): For these clusters, a one-step algorithm with a stiff stability behaviour would be most suitable. So far, we have experimented with IMPEX-2 [6.16], and with DIRK [6.2], but the programming style of these algorithms, as they are at our disposal at the moment, is not sufficiently elaborate to allow for a fair comparison with the extremely careful and sophisticated Kahaner implementation of the Gear algorithm.

Concerning (c): For this cluster, a high-order stiffly-stable one-step method would be most appropriate. Such an algorithm is, however, unknown to date.

As a matter of fact, the feature space as depicted in Fig. 6.6 is still a simplification. To show this, let us consider a system with complex dominant poles close to the imaginary axis. For the treatment of such a system, a stiffly-stable method as outlined in Fig. 6.5 cannot be properly applied. The system has, however, fast transients, making a Runge-Kutta algorithm not suitable either. Thus, these types of systems, which are called "highly oscillatory" systems, will again require special methods (like stroboscopic methods) for efficient handling (C.W.Gear: Private communication). This establishes a fourth feature which is to be used for the determination of the integration method. The reason for the primary simplification lies in the fact that a 3-dim. feature space can be graphed easier than a 4-dim. one (!).

The information provided by these four features is sufficient to determine the best suited integration algorithm for most application problems.

There are even two more features which can be extracted from the eigenvalue distribution of the Jacobian. These are used for other purposes, and will be presented in due course.

So far we have defined features, and we have associated integration methods with them. It remains to determine how these features can be extracted from the state space description of the problem. The first feature (relative accuracy) is user specified on data input. The second feature (smoothness) could also easily be user provided. It is, however, a simple task to detect automatically whether a problem is continuous or combined. If the problem turns out to be combined, one can count the number of event times

during a certain period of simulated time, and decide then whether it is a smooth or a non-smooth combined problem. Concerning the third and fourth features (stiffness / highly oscillatory behaviour) one has to compute the Jacobian out of the state space description of the problem. This can either be numerically approximated at run-time, or one can compute it algebraically by means of formulae manipulation at compile-time. This is rarely done by available simulation compilers but it is feasible, and seems to be a promising approach. The wanted features can now be computed by estimating the critical eigenvalues. The eigenvalues with the largest and smallest absolute values can be approximated by appropriate matrix norms [6.19, Chapter 6.8], whereas the real part of dominant poles can be found by estimating the "margin of stability" [6.11,6.14]. However, since several quantities are needed, we found that it is in most cases faster to compute the whole set of eigenvalues directly by use of the EISPACK software [6.7]. EISPACK is the best tested eigenvalue software as available of today. The required CPU-time turned usually out to be neglectable compared to the time spent for numerical integration.

According to this discussion, there seems to exist a straight forward approach to code this into an algorithm. However, there exist some hidden problems which deserve to be mentioned here.

The stiffness of a system is commonly defined as:

$$S = \frac{\underset{i}{\text{Max}} \; |\text{Re}\{\lambda_i\}|}{\underset{i}{\text{Min}} \; |\text{Re}\{\lambda_i\}|} .$$

Both, the stiffness (S), and the eigenvalue notation were originally defined for linear time-invariant stable systems only, that is for systems which have constant eigenvalues all lying to the left of the imaginary axis of the λ-plane.

If this is the case, the definition for S as stated above makes sense since the numerator denotes the fastest transient (smallest time constant), and the denominator denotes the slowest transient (largest time constant) in the system. It is assumed that the aim of the simulation is to integrate the model over its entire "transient period", that is until even the slowest transient has settled. Therefore, the length of the simulation is determined by the largest time constant:

$$T \sim \mathrm{Max}(T_i) = \max_i \frac{1.0}{|\mathrm{Re}\{\lambda_i\}|} = \frac{1.0}{\min_i |\mathrm{Re}\{\lambda_i\}|}$$

whereas the step size is a function of the fastest transient:

$$dt \sim \mathrm{Min}(T_i) = \min_i \frac{1.0}{|\mathrm{Re}\{\lambda_i\}|} = \frac{1.0}{\max_i |\mathrm{Re}\{\lambda_i\}|} .$$

The CPU-cost to execute one simulation run is proportional to the number of steps to be executed:

$$CPU \sim n_{st} = T/dt = \frac{\max_i |\mathrm{Re}\{\lambda_i\}|}{\min_i |\mathrm{Re}\{\lambda_i\}|} = S .$$

The assumption concerning run-length determination, however, is justified for some standard input functions (like the step function) only. What happens if the dynamics of the simulation are introduced through the input function, as this is common use in control systems? In this case, we can no longer assume that the largest time constant of the system has anything to do with the duration of the simulation run. The slowest transient does not influence the behaviour of the numerical integration at all. It constitutes

just a slowly varying signal superposed to the solution. For this reason, it has been proposed in [6.18] to replace the common definition of S by a new definition:

$$S = T * \max_i |\mathrm{Re}\{\lambda_i\}|$$

where T denotes the (user specified) run-length of the simulation. This new definition seems useful for the prediction of the CPU-cost as long as the fastest transients are not introduced through an input function (driving force).

For a linear combined simulation, we may modify this definition as follows:

$$S = (t^*_{next} - t^*_{current}) * \max_i |\mathrm{Re}\{\lambda_i\}| .$$

S is reevaluated after each event time (t*). The eigenvalues are then recomputed since they may have entirely changed due to event handling, and, instead of using the run-length, we multiply by the time span to the next scheduled event.

This definition gives a valid prediction for the CPU-cost involved in the execution of one simulation run. However, S was meant for a different purpose. We would like to utilize S as an indicator to determine which integration algorithm to use for the solution of a given problem, e.g. by some rule like:

IF S < S* THEN use adams ELSE use gear

where S* could be somewhere around 100.0. However, our new definition for S produces a value which is highly depending on the run-length. Our new definition of S will suggest use of the Gear algorithm for eventually any problem, if the run-length is just made sufficiently large. Such a decision would, however, not be in accordance with our (intuitive) understanding of how the algorithm works! In reality, the

run-length has only an influence in that it accounts for the fixed cost involved in getting the algorithm started (as illustrated in Fig. 6.2). This discussion encourages again another definition of S to be given:

$$S = (1.0 - \exp(-c*T))*\underset{i}{\text{Max}}\ |\text{Re}\{\lambda_i\}|$$

where c is a constant still to be determined. This definition, on a first glance, seems to overcome all deficiencies of previous definitions. However, it is not difficult to show that also this new definition has its own drawbacks.

Given the problem:

$$\dot{\underline{x}} = \underline{f}(\underline{x},t) \quad ; \quad \underline{x}(t=0) = \underline{x0} \ .$$

Let us rescale this problem in time by introducing:

$$\tau = \text{const}*t \ .$$

By substituting τ into the state equation, we obtain:

$$\dot{\underline{z}} = \underline{f}(\underline{z},\tau) \quad ; \quad \underline{z}(\tau=0) = \underline{z0} \ .$$

It is evident that, for this new formulation of the problem, the same CPU-cost must result, and that the same integration algorithm must still be appropriate. However, our new definition of S produces two different values for S when applied to the two formulations.

Most references which enlighten the numerical integration of stiff differential equations from an engineering point of view make use of the term "stiffness" without bothering themselves to give a formal definition for what they mean by it, stating that this term is sufficiently well introduced and understood, and, therefore, requires no further explanation (!). However, the true reason is that any definition of the (rather artificially constructed) quantity "stiffness" may be criticized from one point or another.

Mathematical references (like [6.13]), on the other hand, restrict their view to small subclasses of problems (by avoiding utilization of S) for which they can derive suitable integration algorithms usually after many pages of complicated analytical calculations. This approach is certainly honourable, and it is very useful for a deepened understanding of the mechanisms of numerical integration. It is, however, not directly applicable to the problem of coding robust general purpose simulation software.

Let us assume now that one of the definitions given above suits our purpose. This definition will still be restricted to linear stable systems. We have to consider the question whether and how this definition can be expanded to encompass nonlinear systems as well. A possible answer may be that all nonlinear systems behave like linear systems in the neighbourhood of any working region. Therefore, we can restrict our discussion to the linear part of the system:

$$\dot{\underline{x}} = \underline{f}(\underline{x},t) = A*\underline{x} + \underline{f}_2(\underline{x},t)$$

where the Taylor series expansion of $\underline{f}_2(\underline{x},t)$ has no linear component. A is the linear part of the system which is commonly called the Jacobian:

$$A = J = \frac{\partial \underline{f}(\underline{x},t)}{\partial \underline{x}} \ .$$

In the case of nonlinear systems, A will generally be a function of time, and so will all eigenvalues:

$$\lambda_i = \lambda_i(t) \ .$$

That means that also the stiffness (according to any of the previously given definitions) must be considered a function of time:

$$S = S(t) \ .$$

Such a definition of S can be meaningful as long as the Jacobian is stable throughout the integration. It may well be that one particular integration algorithm is suitable during one part of the simulation whereas another integration algorithm is better appropriate during some other time. However, even very simple nonlinear problems (like the Van-der-Pol equation discussed later in this chapter) do not fall into this class since their Jacobians have eigenvalues with a positive real part during some period of time. In such a case, none of the previously given definitions for S makes any sense. It is entirely unclear how numerical integration errors propagate in this case, and which step size or integration algorithm would be most appropriate. In currently available simulation systems, numerical integration techniques are blindly applied to the solution of such problems without notifying the (credulous) user that -- frankly spoken -- we have no idea at all on how to interpret the results which are produced in this way (!).

For this reason, many to date open research problems must be solved until we can automatically select the most appropriate integration algorithm for all practically arising application problems. For the time being, any of the definitions of S may be used, keeping in mind that the approach must be considered rather heuristical. Indeed, experience has shown that an automated selection basing on such a definition of S determines an appropriate integration method for many (but not for all!) problems. As a matter of fact, such an algorithm has much better means to determine the integration method to be used than the (unskilled) simulation user.

VI.3) Adaptive Selection of Integration Algorithms:

In a nonlinear case, the Jacobian will usually be time dependent, and, with it, also its eigenvalues. This has been shown above. Since the classification is, in general, defined for linear systems, it may well be that in a nonlinear case it would be best to assign the integration algorithm dynamically to the problem. For this purpose, one has to recompute the eigenvalues from time to time to find out whether the integration method in use is still appropriate. It seems a good idea to recompute the eigenvalues as soon as the step-size, which is controlled by the integration rule, has changed by an order of magnitude, but not before a minimum time span of maybe 0.01 times the run length has elapsed. This can then be used to obtain an adaptive selection of the appropriate integration scheme.

VI.4) Verification of Simulation with Respect to Modeling:

Let us assume that a valid model has been derived from the physical system under investigation, and let us question what assurance we have that the time responses which we obtain through simulation represent the (valid) model correctly.

For a variable-step integration method being used, we normally trust in the step-size control mechanism which is equivalent to confiding in the error estimation procedure. This will usually be justified as long as the local error which we control can be used as a valid estimate for the global error in which we are interested.

Experience has shown that local errors will not usually accumulate as long as the system is numerically stable. In a nonlinear case, it may, however, happen that some eigenvalues "walk" into the right half-plane for a short period

of simulated time. Let us consider, as an example, the well known Van-der-Pol equation. Fig. 6.8 shows how the eigenvalues "walk around" during one limit cycle.

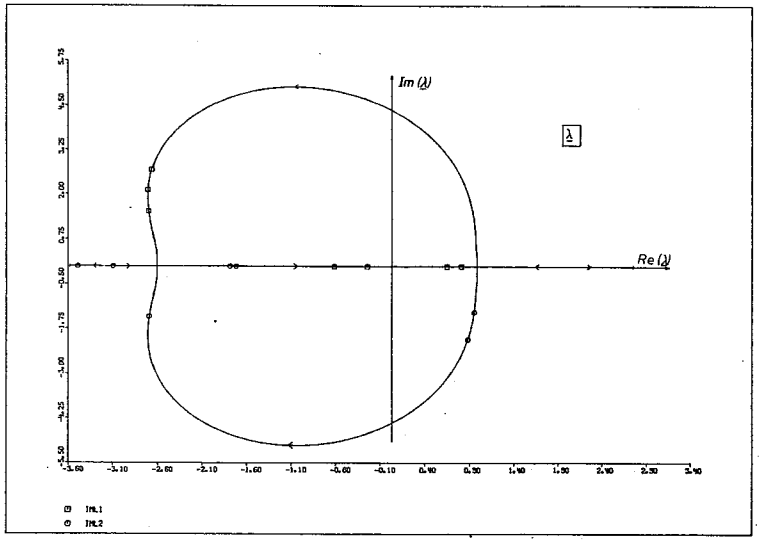

Fig. 6.8: Eigenvalue movement of the Van-der-Pol equation during one limit cycle

As can be seen, the system becomes periodically unstable. During such time intervals, errors will accumulate, and, consequently, we must be careful in the interpretation of the obtained results. This fact should be reported by the software to the user.

For this purpose, we define another feature (stability). A variable STAB is set equal to zero when all eigenvalues lie in the left half-plane and is equal to one as soon as at least one of the eigenvalues moves into the right half-plane.

$$STAB = \begin{cases} 0.0 : Re\{\lambda_i\}<0.0 \ ; \ i=1,\ldots,n \\ 1.0 : \text{otherwise} \end{cases}$$

We now collect statistics on STAB as for time-persistent variables. In this way we obtain the integral of STAB over time divided by the run length:

$$FF5 = \frac{1.0}{|TTFIN-TTBEG|} * \int_{TTBEG}^{TTFIN} (STAB)\,dt$$

The fifth feature (FF5) is a real number between 0.0 and 1.0. If it is close to 0.0, the results obtained by simulation have a good chance to be reliable. If it is close to 1.0, the obtained results are most probably nonsense, and they must be cautiously verified.

VI.5) Validation of the Model with Respect to the System under Investigation:

Another non-trivial question is whether a model, for given experimental conditions, properly represents the system under investigation. Some possible answers to this question have already been mentioned (like dimensional analysis). In this section we want to show that the eigenvalue distribution can also help to answer this question to some extent.

It has been shown in [6.9] that only those eigenvalues of a matrix can be properly computed which fulfil the following inequality:

$$|\lambda_i| > \sigma_1 * \varepsilon^{(1/n)}$$

where σ_1 is the largest singular value of the matrix, ε is the machine resolution (e.g. $\sim 10^{-14}$ on a CDC 6000 series installation), and n is equal to the order of the model. For higher order models $\varepsilon^{1/n}$ approaches 1.0 and hardly any eigenvalues will then be properly computable. Smaller eigenvalues can take any value and small modifications of the elements of the matrix can place them almost anywhere within the band of incertainty.

If we now assume that the matrix under investigation is a Jacobian of a state space description for a real physical system, then the elements of the Jacobian are extracted from measurements, and cannot be computed more accurately than $\hat{\varepsilon}$, which is a relative accuracy of measurement. We, therefore, must assume that, within that relative accuracy $\hat{\varepsilon}$, the elements of the matrix can take any values. In this case, we must also assume that eigenvalues of the Jacobian which do not fulfil the more stringent inequality:

$$|\lambda_i| > \sigma_1 * \hat{\varepsilon}^{(1/n)}$$

can take any value within that broader band, although they can be much more accurately computed as soon as any particular values have been assigned to all elements of the Jacobian. This means that as soon as there exist eigenvalues for which the second (more stringent) inequality does not hold, small variations in the systems parameters which lie within the inaccuracy of the measurement can make the model non-stiff or stiff or even unstable (according to the original definition of stiffness). Physically seen, these eigenvalues correspond to merely constant modes which could as well be eliminated from the equation set resulting in a model reduction. Numerically seen, these eigenvalues can lead to accumulation of errors so that these modes can drift away over a longer span of simulated time, again resulting in incorrect simulation trajectories.

Together with the eigenvalues, we compute the following quantity:

$$BORD = \sigma_1 * \hat{\varepsilon}^{(1/n)}$$

and the number k indicating those eigenvalues whose absolute value is smaller than BORD:

$$k = \sum_{i=1}^{n} (j_i) \; ; \; j_i = \begin{cases} 0 : |\lambda_i| \geq BORD \\ 1 : |\lambda_i| < BORD \end{cases}$$

k represents an integer between 0 and n.

We now collect statistics on the quantity (k/n) as for time-persistent variables, and obtain a sixth feature:

$$FF6 = \frac{1.0}{|TTFIN-TTBEG|} * \int_{TTBEG}^{TTFIN} (k/n)\,dt$$

Also the sixth feature (FF6) is a real number between 0.0 and 1.0. If it is close to 0.0, the model has some chance to be valid. If it is close to 1.0, the model is most probably invalid, and it should be further investigated. Most probably, information has been taken into account for the construction of the model which cannot be validated with the available measurements. In this case, one should either try to simplify the model (by model reduction techniques) or use another measurement technique to obtain better data.

Evaluation of features FF5 and FF6 requires computation of the eigenvalues of the Jacobian once per integration step. Since this can be expensive, it should not be done auto-

matically, but the user must have a switch at his disposal to turn computation on and off. In this way he can use these features during the development of a new model, whereas he can turn computation off during production runs.

Since, even in a nonlinear case, the eigenvalues are most likely to change only slowly with time (except during events), it would be most appropriate to use an iterative method for the computation of the eigenvalues which takes advantage of the knowledge that the eigenvalues are expected to lie in the neighbourhood of some starting values (which are the true solutions at the previous computational instance). Unfortunately, EISPACK does not provide such methods, and no such methods are known to the author.

VI.6) Determination of Critical States:

In section VI.5 we have discussed the case where single eigenvalues were situated close to the imaginary axis, and we have seen that in such a case it may be possible to reduce the order of the model.

It is, however, as interesting to discuss the opposite case where single eigenvalues are located in the λ-plane far to the left. We call these modes the "critical states" of the system. Very often one is not really interested in these fast transients. In such a case one could eliminate these modes from the equation set. If the fast transients are important one could at least try to utilize special integration techniques (like using singular perturbations) to expedite integration.

One can, of course, again compute the eigenvalue distribution for the solution of this problem. However, it is not always easy to see which state equations are responsible for such an eigenvalue. For this reason we recommend the following procedure.

We reserve an integer array of length n $i_n(n)$ which is initialized to zero. Each time an integration step has to be rejected due to accuracy requirements not being met, we increment each element of the array $i_n(k)$ for which the accuracy is not met by one. This implies, of course, that the local truncation error is estimated for all state variables independently. At the end of the simulation run we divide each element of the array by the total number of rejected integration steps and obtain in this way another set of n real numbers between 0.0 and 1.0. Elements with the largest value indicate critical states.

VI.7) Robust Methods for the Numerical Solution of PDE Problems:

VI.7.1) Statement of the Problem:

So far, we have discussed methods to improve the robustness of software for the solution of ODE problems, and we have shown that with the help of ideas commonly used in pattern recognition, we can improve the robustness of this kind of software remarkably without any reduction in generality.

As we have discussed in chapter II of this thesis, the PDE case is much more difficult to handle, since there exists a far stronger link between the problems to be solved on one hand, and the algorithms to be used on the other hand. The solution of slightly different types of PDE problems often requires the development of separate and different numerical algorithms. The development of adequate numerical algorithms is by no means a trivial problem, and the average user has hardly any chance to succeed in this.

For this reason, the demand for robust PDE software is even

more urgent than that for robust ODE software.

Since the ODE problem has been thoroughly considered, modern sophisticated PDE software tries to take advantage of this knowledge by transforming the PDE problem into an equivalent ODE problem which is easier to solve. This leads to the method-of-lines approach as has been discussed in section V.3.4 of this thesis.

In the old days, each person who wanted to solve PDE problems numerically on a digital computer had to become a specialist of numerical mathematics, or had to consult such a specialist to let him propose the appropriate algorithm. Application of the method-of-lines makes the formulation of PDE problems so simple that any person is able to use it and to produce results. However, this is also the most serious fault of the method, since the above problem is not at all solved. It is just hidden to the user and nicely packed up! The user is provided with a menu of different integration procedures, and different differentiation schemes, etc., out of which he must decide:

a) which is the best suited integration algorithm for the numerical integration over time,

b) which is the appropriate differentiation scheme for computation of the spatial derivatives and,

c) which is the optimal grid-width to be used for the discretization of the spatial variables.

That is, the user must select among different kinds of algorithms communicating with each other. Here the situation is even worse than in the ODE case, where the user is requested to choose an integration algorithm, since here there exists an almost infinite choice of combinations of integration, differentiation and grid-width selection. A strategy of

"blind search" to detect an optimal combination is, therefore, rather hopeless.

The situation is even more tragic, since a bad choice does not necessarily result in an error indication. On the contrary, there will often results be produced which look very promising (for all kinds of time responses it is usually possible to find, a posteriori, a theory to explain them!), but which are, nevertheless, entirely incorrect. The reason for this is that there can be no guarantee that any particular combination of integration, differentiation, and state space description will lead to a finite difference scheme which is consistent, convergent, and stable. Stability will usually be taken care off by the step-size control of the numerical integration, but resulting inconsistencies or divergence will not necessarily be detected. This means that the user normally obtains "correct" time responses with respect to the formulated ODE problem, but it is not guaranteed that

a) the resulting difference equation properly approximates the original differential equation (consistency), and that

b) the obtained time responses at discrete points smoothly approximate the continuous time responses which we are looking for (convergence).

We want to discuss, in the following, what can be done to improve the robustness of PDE software.

VI.7.2) Grid-Width Control:

A first measure which we can consider to improve robustness is to establish a grid-width control in space which is similar to the step-size control in time.

We can formulate the following error estimation procedure: We compute the spatial derivatives once with k and once with (2k-1) grid points, and compare the results. This works well, but has a disadvantage which we are going to show immediately.

Let us consider again the heat transfer equation

$$\frac{\partial u}{\partial t} = \sigma * \frac{\partial^2 u}{\partial x^2}$$

as an example. We will now use a fixed-step Euler integration

$$u(x,t+dt) = u(x,t) + dt * \frac{\partial u}{\partial t}(x,t)$$

in time, and a 3-point central formula

$$\frac{\partial^2 u}{\partial x^2}(x,t) = \frac{u(x+dx,t) - 2*u(x,t) + u(x-dx,t)}{dx}$$

in space.

Let us write for the discretized system:

u(x,t) as u[j,n]
u(x,t+dt) as u[j,n+1]
u(x+dx,t) as u[j+1,n]

etc..

Combination of the above three formulae constitutes the following difference scheme:

$$u[j,n+1] = (1-2*\lambda)*u[j,n] + \lambda*(u[j+1,n] + u[j-1,n])$$

where:

$$\lambda = \sigma * dt/(dx^2)$$

This is a well known difference scheme for which has been shown (e.g. in [6.17, chapter 12 by H.B.Keller]) that it is stable for

$$\lambda \leq 0.5$$

only.

Let us assume that the accuracy requirements for the problem are not stringent so that we can integrate with the maximum value of dt which grants a stable difference scheme

$$dt = (dx)^2/2*\lambda \ .$$

Computation of the problem for k grid points is said to cost x units of CPU-time. If we now recompute the problem for (2k-1) grid points, we find a new maximum grid width

$$dx' = dx/2 \ .$$

Accordingly we have the double quantity of ODE's which have to be integrated over time using one fourth of its former step-size

$$dt' = dt/4 \ .$$

This time the required amount of CPU-time, consequently, is 8*x units.

Although the error indication which is obtained by comparing the two solutions is normally very good, the proposed method is useless, since we cannot tolerate paying 8 times as much, just to obtain a better software security.

A grid-width control of the spatial discretization (in analogy to the step-size control of the numerical integration) is, therefore, obviously not attractive since reducing the grid-width goes together with an intolerable increase in execution time.

VI.7.3) Order Control:

Another possibility would be to let the number of grid points remain unchanged, and to compare, instead, the spatial derivatives computed from different differentiation formulae (e.g. once from a 3-point central formula and once from a 7-point central formula) for error estimation. This does not result in unduly high computational costs, but, so far, we were unable to find an error indicator which would work well for all kinds of application problems. More research needs to be devoted to this topic.

Let us assume now that a good error indicator has been found. Would it then be feasible to use this error indicator for an automated order control? Unfortunately, we have not much freedom in selecting different differentiation formulae since increasing the order is equivalent to involving more neighbouring grid points in the evaluation of the spatial derivatives. Together with this, those boundary regions where biased, instead of central, formulae must be used will also grow which is not very favourable either.

From all this it can be concluded that an indicator to improve the robustness would be very useful if not too much additional CPU-time is needed for its computation. An

"adaptive" algorithm, as for the numerical integration over time, seems infeasible at the moment.

VI.7.4) Other Methods:

Coming back to the heat transfer problem, one could find that the appropriate way to solve this problem would be to replace the Euler integration by a backward Euler algorithm:

$$u(x,t+dt) = u(x,t) + dt*\frac{\partial u}{\partial t}(x,t+dt)$$

This results in an implicit difference scheme:

$$(1+2*\lambda)*u[j,n+1] = \lambda*(u[j-1,n+1] + u[j+1,n+1]) + u[j,n]$$

This scheme is also well known. It is stable independent of the actual value of λ [6.17].

This is, however, a specific remedy to this particular application problem, and we do not see any way to automate this analysis.

VI.7.5) Conclusions:

As can be seen, we are now exactly where we started in the beginning, namely, that if problems arise, a specialist in numerical mathematics must be consulted to lead us out of them. The method-of-lines approach has not given any final answers, it has just helped to make the formulation of problems easier, and it provides for a large variety of different algorithms for numerical integration and differentiation which can be modularly combined to form almost any imaginable finite difference scheme for the solution of para-

bolic and hyperbolic PDE problems. Furthermore, the step-size control of the numerical integration over time improves the robustness of the algorithm. Experimentation is simplified, and, since the algorithms are precoded, the user can primarily concentrate on problem-specific rather than on procedure-specific questions. The time spent in the formulation of any particular application problem as well as the probability that the algorithm contains coding errors are both remarkably reduced.

For these reasons, the method-of-lines approach to the solution of PDE problems is a very powerful tool which helps to increase the software robustness, although it is no cure-all, as has been demonstrated. Robustness of modern PDE software can be further improved by providing a better error indication concerning error accumulation resulting from the spatial discretization. For this purpose, different differentiation formulae are to be compared with each other where optimal pairs are still to be evaluated. Grid-width controlled or order controlled adaptive algorithms comparable to the ODE case seem, however, infeasible.

References:

[6.1] A.V.Aho, Ullman J.D.: (1972/73) "The Theory of Parsing, Translating and Compiling -- Volume I: Parsing". Prentice-Hall, Series in Automatic Computation.

[6.2] R.Alexander: (1977) "Diagonally Implicit Runge-Kutta Methods for Stiff O.D.E.'s". SIAM Journal on Numerical Analysis, vol. 14, no. 6 : December 1977; pp. 1006 - 1021.

[6.3] A.P.Bongulielmi, Cellier F.E.: (1979) "On the Usefulness of Deterministic Grammars for Simulation Languages". Proc. of the Sorrento Workshop on International Standardization of Simulation Languages (SWISSL), Sorrento, Italy.

[6.4] K.J.Bucher: (1977) "Automatisches Zeichnen von Syntaxdiagrammen, welche in spezieller Backus-Naur Form gegeben sind: Benuetzeranleitung". To be ordered from: Institute for Informatics, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[6.5] F.E.Cellier, Moebius P.J.: (1979) "Towards Robust General Purpose Simulation Software". Proc. of the ACM SIGNUM Symposium on Numerical Ordinary Differential Equations. April 3-5, 1979, University of Illinois at Urbana-Champaign; pp. 18.1 - 18.5.

[6.6] M.S.Elzas: (1978) "What is Needed for Robust Simulation". Proc. of the Symposium on Modeling and Simulation Methodology, Rehovot, Israel. Published by North-Holland Publishing Company (Editors: B.P.Zeigler, M.S.Elzas, G.J.Klir, T.I.Oren); pp. 57 - 91.

[6.7] B.S.Garbow, Boyle J.M., Dongarra J.J., Moler C.B.: (1977) "Matrix Eigensystems Routines - EISPACK Guide Extension". Springer Verlag, Lecture Notes in Computer Science, vol. 51.

[6.8] C.W.Gear: (1971) "Numerical Initial Value Problems in Ordinary Differential Equations". Prentice Hall, Series in Automatic Computation.

[6.9] G.H.Golub, Wilkinson J.H.: (1976) "Ill-Conditioned Eigensystems and the Computation of the Jordan Canonical Form". SIAM Review, vol. 18, no. 4 : October 1976; pp. 578 - 619.

[6.10] P.Henrici: (1964) "Elements of Numerical Analysis". John Wiley.

[6.11] P.Henrici: (1970) "Upper Bounds for the Abscissa of Stability of a Stable Polynomial". SIAM Journal on Numerical Analysis, vol. 7, no. 4 : December 1970; pp. 538 - 544.

[6.12] D.Kahaner: (1977) "A New Implementation of the Gear Algorithm for Stiff Systems". Unpublished private communication. For further detail contact: Dr. David Kahaner, University of California, Los Alamos Scientific Research Laboratory, Contract W-7405-ENG-36, P.O.Box 1663, Los Alamos NM 87545, U.S.A..

[6.13] H.O.Kreiss: (1978) "Difference Methods for Stiff Ordinary Differential Equations". SIAM Journal on Numerical Analysis, Vol. 15, No. 1, February 1978; pp. 21 - 58.

[6.14] J.D.Lambert: (1973) "Computational Methods in Ordinary Differential Equations". John Wiley.

[6.15] B.Lindberg: (1973) "IMPEX 2 - A Procedure for Solution of Systems of Stiff Ordinary Differential Equations". Report TRITA-NA-7303 . To be ordered from: The Royal Institute of Technology, Stockholm, Sweden.

[6.16] M.A.R.Mansour, Jury E.I., Chapparo L.F.: (1979) "Estimation of the Margin of Stability for Linear Continuous and Discrete Systems". International Journal of Control, Vol. 30, No. 1; pp. 49 - 69.

[6.17] A.Ralston, Wilf H.S.: (1960) "Mathematical Methods for Digital Computers". John Wiley.

[6.18] D.F.Rufer: (1977) "Numerik zur Systemtheorie". Lecture Notes. To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[6.19] J.Stoer, Bulirsch R.: (1973) "Einfuehrung in die Numerische Mathematik II". Heidelberger Taschenbuecher, Bd. 114. Springer Verlag.

[6.20] N.Wirth: (1973) "Systematical Programming: An Introduction". Prentice-Hall, Series in Automatic Computation.
or:
N.Wirth: (1972) "Systematisches Programmieren". Teubner Studienbuecher, Informatik, Vol. 17.

[6.21] (1967) "The SCi Continuous System Simulation Language (CSSL)". Simulation, vol. 9, no. 6 : December 1967; pp. 281 - 303.

VII) ASPECTS OF INFORMATION PROCESSING:

VII.1) Statement of the Problem:

So far, we have discussed the numerical behaviour of a run-time system able to perform combined simulation. Now the question remains: What is the easiest and most convenient way for the user to formulate combined problems to the computer so that the computer will be able to produce properly executable run-time code? For this purpose, we will have to identify the structural elements of combined simulation languages.

VII.2) The Elements of the Language:

A combined simulation language will primarily consist of the well known elements of continuous and discrete simulation languages. There are few additional elements required to weld these two subsystems together.

A) The state-event and its associated state-conditions:

The only essential new element is the state-condition describing conditions of the continuous subsystem status required to branch to the discrete subsystem. The associated state-event, whose execution is triggered by a state-condition, is an event as any other. A typical situation is illustrated in the following:

> When the angular velocity of a DC-motor crosses a threshold of 1500 RPM in the positive direction, the motor has to be loaded.

The crossing of the threshold by the velocity is a

typical state-condition, whereas loading the motor is the associated state-event.

The state-condition, in the above problem, could be coded using a 'CONDIT'-statement in the continuous subsystem:

```
CONTINUOUS
  ...
  CONDIT EV1: OMEGA CROSSES 1500.0 POS
              TOL=1.0E-3 END;
```

and the reaction to this could then be coded by an event description in the discrete subsystem:

```
DISCRETE
  EVENTS
    EV1: TL := 200.0 END;
    ...
```

(the torque load (TL) is to be reset to 200.0). The CONDIT-statement is similar to a CSMP FINISH condition, except that the time of the crossing is iterated until a prespecified tolerance is met (TOL=1.0E-3), and in that the simulation run is not terminated, but control is handed over to the discrete simulation system. After event handling, as described by the discrete subsystem (DISCRETE), control is returned to the continuous subsystem (CONTINUOUS) where the new value of TL will be used somewhere in one or several equations on the right hand side of the equal sign.

B) Operations of the continuous subsystem on the discrete subsystem:

There are none.

C) Operations of the discrete subsystem on the continuous subsystem:

It is most commonly found that not only parameters of the continuous subsystem (as the torque load TL above) change their values at event times but that some of the equations are replaced by others. This situation can be taken care of by the following language elements:

a) The "one-out-of-n" situation:

There are n possible "models" out of which one is always active. This situation can best be expressed by a CASE-statement:

```
CASE NMOD OF
```

where NMOD is an integer number pointing to the currently active model. This language element is used in general to describe n different functional ways of behaviour of one model component, e.g. n continuous branches of a discontinuous (but piece-wise continuous) functional block.

b) The "k-out-of-n" situation:

Another frequently found situation is illustrated by the following example:

> There are n cars in a system, out of which k are moving around and (n-k) are parked somewhere.

This situation can be represented by the following syntactical construct:


```
FOR I:=1 TO N DO
  IF CAR[I] THEN
```

where CAR is a boolean array with the values "true" for cars moving around and "false" for parked cars.

For n = 1 this case degenerates to a simple IF-clause.

c) Example:

Let us consider a mechanical system with a dry friction torque (TFR) modeled somewhere in the system. The functional relationship which models the friction torque (TFR) as a function of the angular velocity (OMEGA) and of the driving torque (T) can be shown by the following graph:

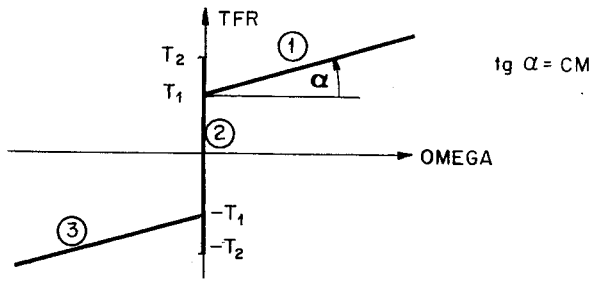

Fig. 7.1: Dry friction torque versus angular velocity

In this example we face the typical "one-out-of-n" situation, where n = 3 are the three continuous branches of the discontinuous TFR-function. Each of them is represented by a different equation and by a

different set of state-conditions.

This situation can be coded as shown in Fig. 7.2. Using this formalism for describing a combined system, the resulting description is not much more complicated than using a normal CSSL-type language, but it allows the preprocessor to produce properly executable run-time code.


```
SYSTEM
  CONTINUOUS
    ...
    ...
    MODEL DRYFRICTION (TFR <- T, OMEGA);
    (* COMMENT: <- SYMBOLIZES A LEFT ARROW AND IS USED
       TO SEPARATE INPUT FROM OUTPUT VARIABLE LISTS *)
      CASE NL OF
        1:  TFR = T1 + CM*OMEGA;
            CONDIT MOD2: OMEGA CROSSES 0.0 NEG TOL=1.0E-3
            END
          END;
        2:  TFR = T;
            CONDIT MOD1: T CROSSES T2 POS TOL=1.0E-3 END;
            CONDIT MOD3: T CROSSES -T2 NEG TOL=1.0E-3 END
          END;
        3:  TFR = -T1 + CM*OMEGA;
            CONDIT MOD2: OMEGA CROSSES 0.0 POS TOL=1.0E-3
            END
          END
        (* DRY FRICTION *)
    ...
  END (* CONTINUOUS SUBSYSTEM *)
  DISCRETE
    EVENTS
      MOD1:  NL := 1 END;
      MOD2:  NL := 2; OMEGA := 0.0 END;
      MOD3:  NL := 3 END
    END (* STATE-EVENTS DESCRIPTION *)
  END (* DISCRETE SUBSYSTEM *)
END (* SYSTEM DESCRIPTION *)
```

Fig. 7.2: Combined description of a dry friction torque

The mode selector switch NL determines the three continuous branches of the discontinuous dry friction torque. Continuous simulation, as described by the CONTINUOUS block, goes on until one of the state-conditions associated with the currently active mode is met. At this moment, integration is interrupted, and control is transferred to the discrete subsystem modeled by the DISCRETE block. The associated state-event is executed which basically changes the selector switch NL, in the above example, to point to another mode. Now, the control is transferred back to the continuous subsystem, and the integration algorithm is restarted to integrate the model over the next inter-event time span.

VII.3) Requirements of the Language:

The following section shall describe the requirements which are to be met by a good language for combined system simulation. These can be summarized in the following 10 points and will be discussed thereafter.

a) The language should provide for flexible structures.

b) It should be extendable (open-ended operator set).

c) The language should be transparent, and the user should have access to its primitives.

d) It should provide structures which allow the user to achieve, as completely as possible, a "one-to-one" correspondence between model and system.

e) Both syntax and semantics of the language should be easy to learn and to remember.

f) The language should contain as few elements as possible but as many as are required.

g) Models should be codable by as few elements as possible.

h) The preprocessor should contain provisions for faithfully detecting coding errors.

i) The language must contain all elements required to enable the preprocessor to produce numerically well-conditioned run-time code.

j) The language should be "robust" in the sense outlined in section VI.1.

Some of these requirements are contradictory. If we want to enable the preprocessor to detect as many errors as possible, introduced by the user in formulating his model, the language must contain some redundancy. This certainly competes with the wish to have user's programs as short as possible.

VII.3.1) Flexible Structures:

Two different aspects can be mentioned in this context.

a) The language should be generally applicable. It should contain elements for proper formulation of all imaginable problems. For this purpose, the language must be constructed in such a way as to let its "atoms" (undividable building blocks) be basic primitives. All more complex language elements must be generated out of these primitives. Basic primitives are the integral operator of the continuous subsystem and the event operator of the discrete subsystem.

In particular, any "master scheme" offered by the language to simplify the coding of special situations will be restrictive and must, consequently, reduce the flexibility of the software. Typical examples of master schemes, as they are frequently found in simulation software, are predefined versions of PDE's for which the user must simply provide appropriate coefficients to formulate his particular model.

b) The language should provide facilities for modular programming. One should be able to declare a part of the system's description as an autonomous submodel which communicates with its environment through a programmable interface (usually a list of formal parameters). This can e.g. be realized by a "MODEL"-element as shown in Fig. 7.2 above. In the context of its environment, a MODEL behaves in the same way as the PROCEDure construct proposed in the CSSL definition. It is a sandwich statement which is regrouped as a whole within the other parallel statements. As in the case of the PROCEDure construct, the formal parameters must be separated in lists of inputs and outputs of the MODEL. They are required only to enable proper sorting of the MODEL with respect to its environment. Global constants and state variables need not be listed, and can be accessed implicitly. Contrary to the PROCEDure construct, the statements of the MODEL are again parallel code, that is, they are sorted among each other. As a matter of fact, all modeling elements apply to a MODEL in the same way as to the whole continuous subsystem (CONTINUOUS). It is, in particular, possible to define MODELs in a hierarchical manner. This language element is not identical with the CSSL-type MACRO either, as shall subsequently be shown. As a matter of fact, MODEL is a new language element which is not accessible in today's CSSL-type languages, and which is most useful for structuring problems, especially when a team of several

scientists is involved in modeling a complex system jointly.

VII.3.2) Extendability:

This requirement also has two different aspects.

A) The user of the software should be able to extend the available simulation operators by his own problem-specific ones (open-ended operator set). Such language extensions can take place on four different levels.

a) On a very basic level, the language operators can be extended by coding (e.g. FORTRAN) SUBROUTINEs. Also the CSSL-type PROCEDure construct belongs to this category, and can be treated in the same manner.

b) On a second level, the language operators can be extended by formulating CONTINUOUS and DISCRETE PROCESSes (equivalent to the CLASS concept offered in SIMULA-67 [7.6]). These PROCESSes can also be preprocessed into subprograms. In the definition of such PROCESSes, the user must include all interacting variables and constants as formal parameters of the PROCESS. Variables not included will be local to the PROCESS when it is reused. A precompiled PROCESS can be called in by declaring it to be an EXTERNAL PROCESS. This is very similar to calling EXTERNAL SUBROUTINEs. However, to allow for proper bookkeeping, the user must, in addition, specify how many state variables (for ODE's and difference equations) and how many history functions [7.23] (requiring a unique identifier each) are internally used in the PROCESS definition body.

A CONTINUOUS PROCESS is a natural extension to the previously discussed MODEL concept. Again, parallel

statements are internally sorted whereas the differential equations defined in the CONTINUOUS PROCESS are never intermixed with others. CONTINUOUS PROCESS bodies, however, are precompiled into SUBROUTINEs, and calls to CONTINUOUS PROCESSes are precompiled into calls to SUBROUTINEs, whereas MODELs are defined where they are used, and the resulting code is directly inserted as in the case of a CSSL-type PROCEDure. For this reason, all communicating variables of CONTINUOUS PROCESSes must be included in the list of the formal parameters (local variable concept). The parameter list of MODELs is required for sorting purposes only. Constants which are used in a MODEL, but which are defined outside of it, need not necessarily be included into the list of the formal parameters of that MODEL (global variable concept).

c) On a third level, the language should provide for a MACRO facility. Formally this looks very similar to the previously presented MODEL and PROCESS facilities. It is, however, treated differently by the compiler. All MACRO calls are first replaced by their MACRO definition bodies, before any further preprocessing (like sorting) takes place. In this way, the statements which form a MACRO definition can be spread throughout the system's description, once an executable sequence of statements has been found. Consequently, MACROs must always be kept in source form in a "symbolic" library. The MACRO facility is needed since it is often not possible to avoid mixing equations from different MACROs to obtain an executable sequence of statements. This has been shown in [7.5]. Thus, the MACRO construct grants a higher degree of modularity compared to the MODEL and PROCESS constructs, but it requires each MACRO definition body to be preprocessed together

with the environment in which it is used.

Since the MACRO replacement must preceed all further preprocessing activities, the MACRO feature need not form an intrinsic part of the language definition. It can as well be taken care of by a separate MACRO handler which is called prior to preprocessing. In this way, one can be more generous in the capabilities offered by the MACRO definition language (like offering interpretative MACRO handling) while saving core memory requirements. The additionally required computation costs are comparatively small [7.2].

d) On a fourth level, the language can provide a programmable topological input description combining the advantages of a network formulation with those of an equation oriented language. This can actually be thought of as an extension to the previously discussed MACRO construct. When coding a MACRO, the modeler must declare which are its inputs and which are its outputs. This has some disadvantages, as will be illustrated in the following example.

Let us consider a small electrical network as depicted in Fig. 7.3. The RC-circuit is to be modeled by a MACRO.

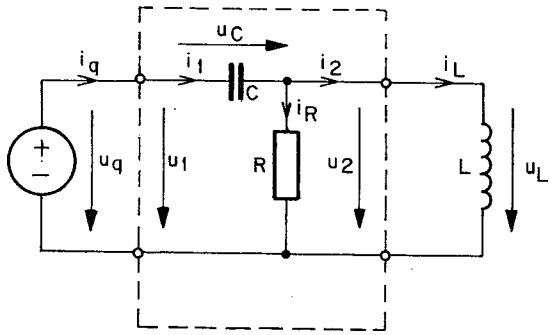

Fig. 7.3: RLC-network with voltage source

Under the assumption that all differential equations are solved for state derivatives (which is reasonable, since integration is numerically much better conditioned than differentiation), there exists only one valid formulation for the required MACRO. This is coded in Fig. 7.4.


```
MACRO RC1 (U2, I1 <- U1, I2, R, C);
  MACVAR
    STATE UC;
    ALGEBR IR;
  MACCONTIN
    UC' = I1/C;
    I1 = I2 + IR;
    IR = U2/R;
    U2 = U1 - UC
  MACEND (* CONTINUOUS *)
MACEND (* RC1 *);
...
CONTINUOUS
  ...
  UQ = f(TIME);
  RC1 (UL, IQ <- UQ, IL, R, C);
  IL' = UL/L;
```

Fig. 7.4: Model of a RLC-network with voltage source

UQ must be specified as an input to the MACRO since it is an externally computed control signal. Also, IL must be an input to the MACRO since it is a state variable of the system which, consequently, cannot be a computed quantity.

Let us now replace the voltage source by a current source as depicted in Fig. 7.5.

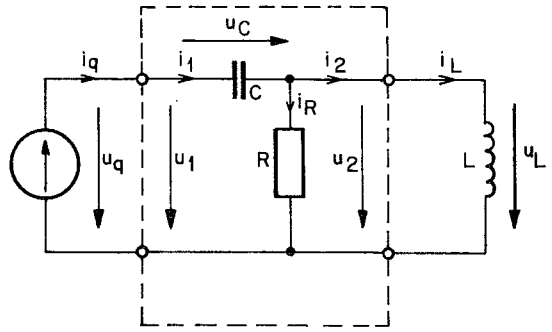

Fig. 7.5: RLC-network with current source

Again, just one valid model can be specified for the MACRO which is depicted in Fig. 7.6.


```
MACRO RC2 (U1, U2 <- I1, I2, R, C);
  MACVAR
    STATE UC;
    ALGEBR IR;
  MACCONTIN
    UC' = I1/C;
    IR = I1 - I2;
    U2 = R*IR;
    U1 = U2 + UC
  MACEND (* CONTINUOUS *)
MACEND (* RC2 *);
...
CONTINUOUS
  ...
  IQ = f(TIME);
  RC2 (UQ, UL <- IQ, IL, R, C);
  IL' = UL/L;
```

Fig. 7.6: Model of a RLC-network with current source

This time, the source current (IQ) must be an input to the MACRO since it is an externally computed control signal.

As one can see, two different MACROs are needed to describe one and the same module. In both MACROs the same equations are represented, but rearranged to meet the demands of the required inputs and outputs. This simple example illustrates that the MACRO element is not really modular either. For this reason, we define a new language element, which we call a "MODULE", as shown in Fig. 7.7a.

```
MODULE RC (U1, U2, I1, I2, R, C);
  VAR
    STATE UC;
    ALGEBR IR;
  CONTINUOUS
    UC' = IC/C;
    I1 = IR + I2;
    U2 = R*IR;
    U2 = U1 - UC
  END (* CONTINUOUS *)
END (* RC *);
```

Fig. 7.7a: MODULE for a RC-circuit

This MODULE can be used in both networks. If a voltage source feeds the RC-circuit, the MODULE can be called as shown in Fig. 7.7b.

```
CONTINUOUS
  ...
  UQ = f(TIME);
  RC (UL=U2, IQ=I1 <- UQ=U1, IL=I2, R, C);
  IL' = UL/L;
```

Fig. 7.7b: Model of a RLC-network with voltage source

If the RC-circuit is fed by a current source, the same MODULE can be used as shown in Fig. 7.7c.


```
CONTINUOUS
...
  IQ = f(TIME);
  RC (UQ=U1, UL=U2 <- IQ=I1, IL=I2, R, C);
  IL' = UL/L;
```

Fig. 7.7c: Model of a RLC-network with current source

In a MODULE, equations may be solved for any variable, as long as their number is correct (problem neither under- nor overspecified), and as long as no contradictory assumptions are made. The same variable may appear several times to the left of the equal sign as U2 in Fig. 7.7a. Formal parameters of a MODULE definition need no longer be separated into inputs and outputs. Only upon usage of a MODULE, one has to specify which are its inputs and which are its outputs. The logical mapping of actual to formal parameters is no longer implicit, but is specified by the names of the formal parameters as expressed in the MODULE definition header. UL=U2, for example, specifies that the actual parameter UL is to replace the formal parameter U2 of the MODULE definition header.

When defining more complex networks in this way, it is almost unavoidable that algebraic loops result. These are groups of algebraic equations which cannot be sorted into any executable sequence. The simplest example of such a loop is:

$$x = fct_1(y)$$
$$y = fct_2(x) \ .$$

This set of equations constitutes an algebraic loop if neither fct_1 nor fct_2 are memory functions.

Memory functions are functions whose outputs depend only on values of inputs at past instances of time. Typical examples of memory functions are the DELAY-function and any explicit numerical integration scheme. Memory functions are, thus, loop breakers.

True algebraic loops can be solved numerically at run-time by applying an implicit loop solver to the set of unsortable equations as it is commonly offered in many CSSL-type simulation systems (e.g. in CSMP-III). This can be specified by coding the set of equations as:

```
IMPLICIT (X, Y <- ; MAX = 30);
  TOLERANCE  X = 0.01, Y = 1.0E-3;
  START  Y = 1.0;
  BEGIN
    X = FCT1 (Y);
    Y = FCT2 (X)
END  (* IMPLICIT LOOP *);
```

which will be translated into an appropriate iteration scheme by which the affected equations are iterated to their correct solution during each evaluation of the state derivatives, that is once or even several times per integration step. A tolerance can be specified for each of the implicit variables. Starting values may be supplied for the "inputs" of the implicit loop body. Finally, the number of iterations may be limited. Even, if the previously iterated values are used as starting vector for the next iteration (which should always be done), this solution can become extremely expensive. It is, therefore, to be avoided where ever possible.

Another feasible way would be to modify equations

analytically at compile time to form an executable set of modified statements. If, for instance, the two equations have the form:

$$x = 3y - 5u$$
$$y = -2x + 3u$$

the compiler can generate out of this specification the statements:

```
ZZ0001 = U/7.0;
X = 4.0*ZZ0001;
Y = 13.0*ZZ0001;
```

which will no longer require any iteration to be carried out. This can be achieved by generalizing the sorting algorithm of currently available simulation compilers. The required methods can be taken from structural algebra, at least for sets of equations which are linear in the implicit variables. This has been shown in [7.7,7.8].

Only in cases where the compiler is unable to do the required analytical manipulations of the model structure, it should generate a call to the implicit loop solver by notifying the user that execution of this program may be costly.

The MODULE definition language is more general than the MACRO definition language in two senses.

1) Besides replacement of parameters, it uses structural algebra and formulae manipulation to reorganize the statements.

2) A MODULE definition may contain an INITIAL block, a TERMINAL block and also a DISCRETE

block in addition to the CONTINUOUS block. When the MODULE is called from within the continuous subsystem (CONTINUOUS), these blocks will automatically be transferred to their correct locations.

The MODULE definition language is less general than the MACRO definition language in that it does not allow any interpretative execution, as we want to allow it for MACROs. (This feature will be illustrated at an example in Fig. 7.13.).

The MODULE definition language is similar to the MACRO definition language in that all involved activities must be performed prior to any further preprocessing. Also, the MODULE handler can (and should) be separated from the preprocessor.

This idea has first been formulated in two Ph.D. theses by Elmqvist [7.7] and by Runge [7.18]. Both scientists came to quite similar constructs independently. In both languages, DYMOLA [7.7,7.9] and MODEL [7.18], there exist language elements comparable to the MODULE presented herein. DYMOLA proceeds by using structural analysis of the equations and formulae manipulation techniques, whereas MODEL leaves the statements as they are, and uses implicit numerical integration techniques during execution, as has been done for years in linear network analysis programs. Implicit integration is somewhat more general since there exist legitimate system's descriptions which cannot be rearranged to form an executable set of statements (if equations cannot be solved for a particular variable in a closed form, or if algebraic loops are involved which inhibit proper grouping of statements to form an executable sequence). Using formulae manipulation is, on the

contrary, somewhat more robust since illegitimate models will be automatically detected whereas this is not necessarily the case when implicit integration is used. Implicit integration will result in lower compilation and higher execution cost compared to the proposed solution technique. A combination of both techniques could possibly be the final answer.

A MODULE can be thought of as a network element with as many legs as there exist formal parameters of the MODULE. Elmqvist and Runge describe their "MODULEs" in a quite similar way to the one proposed except that their "MODULEs" are only intended for purely continuous simulation. However, both apply a programmable topological description to "plug" different MODULEs together, whereas we use an equation oriented approach for that purpose as for the description of MODULEs. (This feature exists as an option in DYMOLA as well.)

Let us return once more to the simple RLC-network presented earlier. Fig. 7.8 shows how this network can be further decomposed.

```
MODULE RES (U, I, R);
  U = R*I
END (* RES *);
MODULE CAP (U, I, C);
  U' = I/C
END (* CAP *);
MODULE IND (U, I, L);
  I' = U/L
END (* IND *);
MODULE RC (U1, U2, I1, I2, R, C);
  VAR
    STATE UC;
    ALGEBR IR;
  CONTINUOUS
    CAP (UC, IC, C);
    I1 = IR + I2;
    RES (U2, IR, R);
    U2 = U1 - UC
  END (* CONTINUOUS *)
END (* RC *);
...
CONTINUOUS
  ...
  UQ = f(TIME);
  RC (UL=U2, IQ=I1 <- UQ=U1, IL=I2, R, C);
  IND (IL=I <- UL=U, L);
```

Fig. 7.8: RLC-network further decomposed

In this program, the user must still know that the IND-MODULE has to compute the current (IL) and not the voltage (UL) to obtain a set of equations in integral form.

He can, however, also automate this procedure by specifying an additional MODULE OUT which has no

inputs and which defines as outputs precisely those variables needed for printout (e.g. IL and UL). This is demonstrated in Fig. 7.9.

```
MODULE OUT (UL, IL);
  VAR
    ALGEBR  UQ,IQ;
    REAL  C,L,R;
  CONTINUOUS
    UQ = f(TIME);
    RC (UQ, UL, IQ, IL, R, C);
    IND (UL, IL, L)
  END (* CONTINUOUS *)
END (* OUT *);
CONTINUOUS
  OUT (UL, IL <-);
END (* CONTINUOUS SUBSYSTEM *)
```

Fig. 7.9: RLC-network finally modular

As one can see, the user must specify the inputs and outputs at MODULE calls only if he uses them directly in the continuous subsystem (CONTINUOUS), but not if they are used within another MODULE definition. This is evident since, in the latter case, one can first replace MODULE calls by their MODULE definition bodies and then handle the already expanded MODULE in globo. That is, MODULEs when called from MODULEs are treated like MACROs. In the above modeling technique, the continuous subsystem will consist of one single statement only to call the root-MODULE "OUT", and the user is entirely relieved of solving any equation for particular variables. This modeling technique combines the flexibility and universality of equation oriented languages with

the convenience of network modeling techniques.

By using a programmable topological definition facility, as offered in DYMOLA or MODEL, the user can define base networks through equations (as above), whereas he is provided with topological description elements to utilize previously defined MODULEs in the system description or for the definition of (hierarchically higher) composed MODULEs. A typical example of such a topological statement could be:

```
CONNECT speed(gear) TO velocity(motor)
```

where "motor" and "gear" are two previously defined MODULEs, and "velocity" and "speed" belong to the sets of their communicating variables (formal parameters). This is a very natural and convenient way to formulate complex networks. A topological description element (like CONNECT) can involve quite complex activities, in that, for instance, the Kirchhoff' laws can be hidden behind such a topological description. They need not be explicitly specified by the user. If a MODULE is called in an equation oriented way (as proposed in Fig. 7.7b or Fig. 7.7c), the Kirchhoff' laws must be specified by the user through additional equations. This can be seen in MODULE RC of Fig. 7.8.

Some disadvantages of topological descriptions (and the reasons for which we do not currently exploit such possibilities in our software) are discussed in chapter VIII.

B) The system engineer should also be given the possibility to extend the basic language definition itself. For this purpose, the preprocessor should be constructed in such a way that it can be easily augmented to accommodate new

ideas. For this task, the most recent compiler building techniques employing structured programming and structured data representation should be applied as described, for instance, by Wirth [7.20]. The author suggests, therefore, to design the language as much as possible as a LL(1) language (cf. section VI.1), for which the syntax is to be described formally either by use of a BNF-notation or by use of syntax diagrams. Compilers for such languages can be written in a straight forward manner and are, thus, easily readable. Modifications can be established locally, usually without leading to "dirty" side effects.

VII.3.3) Transparency and the Access to Primitives:

The language should be defined in such a way that its primitives are accessible by the user directly, and that it is transparent to him how more complex building blocks are composed of these primitives.

On the continuous side, the essential primitive is the integral operator (or the "'"-symbol as in our examples, which are equivalent). The language can provide other dynamical operators (e.g. lead-lag compensator, pipeline, general transfer function, etc.), which are usually resolved by system defined MACROs. The user should then be aware how these are composed of the primitive integral operator. The problem is, however, not so critical in this context.

On the discrete side, the essential primitive is the event. By use of this primitive, all imaginable situations can be formulated. It is, however, often more convenient to combine frequently used specific sequences of events to so-called discrete process descriptions. Different DISCRETE PROCESSes and different process instances (transactions) are executed "in parallel", and it is left to the system to sort out

which is the appropriate sequence of events during execution. This description technique is more convenient since it is closer to the manner in which the modeler is used to look at his problems. In this way, physical units (like a barbershop) can be depicted in the language as independent PROCESSes, and the active elements in that PROCESS can be modeled either as FACILITies and STORAGEs (like the barbers) or as transactions "flowing" through the PROCESS (like the customers). This method of modeling allows for much better structuring of the model, and it is much easier, in this way, to first formulate a rough model of the PROCESS which can then be improved by step-wise refinement.

However, if the program so constructed behaves incorrectly, the modeler will find it, in general, much more difficult to detect the reason for the malfunction than in the case where an event description was used. In the latter case, he can use event tracing to determine why the program did not execute as expected. (For each event, a control line is then printed on output explaining the nature of the currently executed event.) Although event tracing principally works as well in the case where a process description is used, the modeler will find it very difficult to decipher the control output obtained since he has not programmed the events directly by himself. For this reason, it is extremely important that the user understands the way in which the complex building blocks of his process description (like ADVANCE, SEIZE, SENDSIG, etc.) are decomposed into their primitives. This, as a matter of fact, is often the simplest way to explain the intricate semantics of complex building blocks to the user. An example for this will be given in chapter VIII.

VII.3.4) "One-to-one" Correspondence between System and Model:

When developing a simulation language, one should try to

construct it in such a way that the modeler can represent the single building blocks of the physical system under investigation by (eventually composed) building blocks of the language, and to represent the modes in which the elements of the system cooperate with each other by constructs of the language to combine building elements to larger building blocks and, finally, to a functional description of the system. This "one-to-one" correspondence of system and model will certainly lead to a much more methodical way of modeling and, by these means, improve the robustness of the model. Furthermore, there are often several system engineers involved in the formulation of one model of a complex physical system (e.g. an atomic reactor). This modeling approach is essential to allow for a subdivision of the model into single entities which can be constructed and debugged independently, and which cooperate only through interfaces which can be properly described beforehand.

The single modeling element consists partly of structures and partly of data. These should be separable within the building block, but both should, nevertheless, be codable within the same building block so that one physical building element can be expressed by one building block of the language as well. The building elements of the physical system are connected by topological structures. Equivalent constructs must be made available in the language as well. These should, furthermore, also be descriptive elements of the building blocks themselves to allow for a hierarchical structuring of building blocks. Physical "modules" can, of course, consist of partly continuous and partly discrete elements. For this reason, it is certainly useful if this situation can be coded in one single building block of the language as well.

This demand conflicts with the earlier expressed wish to have a clear separation of the continuous and the discrete subsystem. Therefore, we suggest that the user can specify

MODULEs which possibly consist of both a continuous and a discrete part, whereas the MODULE handler regroups the description in such a way that, on output, all continuous subsystems and all discrete subsystems are merged to form a system's description as the preprocessor should find it to produce numerically well-conditioned run-time code.

When a model of a physical system has been constructed, it remains to describe in terms of language elements, the experiment which is to be performed on the model. It seems essential that different experiments can be carried out without need for redesigning the model of the system, as one would do in a real-world experiment. It could prove useful to let several experiments be executed by one simulation program. For this reason, the language should be designed to allow several experiments to be formulated subsequently. These are called experimental frames, or simple FRAMEs. All these FRAMEs form together the EXPERIMENT segment of the simulation program.

The experiment description is composed of one part describing the control signals (input signals) to the model, and one part describing the quantities which are to be measured and output (output signals). It may be convenient to separate these two parts from each other in that the EXPERIMENT segment only describes modes of control whereas an OUTPUT segment is used to describe quantities to be sampled.

The considerations in this section are strongly influenced by the pioneer work in modeling methodology as performed by Zeigler [7.22] and Oren [7.15,7.16,7.17]. These were the first scientists who tried only recently to conceptualize modeling in a methodological manner.

VII.3.5) Ease of Learning Syntax and Semantics:

Two main goals are to be achieved: It should be easy to write programs by one's self in that language, and it should be easy as well to read programs coded by somebody else. These two goals tend to compete with each other. To meet the former goal, we want the different elements of the language to use the same syntactical constructs as much as possible. To meet the latter goal, we want to be as flexible as possible in choosing appropriate mnemonics and close to conversational English constructs.

To give an example of the conflicting nature of the two goals, let us consider, once again, the dry friction example stated above (Fig. 7.2). To meet the second goal, we introduce the '='-symbol in the notation of equations of the parallel section and the ':='-symbol in the notation of statements of the procedural section. By these means the inherent difference between parallel and procedural code is clarified, in that, for instance,

```
I := I + 1;
```

is a meaningful statement, whereas

```
I = I + 1;
```

is a meaningless equation. This rule will, thus, help to improve the readability of programs. However, it will, at the same time, complicate the writing of programs since it simply introduces an additional (not necessarily required) syntactical construct to remember.

VII.3.6) Few Language Elements:

The language should consist of as few elements as possible

to make it easily learnable. On the other hand, we require many language elements to obtain short user's programs. If there are not enough primitives offered by the language, the coding of complex problems may become very cumbersome, and the resulting source program will be long. This factor is best considered by providing a hierarchical structure of both language and documentation. By this means, the user can first read an introductory manual which teaches him how to utilize basic features required for modeling of simple problems. This can be learned in a short time. Later on, when he realizes that his problem is more intricate than he originally thought, he may study another manual which enables him to use advanced features of the language. The user must be able to code simple situations in a simple manner, but should be assisted when coding more complex situations.

VII.3.7) Short Users' Programs:

The user should not be required to provide unnecessary information (like typing FORTRAN COMMON-blocks). This point must be balanced against the following considerations:

VII.3.8) Provisions for Error Detecting:

Some redundancy should be left to the program for error detecting purposes. The author feels that a modern simulation language should require that all variables be declared at the beginning of the program. This enables the preprocessor to detect many typing errors. This statement has been mentioned on many occasions (e.g. in the development of PASCAL [7.11]). It is, therefore, amazing that none of the CSSL-type simulation languages, to our knowledge, have adopted this idea, and that this fact is even praised by many developers of new simulation software.

VII.3.9) Well-Conditioned Run-Time Code:

This point has been discussed in detail in previous chapters.

VII.3.10) Robustness:

This aspect has also been discussed already, partly in chapter VI, and partly in previous remarks of the current chapter. There is no need to resume discussion of this aspect here.

VII.3.11) Discussion:

If we compare the requirements of the language with the solutions presented in previous chapters in the discussion of the GASP-software [7.3,7.4], we can conclude that hardly any of these requirements have been really taken into account when GASP-V was developed. The reason for this comes from the fact that GASP-V is a FORTRAN-IV subroutine package. FORTRAN-IV is an atrocious language with respect to structuring capabilities and programming safety.

VII.4) The Structures of the Language:

VII.4.1) The Overall Structure:

Fig. 7.10 shows the overall structure of the language resulting from the previously stated requirements.

MODULE AND MACRO DEFINITION SEGMENT
DATA DEFINITION SEGMENT
DECLARATION SEGMENT
EXPERIMENT SEGMENT
SYSTEM SEGMENT
OUTPUT SEGMENT

Fig. 7.10: Overall structure of the language

It may seem strange that the DATA DEFINITION SEGMENT preceeds the DECLARATION SEGMENT, but the reason for this can easily be explained by the following example.

```
data definition segment:

CONST
  INTEGER MAXENT=10, MAXSTOR=20, ARRDIM=25;

declaration segment:

TYPE
  ENTRY = RECORD REAL EVTIME; INTEGER EVCODE, REACTNBR;
                 REAL ACCBPROCT, LSTARTT END;
  INFO  = RECORD INTEGER DUMMY, EVCODE; REAL STORTIM END;

VAR
  FILE OF MAXENT ENTRY: EVENTFILE;
  FILE OF MAXSTOR INFO RANKED LVF ON STORTIM: PRIOQUEUE;
  FILE OF MAXSTOR INFO RANKED FIFO: WAITQUEUE;
  ARRAY [ARRDIM] OF STATE CONC;
```

Fig. 7.11: Example of data definition and declaration

As this example shows, it may be valuable to use previously

defined constants in the declaration segment.

VII.4.2) The MODULE and MACRO Segment:

This segment consists of the following four blocks.

INVOCATION BLOCK
DEFINITION BLOCK
DISPOSITION BLOCK
DESCTRUCTION BLOCK

Fig. 7.12: Parts of the MODULE and MACRO segment

MACROs and MODULEs cannot be translated off line since their generated code depends on the environment in which they are used. They can, however, be stored in a "symbolic" (source code) library.

The statement

INVOKE macro- or module name

can be used to load a previously stored MACRO or MODULE from the symbolic library.

Although such an option is directly available on most computers ("INCLUDE" on IBM 360/370, "*CALL" on CDC 6000-series UPDATE, the EXPAND program on PDP-11, etc.) it is more convenient to define the option in the language itself to improve portability of users' programs.

DISPOSE macro- or module name

is used to store a previously defined MACRO or MODULE body

in the symbolic library.

DESTROY macro- or module name

is used to delete a previously stored MACRO or MODULE body from the symbolic library.

MACRO and MODULE definition bodies have been shown in Fig. 7.4 and Fig. 7.6 to Fig. 7.9, and require no further discussion.

One should be able to call previously defined or invoked MACROs and MODULEs from the definition bodies of other MACROs and MODULEs. An example for this has already been shown in Fig. 7.8 and Fig. 7.9. Nesting of MACROs and MODULEs should, in fact, be possible to any depth.

If a MODULE is called from within the continuous subsystem (CONTINUOUS) or from within a MACRO, its inputs and outputs must be identified, whereas this is not the case if it is called from within another MODULE, as it has previously been shown.

MACROs (but not MODULEs) should be interpretative. It should, for example, be possible to specify the following:

> Generate a certain block of statements as many times as indicated by the fifth formal attribute of the MACRO.

This could, for instance, be realized as shown in Fig. 7.13.


```
MACRO HEAT (U <- SIGMA, UUL, UUR, CONST NR);
  MACVAR
    ARRAY OF [vNRv] OF STATE U;
  MACCONTIN
    U[1]' = SIGMA*(U[2] - 2.0*U[1] + UUL);
    MACFOR  I:=2 TO  NR-1  DO
      U[vIv]' = SIGMA*(U[vI+1v] - 2.0*U[vIv] + U[vI-1v]);
    MACEND;
    U[vNRv]' = SIGMA*(UUR - 2.0*U[vNRv] + U[vNR-1v])
  MACEND
MACEND (* HEAT *);
```

Fig. 7.13: Example for an interpretative MACRO

where "CONST NR" indicates that this attribute must be entered by value when the MACRO is called, and v...v means that the actual value must be inserted here. In principle, one can use the notation of any general purpose MACRO-language like ML/I [7.1], but a compromise must be found between the competing versatility of the tool and the readability of the MACRO code. The author believes that not all of the features commonly offered by general purpose MACRO-languages are really needed in a simulation environment.

A MODULE body can be composed of the following sections:

HEADER SECTION
DATA DEFINITION SECTION
DECLARATION SECTION
INITIAL SECTION
CONTINUOUS SECTION
DISCRETE SECTION
TERMINAL SECTION

Fig. 7.14: Sections of a MODULE body

As one can see, the MODULE definition body can consist of almost the same structural components as the overall program except that no experiment description section is foreseen. This structural complexity is required to guarantee true modularity. This is best illustrated at an example. Let us try to write a MODULE for the previously demonstrated dry friction torque. This could be realized as shown in Fig. 7.15.

```
MODULE DRYFRICTION (TFR, T, OMEGA, CM, T1, T2);
  VAR
    INTEGER NL;
    MODEL DRYFRIC;
    SEVENT MOD1,MOD2,MOD3;
  INITIAL
    NL := 2
  END (* INITIAL *);
  CONTINUOUS
    MODEL DRYFRIC (TFR <- T, OMEGA);
      ...
    END (* DRY FRICTION MODEL *);
  END (* CONTINUOUS SUBMODULE *);
  DISCRETE
    EVENTS
      ...
    END (* STATE EVENT DESCRIPTION *)
  END (* DISCRETE SUBMODULE *)
END (* DRY FRICTION MODULE *);
...
CONTINUOUS
  ...
  DRYFRICTION (TF=TFR <- TMOT=T, PHID=OMEGA, CM, T1, T2);
```

Fig. 7.15: MODULE for a dry friction force

This MODULE can be used several times within the continuous subsystem. All internally declared variables (NL, DRYFRIC, MOD1, MOD2, and MOD3) obtain a new unique name during each expansion of the MODULE. It is always to be called from within a CONTINUOUS section. However, the MODULE handler will split up the definition body and place its different sections where they belong.

VII.4.3) Declaration and Data Definition Segments:

These segments look very similar to those of any modern general task language (like PASCAL [7.11]).

In the declaration segment, one will, however, find more predefined types than in a general task language like

STATE, DSTATE, MEMORY, MODEL

to declare state variables (for differential and difference equations), memory variables (loop breakers of the sorting procedure, e.g. outputs of DELAY-functions), and MODELs of the continuous subsystem, as well as

SEVENT, TEVENT, PROCESS, FACILITY, STORAGE, GATE

to declare events (state- and time-events), to declare DISCRETE PROCESSes with their FACILITies, STORAGEs and GATEs of the discrete subsystem (comparable to GPSS-V [7.19]), or to declare CONTINUOUS PROCESSES as discussed above.

In the data definition segment, one will find special constructs like tabular function generators which are not provided in a general task language.

VII.4.4) The EXPERIMENT and OUTPUT Segments:

These two segments describe the experiments which are to be performed on the model of the system, and the output representations to be produced.

The EXPERIMENT segment can describe several independent experiments.

```
experimental FRAME 1
...
experimental FRAME k
```

Fig. 7.16: Blocks of the EXPERIMENT segment

to describe different case studies. Different experimental FRAMEs are used to describe different experiments to be performed, as, for example, the use of different INTEGRATion METHODs. Each experimental FRAME must contain precisely one SIMULATE statement. Each FRAME can, nevertheless, involve several simulation runs, e.g. a whole optimization study, or sets of PARAMETERs and/or INITial CONDitions.

This separation of experiment description from system description implies that experiments have to be formulated for the system as a whole, and not for subunits of it. In such a language, it will, for instance, not be possible to specify that different state equations are to be integrated by use of different integration algorithms during one and the same simulation run. If this feature is wanted, one has to expand the language in such a way as to let each PROCESS be accompanied by a rudimentary experiment description associated with it. This has, however, intentionally not been considered up to now for two reasons:

a) This feature creates many additional difficulties for the synchronization between different PROCESSes. GSL [7.10], for instance, a simulation language which offers this facility, spends approximately 50% of the whole software documentation volume to deal with the problems arising in using several integration rules for different submodels, and how these difficulties are to be overcome. We found, however, very few examples up to now where the advantages of this additional feature were able to compensate for the overhead required for synchronization purposes.

b) Even if such a system exists, the average user will hardly have the knowledge to make proper use of this facility.

For these reasons, we will not discuss this facility any further.

The situation must be judged differently as soon as the simulation compiler is laid out to produce code for several processors to work in parallel (e.g. each PROCESS to run on a processor of its own). In that case it will not do any harm to add this feature to the definition set of the simulation language since then these synchronization problems arise anyhow, and one will have to find a solution to them.

VII.4.5) The SYSTEM Segment:

This segment contains the system's description. It can be structured as follows:

INITIAL BLOCK
CONTINUOUS BLOCK
DISCRETE BLOCK
TERMINAL BLOCK

Fig. 7.17: Blocks of the SYSTEM segment

This looks very similar to the structure of commonly used CSSL software except that the DYNAMIC block is replaced by a CONTINUOUS and a DISCRETE block.

The CONTINUOUS block will, in principle, look like the DYNAMIC block of any CSSL-type language (e.g. CSMP-III [7.23]). Additional elements are the MODELs which can be nested to any depth to allow for better and hierarchical structuring of the systems' descriptions, and the CONDIT statement to describe state-conditions. These have been previously demonstrated (cf. Fig. 7.2).

In such a language, NOSORT and PROCEDural sections are possible to code, but they are much less "useful" than in a so called "continuous" simulation language since the compiler should take care that no illegitimate discontinuities are coded in such a section. (The main advantage of PROCEDural or NOSORT sections as praised by CSSL software is the possibility to code discontinuities by writing IF ... THEN ... ELSE. Precisely this must not be done in a combined system simulation language since it deprives the compiler of any fair chance to generate numerically well-conditioned run-time code.

There are five legitimate ways to code discontinuities in a combined system simulation language:

a) by using precoded discontinuous functions offered by the language (like the GASP-functions of chapter V),

b) by modeling discontinuities as time- and state-events in the discrete subsystem with associated CONDIT statements in the continuous subsystem,

c) by coding subroutines (or function subprograms) which are declared to be CONTINUOUS SUBROUTINEs or CONTINUOUS FUNCTIONs, and which, in fact, enlarge the set of (a),

d) by coding CONTINUOUS PROCESSes as will be illustrated in chapter IX in an example (DOMINO game),

e) by coding MODULEs which are kept as source modules in a symbolic library.

The usual ways to code discontinuities are (a) and (b), whereas (c) and (d) require some sophistication and are not recommended to the unskilled user. (e) is not really an additional concept, but rather an extension to improve modularity.

The DISCRETE block can look like any statement oriented discrete simulation language (e.g. SIMULA-67 [7.6]). Additional elements are

CREATE, SUSPEND, RESUME, DELETE

to activate and deactivate instances of CONTINUOUS PROCESSes, as will be illustrated in the DOMINO example of chapter IX.

VII.5) Global Versus Local Variables:

In the old days of information processing, computer languages used to be constructed in such a way as to let each independent programming unit have its own set of variables

assigned to it. We call this a concept of local variables. A typical example of this type of language is FORTRAN-IV. Each SUBROUTINE has its own variables assigned to it which keep their values even over several calls to the routine. Data communication between SUBROUTINEs is only possible through lists of formal parameters or through COMMON variables.

Good modern computer languages like PASCAL [7.11] make use of a global variable concept. By these means, the user can declare new variables on each hierarchical level which are then valid in the PROCEDURE in which they are defined as well as in any PROCEDURE called by it. There is no need to include any variable in the list of the formal parameters as long as the PROCEDURE is not called several times by different actual arguments. PROCEDUREs which are on the same hierarchical level can communicate data with each other only by declaring variables on a hierarchically higher level for that purpose.

This elegant concept is very clear and clean from the aspect of information processing. It has, however, two major drawbacks:

a) PROCEDUREs making use of global variables (and direct exits) may not be precompiled. They must be stored, if at all, in source form. In fact, PROCEDUREs are not really meant to be compiled separately.

b) PROCEDUREs making use of these possibilities are much more difficult to describe since they do not communicate data through a distinct interface only ("back-door" programming!).

Such a concept is, therefore, not really modular. However, modularity is an important requisite in a simulation environment. In the design of the concepts of a simulation language, one has to take this aspect into account, and design the language in such a way that provisions exist to precompile those structural blocks which can be stored in compiled form. For this reason, a language like PASCAL is very well suited for the coding of a simulation compiler which is a "closed" program, whereas a FORTRAN-like language is much better suited for the coding of a simulation run-time system which is, principally, to be compiled once, but to which different subprograms are to be added for each application problem.

In the long run, it would be more consistent with our ideas to use a PASCAL-like general task language for which the PROCEDURE concept has been enlarged to a PROCESS- or CLASS concept (which again would be modular) also for the simulation run-time system. However, although these concepts have been discussed on many occasions, and although there exist implementations of such features (SIMULA-67 [7.6], MODULA [7.21], PORTAL [7.12,7.13,7.14]), there does not exist any such language to date which is widespread and which has been generally accepted. SIMULA-67 has certainly found many "disciples", but even for this language there does not exist any good library of carefully debugged CLASSes for specific applications like numerical integration which could, for example, compete with the Kahaner implementation of the Gear algorithm. Such libraries exist to date only for FORTRAN-IV and for ALGOL-60.

Moreover, the main disadvantage of using FORTRAN-IV, namely its unsatisfactory programming safety, is not so critical in our application since the target language code of the users' programs is machine generated and not hand coded.

MODULEs and MACROs must be stored in source form, as we have seen. They can, therefore, easily use a global variable concept. Variables which are locally declared obtain a new unique name each time the MODULE (MACRO) is called.

Subprograms can syntactically look similar to PASCAL PROCEDUREs, to grant safe programming, but effectively they must be rather similar to FORTRAN SUBROUTINEs to satisfy the requirements of modularity.

It may, furthermore, also be useful to precompile PROCESSes. Since the statements of a PROCESS are only sorted internally, but remain together as a block, this can be done. The resulting code is again similar to a FORTRAN SUBROUTINE, except that the number of internally used state variables and predefined functions must be declared when a PROCESS is called in as an EXTERNAL PROCESS. Consequently, PROCESSes must also use a concept of local variables for that purpose.

This concludes the description of the useful language structures and of the semantics of variables, as they seemed to be extractable from the requirements of the language as postulated in section VII.3.

References:

[7.1] P.J.Brown: (1975) "Macro Processors and Techniques for Portable Software". John Wiley.

[7.2] F.E.Cellier: (1976) "Macro-Handler for Simulation Packages Using ML/I". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 515 - 521.

[7.3] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[7.4] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 391 - 402.

[7.5] F.E.Cellier, Ferroni B.A.: (1974) "Modular, Digital Simulation of Electro/Hydraulic Drives Using CSMP". Proc. of the 1974, Summer Computer Simulation Conference, Houston, Texas, U.S.A.; pp. 510 - 514.

[7.6] O.J.Dahl, Nygaard K.: (1966) "SIMULA; A Language for Programming and Description of Discrete Event Systems". Oslo, Norwegian Computing Center.

[7.7] H.Elmqvist: (1978) "A Structured Model Language for Large Continuous Systems". Form: CODEN LUTFD2/(TFRT-1015)/1-226/(1978). Ph.D. Thesis. Lund Institute of Technology, Dept. of Automatic Control, Lund, Sweden.

[7.8] H.Elmqvist: (1979) "Manipulation of Continuous Models Based on Equations to Assignment Statements". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company, (Editors: L.Dekker, G.Savastano, G.C.Vansteenkiste); pp. 15 - 21.

[7.9] H.Elmqvist: (1979) "DYMOLA - A Structured Model Language for Large Continuous Systems". Proc. of the Summer Computer Simulation Conference, Torronto, Canada.

[7.10] D.G.Golden: (1972) "A Generalized Language for Discrete/Continuous Simulation". Form: 72-18693. Ph.D. Thesis. Case Western Reserve University, Cleveland, Ohio, U.S.A..

[7.11] K.Jensen, Wirth N.: (1974) "PASCAL User Manual and Report". Lecture Notes in Computer Science, Springer Verlag.

[7.12] H.Lienhard: (1978) "PORTAL Language Definition". To be ordered from: Landis & Gyr AG, Zug, Switzerland. (Partly in German).

[7.13] H.Lienhard: (1978) "Die Echtzeitprogrammiersprache PORTAL, eine Uebersicht". Landis & Gyr Mitteilungen 25(1978); pp. 2 - 8. (Also available in English.)

[7.14] H.Lienhard, Meyer M., Steinle B., Wehrli P.: (1979) "Simulation and Process-Control with Parallel Processes as Implemented in PORTAL - Experience and Outlook". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company, (Editors: L.Dekker, G.Savastano, G.C.Vansteenkiste.)

[7.15] T.I.Oren: (1977) "Modelling, Model Manipulation and Programming Concepts in Simulation: A Framework". Proc. of the IFIP Working Conference on Modelling and Simulation of Land, Air and Water Resources Systems, Ghent, Belgium. Published by North-Holland Publishing Company (Editor: G.C.Vansteenkiste).

[7.16] T.I.Oren: (1978) "Concepts for Advanced Computer Assisted Modelling". Proc. of the Symposium on Modeling and Simulation Methodology, Rehovot, Israel. Published by North-Holland Publishing Company (Editor: B.P.Zeigler).

[7.17] T.I.Oren, Zeigler B.P.: (1978) "Concepts for Advanced Simulation Methodologies". Simulation, vol. 30 no. 6 : June 1978.

[7.18] T.F.Runge: (1977) "A Universal Language for Continuous Network Simulation". Form: UIUCDCS-R-77-866. Ph.D. Thesis. University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, Illinois, U.S.A..

[7.19] T.J.Schriber: (1974) "Simulation Using GPSS". John Wiley.

[7.20] N.Wirth: (1976) "Algorithms + Data Structures = Programs". Prentice Hall, Series in Automatic Computation.

[7.21] N.Wirth: (1977) "MODULA: A Language for Modular Multiprogramming". Berichte des Instituts fuer Informatik, Nr. 18. To be ordered from: Institute for Information Processing, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[7.22] B.P.Zeigler: (1976) "Theory of Modelling and Simulation". John Wiley.

[7.23] (1972) "Continuous System Modeling Program III (CSMP-III) - Program Reference Manual". Program Number: 5734-XS9, Form: SH19-7001-2. To be ordered from: IBM Canada, Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, Canada.

VIII) DISCUSSION OF EXISTING SOFTWARE:

The existing software for combined simulation has recently been reviewed in an excellent survey by Oren [8.10,8.11]. He has collected information on about 30 different simulation systems for combined system simulation. A book is under preparation by the same author which will describe the different software products for combined simulation in greater detail. For this reason, there is no need to duplicate in this thesis the excellent work done by Oren.

Due to the fact that the term "combined system simulation" had never before been explicitly defined, the goal of many of the software products discussed by Oren is quite different from what has been presented herein. Many of these programs would not even belong to the class of combined system simulation languages according to our definition of the term.

Furthermore, most of the surveyed software products have never been released. From the 18 programs discussed in [8.10] only 8 are listed in Appendix B under "available on ...". Out of these only 2 programs (GASP-IV [8.15] and HOCUS-III [8.14]) are available on several different computer makes. Out of these two, HOCUS-III takes a quite different approach to achieve another goal than presented herein. For this reason, considering only those programs being implemented at several different installations and being widely used by different people, very few of them remain under consideration. Others must be considered to be collections of valuable ideas rather than simulation languages. Among those programs which have been implemented at several installations, GASP-IV has by far the largest distribution. Together with its descendant, GASP-V, this program follows the ideas mentioned in chapter IV of this thesis. The numerical behaviour of the GASP software is, in our experience, the best

of all existing run-time packages for combined simulation. Unfortunately, there is no provision in GASP for a user-oriented input definition (no preprocessor is involved). Therefore, none of the ideas presented in chapter VII could be realized in GASP.

From the information processing point of view, pioneer work has been done in the definition of the language GSL [8.7] following the original ideas of Fahrland [8.5,8.6]. GSL has the best language structure of all the languages published so far. Unfortunately, GSL is one of the languages which has never been really released. Moreover, even GSL has severe shortcomings both concerning the underlying run-time structure and concerning the language definition itself (the recent developments in the field of information processing and especially compiler building have not sufficiently been taken into account). Many of the ideas presented in chapter VII have not been accommodated in GSL either. Variables are, for instance, not foreseen to be declared in GSL.

A new simulation language, COSY [8.2,8.3] (standing for COmbined SYstems), is under development by the author. It involves a PASCAL-coded preprocessor which translates a new input definition language (following the ideas developed in chapter VII) into GASP-V executable code. Thus, GASP-V will be used as a target language for COSY. This language, however, will not be released before 1980.

Another promising language under development which has some resemblance to COSY is GEST [8.9]. The original GEST language has never been released. Meanwhile, the language definition has significantly been improved, and an implementation of the new version, GEST'78 [8.12,8.13], is under development.

An entirely different approach has been taken in the defini-

tion of SMOOTH [8.20] which uses a network approach combining GERT networks for discrete simulation [8.16] with STATE networks for the continuous subsystem. A new program of this class, SLAM [8.17], has been released in 1979 by Pritsker combining Q-GERT [8.16] with GASP-IV [8.15]. This approach certainly results in extremely short application programs, at least for such applications for which there are elements provided in the language. However, a network approach cannot be as general as a structured language. Benyon [8.1] states: "Such a diagrammatic approach to modelling can be very useful in some instances, but experience with the continuous languages has been that the diagrams soon grow too complicated to be enlightening, once one advances beyond quite simple models".

Moreover, it is often stated that network languages are easier to learn than equation oriented structured languages. We would deny this statement for two reasons:

a) The number of language elements (primitives) of such block oriented languages required to obtain at least a certain degree of flexibility is much larger than for equation oriented languages. GPSS-V [8.19], for example, consists of 41 building blocks and Q-GERT [8.16] offers 24 of them (Q-GERT requires a smaller number of blocks for an even higher degree of flexibility, because the single building blocks are more decomposable and recombinable). All of these building blocks must be understood before a truly complex application program can be coded.

b) Since the single building block describes a rather complex entity compared to a simple event description, the semantics required to describe such an element are much more complex. (A complex situation can either be expressed by a complex syntax consisting of many "small" building blocks with primitive semantics, or by a simple

syntactical construct consisting of few "large" building blocks with complex semantics, but never by both simple syntax and semantics.) This can be illustrated in an example. Considering the GENERATE-block of GPSS-V, it seems that the meaning of this block is very easily explained.

GENERATE A, B

means that a new transaction is to be generated with a uniform distribution in the interval [A-B,A+B]. The novice user of GPSS-V will take this definition for granted, and let this block be followed by a SEIZE-block to have the transaction occupying a FACILITY. In practice, if this FACILITY is already occupied when the transaction is born in the system, this transaction will remain in the GENERATE-block and inhibit the generation of new transactions. This shows that the semantics required to describe this simple situation properly are much more complex than might be thought. Very commonly, an error occurs due to the fact that semantics are involved which have not been reported to the user in the introductory manual.

This last example unveils another weak point of network languages: The program, as specified above, will "work" which means that output will be produced, although this output will be incorrect. With a high probability, the user will, thus, never detect that his program is erroneous. The reason for the inability of GPSS-V to detect the error, lies in the fact that hardly any redundancy has been left in the code which could enable the system to detect errors in the source program. The standard of programming robustness in such languages is, therefore, generally very low. We are, however, convinced that a lot can still be done to improve software robustness in this respect. The problem has simply not been considered carefully enough to date.

Much more promising, it seems to us, is the new network approach as proposed by Elmqvist [8.4] and by Runge [8.18] which is based on equation-oriented languages for the construction of network elements. These are then connected by a programmable topological description facility. Intended by both authors for continuous system simulation only, their concept can be extended to encompass discrete system simulation as well. This has been shown in chapter VII in the discussion of the MODULE element. Also discrete network languages could (and should) enable the user to code his own base networks by use of an equation-oriented language. In this way, languages like Q-GERT would gain remarkably in terms of flexibility. To our knowledge, none of the currently available discrete network languages offers such facilities.

One final insufficiency of network description facilities, as they are offered in today's network simulation languages, is to be discussed. This concerns their restricted capabilities with respect to welding different types of networks together. Different types of continuous networks (like electrical networks, mechanical networks, hydraulic networks, pneumatic networks, and thermodynamic networks) may communicate with each other in DYMOLA and MODEL. However, they may not by graphically coupled. "Nodes", at which different types of networks interfere with each other, must be modeled as base (equation-oriented) MODULEs. These correspond then to the transducers of a real equipment.

The only quantity being common to all of these network types is energy. If, in the model, the energy flow is represented rather than the signal flow, we obtain an easy and elegant mean to connect different network types. This is possible by using the bond-graph modeling technique. However, bond-graphs, as they are understood today, are not universally applicable yet. There exist systems (e.g. in control) where one does not get around signal descriptions.

For this reason, a generalized bond-graph theory [8.8] has recently been suggested in which energy flows and signal flows can be modeled simultaneously. This could become an alternative network description facility for continuous simulation languages in the future.

Connections between discrete and continuous networks, on the other hand, are already possible in SMOOTH [8.20], but further research will be required to make these links sufficiently general and flexible.

References:

[8.1] P.R.Benyon: (1976) "Improving and Standardizing Continuous Simulation Languages". Proc. of the SIMSIG Simulation Conference, Melbourne, Australia, May 17-19, 1976; pp. 130 - 140.

[8.2] A.P.Bongulielmi: (1978) "Definition der allgemeinen Simulationssprache COSY". Semesterwork, Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich. To be obtained on microfiches from: The main library, ETH - Zentrum, CH-8092 Zurich, Switzerland. (Mikr. S637).

[8.3] F.E.Cellier, Bongulielmi A.P.: (1979) "The COSY Simulation Language". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company.

[8.4] H.Elmqvist: (1978) "A Structured Model Language for Large Continuous Systems". Form: CODEN LUTFD2/(TFRT-1015)/1-226/(1978). Ph.D Thesis. Lund Institute of Technology, Dept. of Automatic Control, Lund, Sweden.

[8.5] D.A.Fahrland: (1968) "Combined Discrete Event / Continuous System Simulation". MS Thesis, Systems Research Center Report SRC-68-16, Case Western Reserve University, Cleveland, Ohio.

[8.6] D.A.Fahrland: (1970) "Combined Discrete-Event Continuous System Simulation". Simulation vol. 14 no. 2 : February 1970; pp. 61 - 72.

[8.7] D.G.Golden, Schoeffler J.D.: (1973) "GSL - A Combined Continuous and Discrete Simulation Language". Simulation vol. 20 no. 1 : January 1973; pp. 1 - 8.

[8.8] D.Karnopp: (1979) "Bond Graphs in Control: Physical State Variables and Observers". To be ordered from: Department of Mechanical Engineering, University of California, Davis CA 95616, U.S.A.; to appear in: Journal of the Franklin Institute.

[8.9] T.I.Oren: (1971) "GEST: A Combined Digital Simulation Language for Large Scale Systems". Proc. of the AICA Symposium on Simulation of Complex Systems, Tokyo, Japan, September 3-7, 1971; pp. B-1/1 - B-1/4.

[8.10] T.I.Oren: (1977) "Software for Simulation of Combined Continuous and Discrete Systems: A State-of-the-Art Review". Simulation, vol. 28 no. 2 : February 1977, pp. 33 - 45.

[8.11] T.I.Oren: (1977) "Software Additions". Simulation, vol. 29 no. 4 : October 1977, pp. 125 - 126.

[8.12] T.I.Oren: (1978) "Reference Manual of GEST'78 - Level 1 (A Modeling and Simulation Language for Combined Systems)". Technical Report 78-02, Computer Science Dept., University of Ottawa, Ottawa, Canada.

[8.13] T.I.Oren, den Dulk J.A.: (1978) "Ecological Models Expressed in GEST'78". Technical Report Prepared for the Dept. of Theoretical Plant Ecology, Dutch Agricultural University Wageningen, The Netherlands.

[8.14] T.G.Poole, Szymankiewicz J.Z., Holme H.G.: (1974) "Simulation: A Problem Solving Oriented Approach". To be ordered from: P-E Consulting Group, Ltd., Surrey, U.K..

[8.15] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

[8.16] A.A.B.Pritsker: (1977) "Modeling and Analysis Using Q-GERT Networks". John Wiley, Halsted Press.

[8.17] A.A.B.Pritsker: (1979) "Introduction to Simulation and SLAM". John Wiley, Halsted Press & Systems Publishing Corp..

[8.18] T.F.Runge: (1977) "A Universal Language for Continuous Network Simulation". Form: UIUCDCS-R-77-866. Ph.D Thesis. University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, Ill., U.S.A..

[8.19] T.J.Schriber: (1974) "Simulation Using GPSS". John Wiley.

[8.20] C.E.Sigal, Pritsker A.A.B.: (1973) "SMOOTH: A Combined Continuous/Discrete Network Simulation Language". Proc. of the 4th Annual Pittsburgh Conference on Modeling and Simulation. Pittsburgh, Penn., U.S.A., April 23-24, 1973, pp. 324 - 329.

IX) COSY:

IX.1) General Concepts:

In the first chapters of this thesis, we have discussed the numerical requirements which should be met by a good run-time system for combined simulation. We came to the conclusion that GASP-V [9.2,9.3] satisfies these requirements to a great extent.

Subsequently, we have discussed the requirements of combined simulation languages from an information processing point of view. If we compare these demands with the facilities offered in GASP-V, we must come to the conclusion that hardly any of them have been considered in the design of GASP-V. In fact, GASP-V is very poor in this respect. This is not really surprising since FORTRAN-IV is also very weak in these aspects, and GASP-V (being a FORTRAN-IV application program) cannot provide any facility which is not supported by FORTRAN-IV.

The aim of this chapter is to provide a remedy for these evident shortcomings of GASP-V. For this task, we asked ourselves which were the structural elements characterizing a GASP application program, and what would be the safest and most convenient way to formulate these elements for the computer. By these means, we came to a new language definition, COSY [9.1,9.4], which should be as general in its applicability as GASP-V, but, at the same time, take into account all the requirements formulated in chapter VII. As a matter of fact, all examples given in chapter VII use the notation of COSY. A PASCAL-coded preprocessor translates any COSY application program into an equivalent GASP-V application program by generating the required "user-supplied"\ GASP-subroutines like STATE, EVNTS, asf.. GASP-V is, thus, used as target code for COSY, whereas FORTRAN-IV is the real

target language of COSY.

In a first pass, a MODULE handler administers the symbolic library and replaces calls to MACROs and MODULEs by their definition bodies. The output of this pass is still a COSY source program with all MACROs and MODULEs being expanded. On this level, the terms "MACRO" and "MODULE" do no longer exist. Upon request, a listing of the original program is output.

In a second pass, COSY programs are analysed, and three files are generated containing:

a) all required GASP subroutines, and all GASP data input, except for the body of subroutine STATE,

b) a "C-code" in which the equations of the continuous block are prepared, and

c) (upon request) a listing of the COSY program with expanded MACROs and MODULEs, and/or crossreference tables.

The third pass contains:

a) a program which sorts the equations of the continuous block, as specified in the C-code, into executable order, and generates subroutine STATE, as well as

b) a program to generate subroutine JACOB (for the Jacobian of the system) out of the C-code by means of algebraic differentiation (optional).

A fourth "pass" is used to decode previously collected error numbers into meaningful error messages.

IX.2) Restrictions:

It is evident that only such features can be offered in COSY which are codable in GASP-V as well. This imposes some restrictions on the definition set of COSY which are to be discussed.

a) The data structuring capabilities of COSY are as limited as those offered in FORTRAN-IV. The TYPE statement may be used only to define RECORDs of file entries where a RECORD may consist of a variable number of attributes which are adminstered by GASP-V as forward and backward linked linear lists. No more complex RECORD structures are available, and no pointer variables to link RECORDs in a programmable manner are accessible to the user. No symbolic TYPEs can be defined. All user variables must be declared to belong to one of the (admittedly high number of) predefined TYPEs:

   ALGEBR, BOOLEAN, DSTATE, FACILITY, GATE, INTEGER, LOGIC, MEMORY, MODEL, PROCESS, RANDOM, REAL, SAMPLE, SEVENT, STATE, STORAGE, TEVENT

   or ARRAYs of them. No SETs are available either.

b) Routines, although looking very much like PASCAL PROCEDUREs, are really SUBROUTINEs in that variables which are locally declared keep their assigned values over several calls. Routines may not be called recursively, and it is forbidden to leave them by a GO TO statement (which is a very doubtful option even in PASCAL (!)).

c) DISCRETE PROCESSes are not as versatile as they could be. A more axiomatic approach to their semantics would be useful, but FORTRAN SUBROUTINEs are a very unwieldy carrier of such a language element, and without heavy

constraints not applicable at all.

For these reasons, one may conclude that other languages like SIMULA-67 [9.5] or PORTAL [9.9,9.10] are still more useful for purely discrete simulation problems than COSY, although these languages offer much fewer language elements directly dedicated to simulation, and although their use will, consequently, result in longer, less safe, and less readable application code.

These obvious shortcomings of COSY cannot be overcome as long as there does not exist any other generally accepted language which could replace FORTRAN-IV as a target language for COSY. We have seen that either FORTRAN-IV or PASCAL are not suitable for this purpose. ALGOL-60 has drawbacks similar to PASCAL. PL/I would be a possible candidate, but many people consider its definition set too large and its semantics not axiomatic enough to make this language very enlightening either. (PL/I programs coded by others are usually not easily readable. This can, for instance, be seen in the PL/I-code generated of SIMPL/I programs [9.16], a discrete simulation language on the basis of PL/I.) PORTAL [9.9,9.10,9.11] would, in our opinion, be a promising candidate, but since this language is not administered either by a computer manufacturer or by a non-profit organization, we give it little chance to take the barrier of international acceptance in the near future. We do not see any chance to overcome this problem, as long as the computer manufacturers find a market for their hardware without offering appropriate software for it. Only the customer will, finally, be able to force them to sit together to come to an agreement concerning a modern, widely supported general task language for which not only an efficient and well tested compiler is developed, but for which also cautiously debugged mathematical application software is made available. A possible answer to this may be ADA [9.17,9.18]. Since the language definition of ADA is a product of a standardization

committee, many computer manufacturers are expected to come up with compilers for this new language in relatively short time from now. ADA shares many good ideas with PORTAL. However, its language definition is much larger. It is still too early to predict whether ADA shall become a break through. Meant for real-time applications, ADA is supposed to execute on process computers, that is (currently) on 16 bit machines. Since the language definition is rather large, it may be quite difficult to achieve reasonably efficient implementations of the full language definition for this type of computers. However, the inaugurators of ADA have forbidden any subsets (or supersets) of ADA to be implemented under the name of ADA. This is a positive contribution as it will certainly improve the portability of the software. It may, however, jeopardize an implementation on process computers. The success of ADA will, therefore, largely depend on its implementability. For the time being, no implementation of ADA exists yet.

IX.3) Examples:

This description of COSY (as well as the previous one of GASP-V) is not meant to replace a users' manual. The aim of this work is to illustrate the methodological concepts which are to be used in combined system simulation software rather than to describe in detail any particular piece of software. The subsequent examples may clarify the previously discussed concepts. A full explanation of the syntax, as it is used in COSY, will not be given. For this purpose, we refer to [9.1]. However, since one of the goals in the development of this new simulation language was to let programs coded in it be easily readable, most of the statements should be self explanatory.

IX.3.1) Continuous Simulation -- Van-der-Pol's Equation:

Statement of the problem: There are few examples of truly "continuous" systems beside of (more or less trivial) school examples. Even most of the often cited benchmark problems for continuous system simulation belong to combined system simulation according to our terminology. The well known Van-der-Pol equation, however, is a real continuous example. (Even this very simple problem is numerically difficult to treat as it has been shown in chapter VI.)

Although this thesis deals with combined simulation, this problem, as well as the following two examples on discrete simulation, have been added to demonstrate that coding purely continuous or purely discrete problems is not more difficult using a combined system simulation language than using e.g. CSMP-III or GPSS-V.

System description: The following ODE describes the Van-der-Pol oscillator as we use it in our example:

$$\ddot{y} - \mu(1.0 - y^2)\dot{y} + y^3 = 0.0$$

This second order ODE can be rewritten as a set of two first order ODE's:

$$\dot{x}[1] = x[2]$$
$$\dot{x}[2] = \mu(1.0 - x[1]^2)x[2] - x[1]^3$$

where:

$$y = x[1] \ .$$

Experiment description: Two experiments are planned.

a) In a first experiment, the parameter is to be varied. It is supposed to take the five values

$$\mu = \{0.1\},\ \{0.5\},\ \{1.0\},\ \{2.0\},\ \{5.0\}$$

In all five cases the initial conditions are:

$$x[1](t=0.0) = +0.5$$
$$x[2](t=0.0) = -0.5$$

b) In a second experiment, we let $\mu = 2.0$ be constant, whereas now the initial conditions are to be varied:

$$(x[1]),x[2]) = \{(0.0,0.5)\},\ \{(0.0,1.0)\},\ \{(0.0,2.0)\},$$
$$\{(1.0,2.0)\},\ \{(2.0,2.0)\},\ \{(3.0,2.0)\}$$

Output description: For each of the 11 runs we want to produce a high quality graph of x[2] plotted against x[1] (phase plane plot) on a (x,y) plotting device.

This concludes the description of the Van-der-Pol system. Fig. 9.1 shows the COSY program for this problem.


```
{*****************************************}
{*                                       *}
{*      CONTINUOUS SYSTEM SIMULATION     *}
{*                                       *}
{*         VAN-DER-POL'S EQUATION        *}
{*                                       *}
{*****************************************}

PROGRAM VANDERPOL (INPUT,OUTPUT);

PROJECT 49 BY ICELLIERE;


MACRO OUTGRAPH (<- CONST NR);
  MACFOR  I:=1 TO #NR# DO
  MACBEGIN
    TITLE 'VAN-DER-POL EQUATION: RUN #I#';
    GRAPH (#I#) VERSUS X1: X2;
  MACEND
MACEND  (* OUTGRAPH *);


CONST
  INTEGER  NRRUNS = 11;


VAR
  REAL  MY;
  ALGEBR  X1SQR;
  STATE  X1, X2;


STORE
  X1, X2;


EXPERIMENT
  FRAME  (* 1 *)
    PARAMETER  MY = (0.1, 0.5, 1.0, 2.0, 5.0);
    INITCOND  X1 = 0.5, X2 = -0.5;
    SIMULATE  FROM 0.0 TO 20.0  COMINT = 0.05
  END  (* FRAME 1 *);
  FRAME  (* 2 *)
    PARAMETER  MY = 2.0;
    INITCOND
      X1 = (0.0*3, 1.0*3+1.0),
      X2 = (0.5, 1.0, 2.0);
    SIMULATE  FROM 0.0 TO 20.0  COMINT = 0.05
  END  (* FRAME 2 *)
END  (* EXPERIMENT BLOCK *);


SYSTEM

  INITIAL
    SAVE  (* STORE DATA ON SAVE-FILE *)
  END;

  CONTINUOUS
    X1' = X2;
    X2' = MY*(1.0 - X1SQR)*X2 - X1*X1SQR;
    X1SQR = X1*X1
  END  (* CONTINUOUS SUBSYSTEM *)

END  (* SYSTEM DESCRIPTION *);


OUTPUT
  OUTGRAPH (<- NRRUNS)
END  (* OUTPUT DESCRIPTION BLOCK *)

END .
```

Fig. 9.1: COSY program for the Van-der-Pol oscillator.

IX.3.2) Discrete Simulation (Event-Oriented) -- Joe's Barbershop:

This is a known benchmark problem in discrete simulation. It is often referred to as the "single-server/single-queue" example.

System description: A barbershop has one barber (Joe) and a certain finite number of chairs in the waiting area. Customers enter the shop with their interarrival times being exponentially distributed (which is equivalent to a Poisson distribution of the number of clients entering the shop per time unit). If Joe is idle, the newly arriving client will be served immediately, otherwise he joins the waiting queue which is ranked first-in first-out (FIFO). The service time for all clients is uniformally distributed.

This situation can be modeled by using an event-oriented approach. Two types of time-events can be distinguished.

a) Arrival of a new customer. Fig. 9.2 shows the flow-chart for the activities which are involved with this type of event.

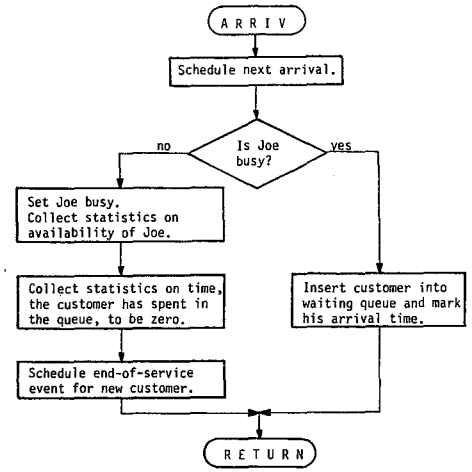

Fig. 9.2: Flow-chart of the arrival event.

An arriving customer has to check whether the server (Joe) is idle or busy. If he is idle, the customer can be served immediately, otherwise he has to join a waiting queue.

The arrival of customers is modeled by letting each arriving customer initiate the arrival of the next customer. This is coded at the very beginning of the event description since it has, in principle, nothing to do with the logical behaviour of the arriving customer, and, therefore, becomes easily lost. The first arrival is scheduled in the INITIAL block.

b) The end-of-service event. Fig. 9.3 shows the appropriate flow-chart for this type of event.

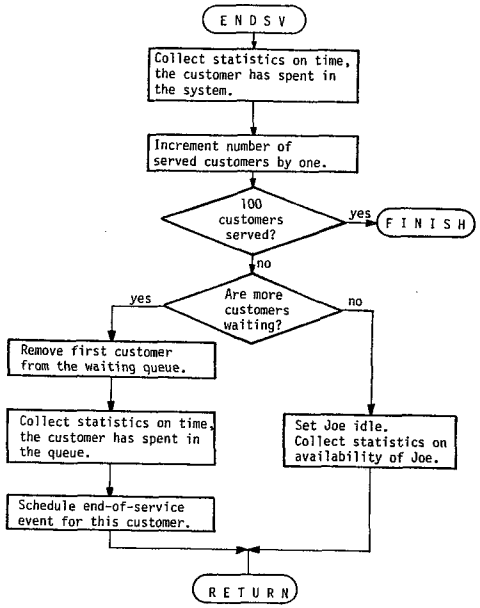

Fig. 9.3: Flow-chart of the end-of-service event.

A customer has been served and leaves the shop. Joe must now check whether another customer is waiting in the queue. In this case, the first customer can be removed from the queue, and a new end-of-service event can be scheduled. Otherwise, Joe can relax until the next client enters the shop.

Experiment description: Measurements have shown that the average inter-arrival time is 15 minutes, and that Joe requires between 12 to 18 minutes to serve a client. The simulation is supposed to go along until 100 clients have been served.

Output description: Statistics are to be collected with respect to

a) the average waiting time of the customers in the queue,

b) the average time each user spent in the shop and,

c) the utilization of Joe (equal to the percentage of time Joe was busy).

Fig. 9.4 shows a possible COSY program for this problem.

```
(*****************************************)
(*                                       *)
(*        DISCRETE SYSTEM SIMULATION     *)
(*             EVENT ORIENTED            *)
(*                                       *)
(*           JOE'S BARBERSHOP            *)
(*                                       *)
(*****************************************)

PROGRAM BARBERSHOP (INPUT,OUTPUT);

PROJECT 52 BY 'CELLIER';


CONST
  INTEGER  MAXCOUNT = 100;


TYPE
  ENTRIES = RECORD REAL EVTIME; TEVENT EVCODE; REAL ARRIVTIME END;


VAR
  FILE OF 3 ENTRIES: EVENTFILE;
  FILE OF 50 ENTRIES RANKED FIFO: WAITQUEUE;
  REAL  TISYS, TIQUE;
  INTEGER  COUNT;
  BOOLEAN  BUSY;
  TEVENT  ARRIVAL, ENDSERVICE;
  RANDOM INTERARRT,SERVICETIM;


EXPERIMENT
  HISTOGRAM
    TIQUE: 30 CELLS  WIDTH = 2.0  FUPLIM = 0.0;
    TISYS: 20 CELLS  WIDTH = 3.0  FUPLIM = 13.0 END;
  COLLECT  TIQUE; TISYS END;
  TIMEPERS  BUSY  START = FALSE END;
  RANDOMGEN
    INTERARRT:  EXPOND  AVTIM = 15.0  SEED = 23577;
    SERVICETIM: UNIFORMD  MIN = 12.0  MAX = 18.0  SEED = 74511 END;
  SIMULATE FROM 0 TO FINISH
END  (* EXPERIMENT *);
```

```
SYSTEM

  INITIAL
    COUNT := 0;
    SCHEDULE ARRIVAL AT INTERARRT
  END  (* INITIAL *);

  DISCRETE

    EVENTS

      TEVENTS :

        ARRIVAL:
          (* SCHEDULE ARRIVAL OF NEXT CUSTOMER *)
          SCHEDULE ARRIVAL AT TIME + INTERARRT;
          IF  BUSY  THEN
            (* JOE IS BUSY -> PUT NEW CUSTOMER INTO WAITING QUEUE *)
            INSERT WAITQUEUE WHERE ARRIVTIME := TIME END
          ELSE
            IF  NOT BUSY  THEN
            BEGIN (* JOE IS IDLE -> NEW CUSTOMER IS SERVED IMMEDIATELY *)
              BUSY := TRUE;  (* SET JOE BUSY *)
              TIQUE := 0.0;
              (* SCHEDULE END-OF-SERVICE EVENT *)
              SCHEDULE ENDSERVICE AT TIME + SERVICETIM
                       WHERE ARRIVTIME := TIME END
            END
            ELSE ERROREXIT 1 $IMPOSSIBLE STATE OF BARBER$
        END  (* ARRIVAL EVENT *);

        ENDSERVICE:
          TISYS := TIME - ARRIVTIME;  (* TIME IN SYSTEM *)
          COUNT := COUNT + 1;
          IF  COUNT > MAXCOUNT  THEN FINISH;
          IF  ENTRYNR(WAITQUEUE) = 0  THEN
            BUSY := FALSE  (* SET JOE IDLE *)
          ELSE
          BEGIN  (* REMOVE FIRST WAITING CUSTOMER FROM QUEUE *)
            REMOVE WAITQUEUE;
            TIQUE := TIME - ARRIVTIME;  (* TIME IN QUEUE *)
            (* SCHEDULE END-OF-SERVICE EVENT
               FOR NEWLY SERVED CUSTOMER *)
            SCHEDULE ENDSERVICE AT TIME + SERVICETIM
          END
        END  (* END-OF-SERVICE EVENT *)

      END  (* TIME EVENTS DESCRIPTION *)

    END  (* EVENTS DESCRIPTION *)

  END  (* DISCRETE SUBSYSTEM *)

END  (* SYSTEM DESCRIPTION *)

END .
```

Fig. 9.4: COSY program for Joe's barbershop (event-oriented)

IX.3.3) Discrete Simulation (Process-Oriented) -- Joe's Barbershop:

Let us discuss the same problem once more. This time, however, a process-oriented approach will be used. This approach will lead to a much shorter COSY program since the language elements used are highly aggregated. This is a more natural way of describing the system, although one may find the flow of the simulation program somewhat less transparent.

In this approach, the barbershop is modeled as a DISCRETE PROCESS in which a maximum of 51 transactions (process instances) can reside simultaneously. These are the customers of the shop. Each time, a new customer enters the barbershop, he passes through the GENERATE statement which causes a new arrival to take place after some time. Then he passes through the SEIZE statement in which he tries to allocate the FACILITY Joe. If he is successful, he passes immediately to the next statement, otherwise he is detained in an automatically administered waiting queue associated with the FACILITY Joe until he can be served. The ADVANCE statement detains the customer for the service time. Then the FACILITY Joe is RELEASEd, and passing through the END statement, the customer, finally, leaves the shop.

Notice the different semantics of a GENERATE statement in COSY as compared to GPSS-V [9.15]. In GPSS-V, a GENERATE block is a source of transactions which enter the system at the GENERATE statement in stochastic time intervals. The GENERATE block is the source-node for transactions. The TERMINATE block is, accordingly, the sink-node for the transactions. In COSY, the source-nodes and the sink-nodes are the PROCESS statement and the END statement, resp.. The GENERATE statement can appear anywhere in the PROCESS description, and causes precisely one transaction to enter the assigned PROCESS (either immediately or later) each time a

transaction passes through it. Also, a TERMINATE statement exists in COSY. This is infrequently required to take another transaction of the same, or of another, PROCESS out of the system prior to its reaching the END statement.

Fig. 9.5 shows a possible COSY program for solving this problem.

```
(*******************************************)
(*                                         *)
(*        DISCRETE SYSTEM SIMULATION       *)
(*             PROCESS ORIENTED            *)
(*                                         *)
(*             JOE'S BARBERSHOP            *)
(*                                         *)
(*******************************************)

PROGRAM BARBERSHOP (INPUT,OUTPUT);

PROJECT 53 BY 'CELLIER';


CONST
  INTEGER  MAXCOUNT = 100;

TYPE
  CUSTOMERS = RECORD INTEGER TRANSNBR, BLOCKNBR, PRIORNBR;
                     REAL GENTIM, MARKTIME END;

VAR
  FILE OF 51 CUSTOMERS RANKED FIFO: SHOPFILE;
  INTEGER  COUNT, CUSTOMER;
  REAL TIQUE, TISYS;
  BOOLEAN  BUSY;
  PROCESS  BARBERSHOP  FILE = SHOPFILE;
  RANDOM   INTERARRT, SERVICETIM;
```



```
DISCRETE PROCESS BARBERSHOP (VAR INTEGER COUNT; INTEGER MAXCOUNT;
                             VAR RANDOM INTERARRT, SERVICETIM;
                             VAR REAL COLLECT HISTOGRAM TIQUE, TISYS;
                             VAR BOOLEAN TIMEPERS BUSY;
                             VAR FILE SHOPFILE);

  TYPE
    CUSTOMERS = RECORD INTEGER TRANSNBR, BLOCKNBR, PRIORNBR;
                       REAL GENTIM, MARKTIME END;

  VAR
    FILE OF 51 CUSTOMERS RANKED FIFO: SHOPFILE;
    INTEGER  CUSTOMER;
    FACILITY  JOE;

  BEGIN
    (* CAUSE NEXT CUSTOMER TO ENTER BARBERSHOP *)
    GENERATE CUSTOMER FOR BARBERSHOP AT TIME + INTERARRT;
    SEIZE JOE;  (* SEIZE JOE IF HE IS IDLE, ELSE WAIT IN QUEUE *)
    TIQUE := TIME - SHOPFILE.TRANSNBR.GENTIM;  (* TIME IN QUEUE *)
    BUSY := TRUE;
    ADVANCE SERVICETIM;  (* SERVICE TIME IF JOE IS IDLE *)
    RELEASE JOE;  (* SET JOE IDLE *)
    BUSY := FALSE;
    TISYS := TIME - SHOPFILE.TRANSNBR.GENTIM;  (* TIME IN SYSTEM *)
    COUNT := COUNT + 1;  (* CUSTOMER COUNT *)
    IF  COUNT > MAXCOUNT  THEN FINISH   (* END SIMULATION *)

END  (* DESCRIPTION OF THE DISCRETE PROCESS BARBERSHOP *);


EXPERIMENT
  HISTOGRAM
    TIQUE: 30 CELLS  WIDTH = 2.0  FUPLIM = 0.0;
    TISYS: 20 CELLS  WIDTH = 3.0  FUPLIM = 13.0 END;
  COLLECT  TIQUE; TISYS END;
  TIMEPERS  BUSY  START = FALSE END;
  RANDOMGEN
    INTERARRT:  EXPOND  AVTIM = 15.0  SEED = 23577;
    SERVICETIM:  UNIFORMD  MIN = 12.0  MAX = 18.0  SEED = 74501 END;
  SIMULATE FROM 0 TO FINISH
END  (* EXPERIMENT BLOCK *);


SYSTEM

  INITIAL
    COUNT := 0;
    (* CAUSE FIRST CUSTOMER TO ENTER BARBERSHOP *)
    GENERATE CUSTOMER FOR BARBERSHOP AT INTERARRT
  END  (* INITIAL *);

  DISCRETE

    PROCESSES

      BARBERSHOP (COUNT, MAXCOUNT, INTERARRT, SERVICETIM, TIQUE,
                  TISYS, BUSY, SHOPFILE)

    END  (* PROCESS DESCRIPTION *)

  END  (* DISCRETE SUBSYSTEM *)

END  (* SYSTEM DESCRIPTION *)

END .
```

Fig. 9.5: COSY program for barbershop (process-oriented)

IX.3.4) Combined Simulation -- Pilot Ejection Study:

This is a commonly cited benchmark problem for "continuous" simulation. According to our new terminology, it belongs to the class of combined problems.

System description: The pilot ejection system, when activated, causes the pilot and his seat to travel along rails at a specified exit velocity VE at an angle THETAR (measured in radians) or THETAD (specified in degrees) backward from vertical. After traveling a vertical distance Y1, the seat becomes disengaged from its mounting rails and, at this point, the pilot is considered out of the cockpit. When this occurs, a second phase of operation begins during which the pilot trajectory is influenced by the force of gravity and atmospheric drag.

This problem belongs to the class of combined systems since the model is discontinuous at the moment when the ejector seat is disengaged from its mounting rails. Even the number of state equations is time dependent. The system is of second order (NNEQD := 2) during the first phase, but it is of fourth order (NNEQD := 4) during the second phase of the simulation run. During the first phase, there exists a state-condition to establish the condition when to branch to the second phase. During the second phase, this state-condition is no longer active.

Fig. 9.6a and Fig. 9.6b show a graphical description of the pilot ejection model and of the trajectories.

Pilot Ejection Study: Geometry

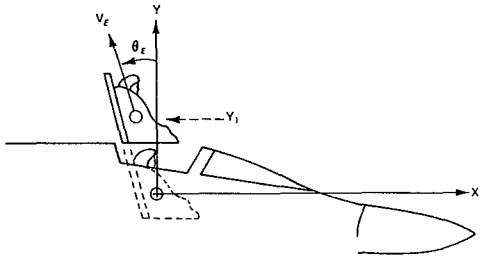

Fig. 9.6a: Pilot ejection study: Geometry.

Pilot Ejection Study: Trajectory

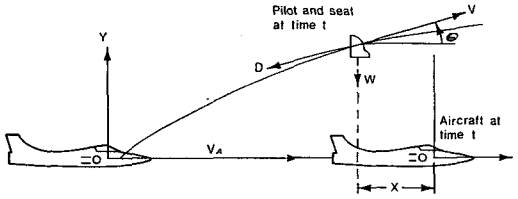

Fig. 9.6b: Pilot ejection study: Trajectory.

Fig. 9.7 defines the variables which are used in the model.

Variable	Definition	Units	Value
CD	Coefficient	-	1.0
D	Drag	$kg*m/s^2$	comp.
G	Gravity	m/s^2	9.81
H	Height above sea-level	m	par.
M	Mass of pilot and seat	kg	150.0
RHO	Air-density	kg/m^3	table
S	Active surface exposed to the wind	m^2	1.0
THETAD	Angle of ejection	deg	15.0
THETAR	Angle of ejection	rad	comp.
THETAS	Angle for pilot and seat movement	rad	state
V	Pilot and seat velocity	m/s	state
VA	Aircraft velocity	m/s	par.
VE	Ejection velocity	m/s	18.0
X	Horizontal distance from point of ejection relative to the movement of aircraft	m	state
Y	Vertical distance from ejection point	m	state
Y1	Vertical distance above ejection point where first phase terminates	m	1.2

Fig. 9.7: Variables of the pilot ejection model.

Experiment description: The experiment is carried out to analyse how large the maximum velocity of the aircraft (VA) may be, as a function of the height above sealevel (H), in

order to allow for a secure ejection. An ejection is said to be "secure" if the ejection seat clears the vertical stabilizer of the aircraft which is 9m behind and 3.5m above the cockpit in a distance of at least 2.5m. For this purpose, a first simulation run is carried out with a small velocity (VA := 30.0) at ground level (H := 0.0). If this ejection is successful (according to our rules (!)), the velocity is increased by 15.0 m/s, and the experiment is repeated. If the ejection is not successful, the height is increased by 150.0 m, and the experiment is repeated. In this way we proceed until either the velocity has reached a value of 270.0 m/s or until the height has reached a value of 16500.0 m above sealevel, which ever occurs first. The experiment cannot start at zero velocity since, for this velocity, the specified model is not valid.

Usually, the simulation is terminated by a CSSL-type FINISH-statement

```
FINISH X = -9.0;
```

to indicate that the critical portion of the pilot trajectory is over. However, in this way, the simulation is continued over the last integration step and may end at a much more negative value of X. Since we are interested in knowing the value of Y at X = -9.0 to decide upon success or failure of the ejection, we must compute this value with some accuracy. For this reason, a continuous simulation language will have to restrict the integration step-size artificially whereas, in a combined system simulation language, we can replace the FINISH-condition by another state-CONDITion to locate the critical point, X = -9.0, precisely. The associated state-event can then be used to terminate the simulation run.

This specific experiment description has been taken from [9.8], except that all data have been converted to metric

units.

Output description: After each successful ejection, the actual values of the ejection level (H), and of the aircraft velocity (VA), are to be stored for later graphical representation of H as a function of VA. During the last run, we want, furthermore, to store X and Y. These values are to be plotted versus time.

Fig. 9.8 shows the listing of a possible COSY program for this benchmark problem.


```
{*****************************************}
{*                                       *}
{*        COMBINED SYSTEM SIMULATION     *}
{*                                       *}
{*           PILOT EJECTION STUDY        *}
{*                                       *}
{*****************************************}

PROGRAM PILOTEJECT (INPUT,OUTPUT);

PROJECT 56 BY SCELLIER;


MACRO ROTATE (X, Y <- V, THETA, CONST FLAG);
  MACIF #FLAG# = 1 THEN
  MACBEGIN
    X = V*COS (THETA);
    Y = V*SIN (THETA)
  MACEND
  MACELSE
  MACBEGIN
    X := V*COS (THETA);
    Y := V*SIN (THETA)
  MACEND
MACEND  (* ROTATE *);


LABEL
  100;


CONST
  REAL
    VE      = 12.0    (* M/S *),
    M       = 100.0   (* KG *),
    Y1      = 1.2     (* M *),
    THETAD  = 15.0    (* DEGREES *),
    CD      = 1.0     (* --- *),
    S       = 1.0     (* M*M *),
    G       = 9.81    (* M/(S*S) *);


VAR
  REAL  DIST, H, HELP, RHO1, THETAR, VA, VECOS, VESIN;
  STATE  X, Y, V, THETAS;
  ALGEBR  D, VX, VY, GX, GY;
  INTEGER  PHASE;
  BOOLEAN  LAST;
  SEVENT  DISENGAGE, OVER;
  MODEL  INOROUT;


STORE
  H, VA, X, Y;


FUNCTABLE
  SPLINE RHO = (    0.0 : 1.293) (  300.0 : 1.255) (  600.0 : 1.220)
               ( 1200.0 : 1.152) ( 1800.0 : 1.082) ( 3000.0 : 0.955)
               ( 4500.0 : 0.815) ( 6000.0 : 0.675) ( 9000.0 : 0.476)
               (12000.0 : 0.319) (15000.0 : 0.196) (18000.0 : 0.122);


EXPERIMENT
  INITCOND  X = 0.0,  Y = 0.0;
  LAST := FALSE;  STOREOFF;
  THETAR := THETAD/57.2957795;
  ROTATE (VECOS, VESIN <- VE, THETAR, 2);
  H := 0.0;  VA := 31.0;
  SIMULATE  FROM 0.0 TO FINISH  COMINT = 0.2
END  (* EXPERIMENT BLOCK *);
```

```
SYSTEM

  INITIAL
    HELP := VA - VESIN;
    V := SQRT (HELP*HELP + VECOS*VECOS);
    THETAS := ATAN (VECOS/HELP);
    RHO1 := 0.5*CD*S*RHO (H);
    PHASE := 1;  NNEQD := 2
  END  (* INITIAL SECTION *);

  CONTINUOUS
    ROTATE (VX, VY <- V, THETAS, 1);
    X' = VX - VA;  Y' = VY;
    MODEL INDROUT (GX,GY,D <-);
      CASE PHASE OF
      1:   CONDIT DISENGAGE: Y CROSSES YI POS TOL=1.0E-3 END
           END;
      2:   ROTATE (GX, GY <- G, THETAS, 1);
           V' = -D/M - GY;
           THETAS' = -GX/V;
           D = RHO1*V*V;
           CONDIT OVER: X CROSSES -9.0 NEG TOL=1.0E-3 END
           END
    END  (* MODEL IN-OR-OUT *)
  END  (* CONTINUOUS SUBSYSTEM *);

  DISCRETE
    EVENTS
      DISENGAGE:  PHASE := 2;  NNEQD := 4  END;
      OVER:  FINISH  END
    END  (* STATE-EVENT DESCRIPTION *)
  END  (* DISCRETE SUBSYSTEM *);

  TERMINAL
    IF  NOT LAST  THEN
    BEGIN
      IF  ((H <= 16500.0) AND (VA <= 270.0))  THEN
      BEGIN
        IF  Y > 6.0  THEN
        BEGIN
          DIST := Y - 2.5;
          FORTRAN
            WRITE (OUTPUT,100)  X, Y, DIST
       100 FORMAT (' X = ',E12.4,' Y = ',E12.4,' DIST = ',E12.4)
          END  (* FORTRAN *);
          CROSSPLOT;  VA := VA + 15.0
        END ELSE
          H := H + 150.0
      END ELSE
      BEGIN
        LAST := TRUE;  STOREON
      END;
      RERUN
    END
  END  (* TERMINAL SECTION *)
END  (* SYSTEM DESCRIPTION *);


OUTPUT
  TITLE '$PILOT EJECTION STUDY$';
  LIST (CROSSF): VA, H;
  PLOT (CROSSF) VERSUS VA: H;
  FACTOR  XFAK = 2.0  YFAK = 2.0;
  GRAPH (CROSSF) VERSUS VA: H;
  GRAPH (TIMEF): X, Y
END  (* OUTPUT BLOCK *)

END .
```

Fig. 9.8: COSY program for the pilot ejection problem.

Fig. 9.9 shows the resulting graph of the ejection level depicted versus the aircraft velocity.

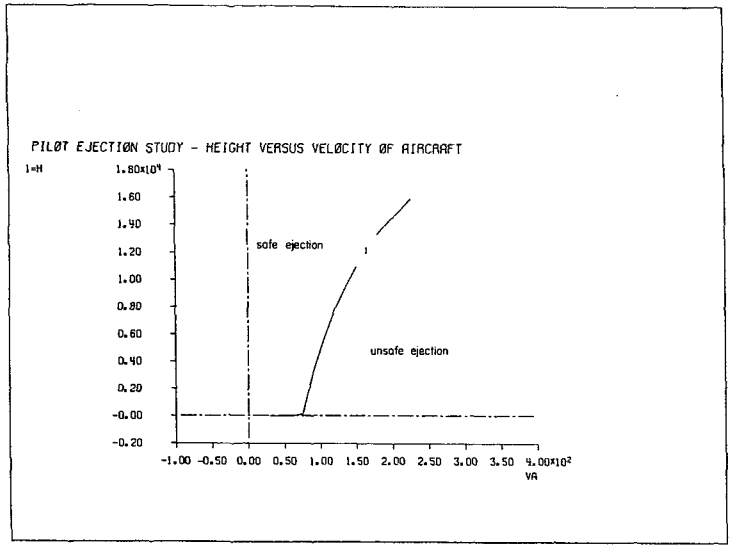

Fig. 9.9: Safe and unsafe ejection.

IX.3.5) Combined Simulation -- SCR Control Problem:

This problem requires no further explanation since it has already been discussed in chapter III when we presented the GASP solution to this problem. Fig. 9.10 shows the COSY program for the SCR control problem. In fact, this COSY program may be used to generate the GASP-V program of Fig. 3.4.

```
(*****************************************)
(*                                       *)
(*        COMBINED SYSTEM SIMULATION     *)
(*                                       *)
(*           SCR CONTROL PROBLEM         *)
(*                                       *)
(*****************************************)

PROGRAM SCR (INPUT,OUTPUT);

PROJECT 58 BY MCELLIERS;


CONST
  REAL   UE = 20.0, ECTR = 5.0, ECXS = 10.0, XL1 = 2.0,
         ZN2 = 0.075, VE = 1.0, US = 1.0, BT = 200.0,
         W = 1.0;


VAR
  ALGEBR  AZ, DT, ILDOT, T, UF, UFS, UL, ULS, UZS;
  STATE   IL, ISP, SS5, UC, UZ;
  REAL    C1, PI, ULD, WF, X, XCS, XCSP, XL, XN, XS, XSI,
          XSP, YP;
  INTEGER  NMOD;
  SEVENT   CONDUCT, NONCONDUCT;
  MODEL    CHOPPER;


STORE
  T, IL, ULS, UFS, UES, DT;


EXPERIMENT
  INITCOND  IL = 0.0, UC = 1400.0, UZ = 1200.0, SS5 = 0.0;
  INTEGRAT  METHOD RKS;
  SIMULATE  FROM 0.0 TO 3.0 COMINT = 0.02
END  (* EXPERIMENT *);


SYSTEM

  INITIAL
    (* DEFINITION OF THE MATHEMATICAL NUMBER PI *)
    PI := 4.0*ATAN (1.0);
    (* DEFINITION OF AUXILIARY VARIABLES *)
    WF := 100.0*PI/3.0;
    XL := XL1/(UE*UE);
    XSI := ZN2*ECTR/100.0;
    XN := XL + XSI;
    XS := ZN2*ECXS/100.0;
    X := XN + XS;
    XCSP := 1.0/(0.024868*WF);
    XSP := XCSP/(4.0*VE);
    XCS := 1.0/(0.03*WF);
    ULD := 750.0*US*SQRT (2.0);
    YP := 7.5E+6 * 2.0/ULD;
    C1 := (YP + BT/2.0)*ULD/(2.0*UZ);
    (* COMPUTED INITIAL CONDITION FOR ISP *)
    ISP := - C1;
    (* SELECT INITIAL MODEL *)
    NMOD := 1
  END  (* INITIAL *);
```



```
  CONTINJOUS
    (* COMPUTATION OF LINE VOLTAGE UL *)
    UL = ULD*SIN (W*TIME);
    (* STATE SPACE DESCRIPTION *)
    ILDOT = (1.0/X)*(UL - AZ*UZ);
    IL' = ILDOT;
    ISP' = (1.0/XSP)*(UZ - UC);
    UC' = XCSP*ISP;
    UZ' = XCS*(AZ*IL - ISP - C1);
    (* ADDITIONAL STATE VARIABLE TO ENABLE INVERSE HERMITE' INTERPOLATION *)
    SS5' = ILDOT - YP*COS (W*TIME);

    MODEL CHOPPER (AZ <-);
      CASE NMOD OF
      1:   AZ = 0.0;
           CONDIT NONCONDUCT: IHERMITE SS5-BT POS TOL=1.0E-13 END
         END;
      2:   AZ = 1.0;
           CONDIT CONDUCT: IHERMITE SS5+BT NEG TOL=1.0E-13 END
         END
    END  (* CHOPPER *);

    COMMUNIC DATASTORE (<- UL, UZ, ILDOT);
      (* STORE DATA FOR LATER OUTPUT *)
      T := 30.0*TIME/PI;  DT := DTFUL;
      ULS := 10.0*UL;  UZS := 10.0*UZ;
      UF := UL - ILDOT*XL;  UFS := 10.0*UF
    END  (* DATA STORAGE AT COMMUNICATION TIME *)

  END  (* CONTINUOUS SUBSYSTEM *);

  DISCRETE

    EVENTS
      CONDUCT: NMOD := 1 END;
      NONCONDUCT: NMOD := 2 END
    END  (* STATE EVENTS DESCRIPTION *)

  END  (* DISCRETE SUBSYSTEM *)

END  (* SYSTEM DESCRIPTION *);


OUTPUT
  TITLE $SCR - CONTROL CIRCUIT (COMBINED MODELING TECHNIQUE)$;
  FACTOR  XFAK = 2.0  YFAK = 2.0;
  GRAPH VERSUS T: IL, ULS, UFS, UZS;
  TITLE $DT SIZE (COMBINED MODELING TECHNIQUE)$;
  GRAPH VERSUS T: DT
END  (* OUTPUT *)

END .
```

Fig. 9.10: COSY program for the SCR control problem.

If one compares the COSY program of Fig. 9.10 with the CSMP program of Fig. 3.5, one notes that the COSY program is not significantly more difficult to code than the CSMP program. In fact, both programs look quite similar. However, since the COSY program is translated into GASP-V executable code, it will run correctly and as efficiently as the GASP-V program of Fig. 3.4, except for the (neglectable) compilation time required for the generation of the GASP-V program.

IX.3.6) Combined Simulation (Variable Structures) -- DOMINO Game:

Statement of the problem: This is a new benchmark problem which can be used to test the capability of a simulation language to digest variable structures. A variable structure simulation is defined as any combined simulation in which the number of differential and/or difference equations varies with time. According to this definition, even the simple pilot ejection study belongs to this class of problems. However, typical examples of variable structure systems are traffic simulation studies with a variable number of cars in the system, the steel soaking pit and slabbing mill [9.6,9.7], or the chemical batch process [9.12]. A new example of this class is the DOMINO game as will be illustrated. This may, in the future, be used as a new benchmark problem to evaluate the capability of a combined simulation language to cope with variable structures.

System description: All (identical) stones of the DOMINO game (NBRSTONES := 55) are placed in a sequence with a distance of D space units between any two consecutive stones. If the first stone is pushed, all stones fall flat. Each stone has one degree of freedom to move. According to Newton's law, it is described by a second order differential equation. The overall order of the system is, therefore, 110. This is a typical k-out-of-n situation with

k stones simultaneously moving whereas (n-k) stones are not moving, either because they have fallen already or because they have not yet been touched. The number (k) depends on the distance (D) between stones.

This situation could easily be expressed as suggested in chapter VII:

```
FOR I:=1 TO NBRSTONES DO
  IF STONE[I] THEN
```

However, by these means, we must keep the attributes of all the stones in memory.

In GASP-V, the numerical integration is organized to proceed array-wise starting from index one of the DD-vector (denoting state derivatives). To let the numerical integration execute as efficiently as possible, we would, therefore, like to have the state derivatives of all currently active stones dense and always occupying the first elements of the DD-vector. This can be achieved by addressing the state variables indirectly through computed indices:

```
J = f(TNOW)
DD(J) = ...
```

By these means, we arrange for a garbage collection of state variables.

On the level of COSY programming, this procedure can be automated. For this task, we can define a CONTINUOUS PROCESS in which the behaviour of the single STONE is modeled. Several instances (copies) of the CONTINUOUS PROCESS can be active simultaneously with their individual attributes being stored in entries of a PROCESS FILE. This can be declared as follows:

```
CONST
  INTEGER  NBRSTONES = 55, SIMULT = 20;

TYPE
  STONEATTR = RECORD INTEGER COPYNBR, COPYSTAT, POINTDD,
              POINTSS, POINTGF; STATE PHI, OMEGA;
              REAL OMEGAS; INTEGER PHASE; SEVENT CRASH,
              DOWN END;

VAR
  FILE OF SIMULT STONEATTR RANKED LVF ON COPYNBR: STONEFILE;
  PROCESS STONE  FILE = STONEFILE;
```

Individual copies of CONTINUOUS PROCESSes can be CREATEd, SUSPENDed, RESUMEd, and DELETEd. A PROCESS FILE must accompany the declaration of each PROCESS. The PROCESS FILE is administered by the system automatically. Its first entry is called the "PROCESS entry". It has six standard attributes:

ATRIB(1) = 0.0
ATRIB(2) = number of attributes of the PROCESS FILE.
ATRIB(3) = number of differential equations of each PROCESS copy.
ATRIB(4) = number of difference equations of each PROCESS copy.
ATRIB(5) = number of GASP-functions used by the PROCESS.
ATRIB(6) = maximum number of entries.

Subsequent entries are used to store the attributes of one individual PROCESS copy each. Such entries have five standard attributes:

ATRIB(1) = copy number.
ATRIB(2) = copy status with the following meaning:
:=1 : INITIAL section of PROCESS
:=2 : CONTINUOUS section of PROCESS
:=3 : STATE-CONDITION section of PROCESS
:=4 : STATE-EVENT section of PROCESS
:=5 : copy is currently SUSPENDed
:=6 : TERMINAL section of PROCESS.
ATRIB(3) = pointer to section of state space used for differential equations of the PROCESS
ATRIB(4) = pointer to section of state space used for difference equations of the PROCESS
ATRIB(5) = pointer to section of unique identifiers used for the GASP-functions of the PROCESS.

Individual attributes may follow. The pointers will, normally, not be accessed by the user when programming in COSY.

In our example, each entry has 11 attributes (5 standard attributes and 6 individual attributes). 19 out of the 55 STONEs can be simultaneously active (SIMULT := 20).

As one can see, the attribute administration mechanisms are precisely the same as they are used in a discrete process-oriented simulation.

Simulation of a STONE involves two different PHASEs. In the first PHASE, a state-condition (CRASH) is active to determine the angle (PHIPUSH) at which the current STONE touches the next one. The associated state-event will CREATE a new STONE-copy if there is a copy left to be CREATEd. During execution of the second PHASE, another state-condition (DOWN) is active to determine the angle (PI/2.) at which the current STONE touches ground. The associated state-event DELETEs the current STONE from the system. In the TERMINAL section of the STONE PROCESS, we

check on the number of entries in the PROCESS FILE. If this is equal to two, we are obviously about to DELETE the last STONE from the system, and the simulation can terminate. It would be incorrect to check whether the actual copynumber (COPYNBR) is equal to the number of stones (NBRSTONES) since the last STONE does not CRASH into any further STONE, and may, thus, be DOWN earlier than some of the previous ones.

Fig. 9.11 shows a STONE of the DOMINO game.

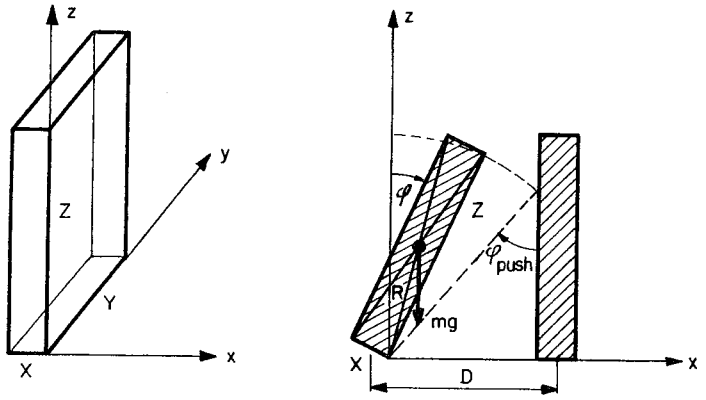

Fig. 9.11: STONE of the DOMINO game.

The motion of the STONEs is governed by Newton's law:

$$\text{THETA}*\text{OMEGA}' = T$$

where THETA denotes the inertia of the stone, and OMEGA denotes its angular velocity. The torque (T) can be computed as:

$$T = m*g*R*\sin(\phi)$$
$$R = 0.5*\text{SQRT}(X*X + Z*Z)$$

At the moment of the CRASH, the horizontal component of the impulse is transferred to the newly CREATEd STONE, whereas the vertical component remains with the previous STONE. It can be easily computed that the initial angular velocity of the CRASHed STONE (ω°_{k+1}) must be:

$$\omega^{\circ}_{k+1} = KO*\omega^{*-}_{k}$$

with: $KO = \cos^2(\phi_{push})$

where: $\phi_{push} = \sin^{-1}((D-X)/Z)$

ω^{*-}_{k} : angular velocity of previous STONE before CRASH.

The angular velocity of the CRASHing STONE is, thereby, reduced to:

$$\omega^{*+}_{k} = KR*\omega^{*-}_{k}$$

with: $KR = 1.0 - \cos(\phi_{push})$

Experiment description: The aim of the experiment is to determine the distance (D) between consecutive STONEs which maximizes the velocity of the chain. This constitutes a unidimensional nonlinear programming problem with two inequality constraints since the distance (D) must obviously lie in the interval

$$D \in [X, Z+X) \ .$$

To avoid execution of a (more expensive) constraint optimization, we use a modified optimization parameter:

$$\text{DMOD} = \tan[(PI/Z)*(D - Z/2. - X)]$$

corresponding to:

$$D = (Z/PI)*\tan^{-1}(\text{DMOD}) + Z/2. + X$$

```
 D  I        DMOD
----I------------------
 X  I negative infinite
Z+X I positive infinite
```

We want to utilize the precoded nonlinear programming package, NLP [9.13,9.14], which MINIMIZEs a performance index (F) with respect to one or several parameters, possibly subject to additional equality and/or inequality constraints. The performance index can be defined as:

$$F = 1.0/V$$

where V denotes the chain velocity, and can be expressed as:

$$V = D*(NBRSTONES-1)/ (TTFIN-TTBEG)$$

Output description: We want to produce a graphical representation of the chain velocity (V) with respect to the distance (D).

Fig. 9.12 shows a possible COSY program for this problem.


```
{*****************************************}
{*                                       *}
{*      COMBINED SYSTEM SIMULATION       *}
{*        VARIABLE STRUCTURES            *}
{*                                       *}
{*            DOMINO GAME                *}
{*                                       *}
{*****************************************}

PROGRAM DOMINO (INPUT,OUTPUT);

PROJECT 60 BY SCELLIERS;


LABEL
  100, 101;

CONST
  REAL
    G       = 9.81   (* M/(S*S) *),
    M       = 0.01   (* KG *),
    X       = 0.008  (* M *),
    Y       = 0.024  (* M *),
    Z       = 0.046  (* M *);
  INTEGER  NBRSTONES = 55, SIMULT = 32;

TYPE
  STONEATTR = RECORD INTEGER STONENBR, STONESTAT, POINTDD, POINTSS, POINTGF;
                     STATE PHIA, OMEGA;  REAL OMEGAS;  INTEGER PHASE;
                     SEVENT CRASH, DOWN  END;

VAR
  FILE OF SIMULT STONEATTR RANKED LVF ON STONENBR: STONEFILE;
  REAL  J, DMOD, HEIGHT, HELP1, HELP2, HELP3, IMP, K1, K2, K3, PHIPUSH,
        PI, PI2, PIE2A, R, STNRM, THETAA, V, XSQR, ZSQR;
  INTEGER  NEWSTONE;
  PROCESS  STONE  FILE = STONEFILE;

STORE
  D, V, STNRM;
```

```
CONTINUOUS PROCESS STONE (<- REAL HEIGHT, IMP, KM, KO, KR, PHIPUSH, PI, PI2,
                            THETAA; INTEGER NBRSTONES, SIMULT; VAR FILE SF);

  TYPE
    SA = RECORD INTEGER SN, STONESTAT, POINTDO, POINTSS, POINTGF;
                STATE PHIA, OMEGA;  REAL OMEGAS;  INTEGER PHASE;
                SEVENT CRASH, DOWN  END;

  VAR
    FILE OF SIMULT SA RANKED LVF ON SN: SF;
    ALGEBR  T;
    INTEGER  NEWSTONE;
    MODEL  TERMIN;

  BEGIN
    INITIAL
      WITH SF.SN DO
      BEGIN
        PHIA := 0.0;
        IF  SN = 1  THEN
          OMEGA := HEIGHT*IMP/THETAA
        ELSE
          OMEGA := KO*OMEGAS;
        PHASE := 1;  CRASH := 0;  DOWN := 0
      END  (* WITH STATEMENT *)
    END  (* INITIAL PART OF THE CONTINUOUS PROCESS STONE *);

    CONTINUOUS
      WITH SF.SN DO
      BEGIN
        T = KM*SIN (PHIA);
        OMEGA' = T/THETAA;  PHIA' = OMEGA;
        MODEL  TERMIN (<- PHIA);
          CASE PHASE OF
          1:   CONDIT CRASH: PHIA CROSSES PHIPUSH POS  TOL = 1.0E-3 END
            END;
          2:   CONDIT DOWN: PHIA CROSSES PI2 POS  TOL = 1.0E-3 END
            END
        END (* MODEL TERMIN *)
      END  (* WITH STATEMENT *)
    END  (* CONTINUOUS PART OF THE CONTINUOUS PROCESS STONE *);

    DISCRETE
      CRASH:
        WITH SF.SN D)
        BEGIN
          PHASE := 2;
          IF  SN < NBRSTONES  THEN
          BEGIN
            CREATE STONE(NEWSTONE) WHERE OMEGAS := OMEGA END;
            OMEGA := KR*OMEGA
          END
        END (* WITH STATEMENT *)
      END  (* CRASH *);
      DOWN: DELETE SF.SN  END
    END  (* DISCRETE PART OF THE CONTINUOUS PROCESS STONE *);

    TERMINAL
      IF  ENTRYNBR(SF) <= 2  THEN FINISH
    END  (* TERMINAL PART OF THE CONTINUOUS PROCESS STONE *)

END  (* DESCRIPTION OF THE CONTINUOUS PROCESS STONE *);
```



```
EXPERIMENT
  PI := 4.0*ATAN (1.0); PI2 := PI/2.0;  HEIGHT := Z/2.0;  IMP := 1.0E-5;
  XSQR := X*X;  ZSQR := Z*Z;  HELP1 := XSQR + ZSQR;
  THETAA := M*HELP1/3.0;  R := 0.5*SQRT (HELP1);  KM := M*G*R;
  MINIMIZE
    PIEVA;  DMOD := 1.0;  TOL = 1.0E-4 END;
    SIMULATE  FROM 0.0 TO FINISH;
    PIEVA := 1.0/V
  END  (* OPTIMIZATION *);
  D := (Z/PI)*ATAN (DMOD) + Z/2.0 + X;
  V := 1.0/PIEVA;
  FORTRAN
    WRITE (OUTPUT,101)  V, D
 101 FORMAT ($1THE CHAIN VELOCITY TAKES ITS MAXIMAL VALUE OF$,E12.4,
/                $ M/SEC/$ AT A DISTANCE OF$,E12.4,$ M  BETWEEN TWO STONES.$)
  END  (* FORTRAN *)
END  (* EXPERIMENT BLOCK *);


SYSTEM

  INITIAL
    HELP2 := D - X;  HELP3 := ZSQR - HELP2*HELP2;  KO := HELP3/ZSQR;
    PHIPUSH := ASIN (HELP2/Z);  KR := 1.0 - SQRT (KO);
    D := (Z/PI)*ATAN (DMOD) + Z/2.0 + X;
    CREATE STONE(NEWSTONE)
  END  (* INITIAL SECTION *);

  CONTINUOUS
    STONE (<- HEIGHT, IMP, KM, KO, KR, PHIPUSH, PI, PI2, THETAA,
              NBRSTONES, SIMULT, STONEFILE);
  END  (* CONTINUOUS SUBSYSTEM *);

  TERMINAL
    V := D*FLOAT (NBRSTONES-1)/TIME;
    STNRM := FLOAT (#MAXQ(STONE) - 1)/50.0;
    CROSSPLOT;
    FORTRAN
      WRITE (OUTPUT,100)  D, V, STNRM
  100 FORMAT ($ D =$,E12.4,$ V =$,E12.4,$ STNRM =$,E12.4)
    END  (* FORTRAN *)
  END  (* TERMINAL SECTION *)

END  (* SYSTEM DESCRIPTION *);


OUTPUT
  TITLE  $CHAIN VELOCITY AND MAX. NR. OF STONES VERSUS DENSITY$;
  FACTOR  XFAK = 2.0  YFAK = 2.0;
  GRAPH (CROSSF) VERSUS D: V, STNRM
END  (* OUTPUT BLOCK *)

END .
```

Fig. 9.12: COSY program for the DOMINO game.

Fig. 9.13 shows the GASP-V program which is produced by the COSY preprocessor.

```
*CHAIN VELOCITY AND MAX. NR. OF STONES VERSUS DENSITY
FACTOR(,2.0,2.0)
GRAPHXY(CROSS)D,V,STNRM
END
```

```
      PROGRAM MAIN (INPUT,OUTPUT,MONITR,TIME,CROSS,SAVE,TAPE1=MONITR,
     1TAPE2=TIME,TAPE3=CROSS,TAPE4=SAVE,TAPE5=INPUT,TAPE6=OUTPUT)
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLP,TTFIN,TRIB(25),TTSET
      COMMON /NLP/ NP,PA(40),FMIN,FTOL,IDER,IOU,NFE,FETOL(40),NFI,FITOL(
     140),FAC,IND,FRES,FERES(40),FIRES(40)
      EXTERNAL  PIEVA
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     2E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      COMMON QSET(416)
      NCRDR = 5
      NPRNT = 6
C
C*****CONSTANTS
C
      G = 9.81
      M = 0.01
      X = 0.008
      Y = 0.024
      Z = 0.046
      NBRSTO = 55
      SIMULT = 32
C
C*****EXPERIMENT
C
      PI = 4.0*ATAN (1.0)
      HEIGHT = Z/2.0
      IMP = 1.0E-5
      XSQR = X*X
      ZSQR = Z*Z
      HELP1 = XSQR + ZSQR
      THETA = M*HELP1/3.0
      R = 0.5*SQRT (HE_PI)
      KM = M*G*R
C
C*****MINIMIZATION
C
      NP = 1
      PA(1) = 0.0
      FMIN = 1.0E10
      FTOL = 1.0E-4
      IDER = 0
      IOU = 3
      NFE = 0
      NFI = 0
      FAC = 0.1
      CALL  NLP (PIEVA)
      CALL  SUMRY
      IF (IND.NE.0)  WRITE (NPRNT,10013)  IND
10013 FORMAT (34H0*****ERROR IN OPTIMIZATION= IND =,I5)
C
      D = (Z/PI)*ATAN (PA(1)) + (Z/2.0) + X
      V = 1.0/FRES
      WRITE (NPRNT,101)  V, D
  101 FORMAT (46H1THE CHAIN VELOCITY TAKES ITS MAXIMUM VALUE OF,
     1E12.4,4H M/S/17H AT A DISTANCE OF,E12.4,
     223H M  BETWEEN TWO STONES.)
C
      CALL  BYE
      END
```



```
      SUBROUTINE  PIEVA
      COMMON /FUNCT/ IFUN,P(40),F,FEI(40),DF(40),DFEI(1600),IERR
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     2E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      CALL  GASP
      F = 1.0/V
      RETURN
      END
```

```
      SUBROUTINE  INFLC
      COMMON /FUNCT/ IFUN,P(40),F,FEI(40),DF(40),DFEI(1600),IERR
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     2E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      DIMENSION ATR(20)
      HELP2 = D - X
      HELP3 = ZSQR - HELP2*HELP2
      KO = HELP3/ZSQR
      PHIPUS = ASIN (HELP2/Z)
      KR = 1.0 - SQRT (KO)
      D = (Z/PI)*ATAN (P(1)) + (Z/2.0) + X
      XSTONE = INITM (SIMULT, 2, 0, 4, 0)
      NEWSTO = NEW (XSTONE)
      CALL  CREAT (XSTONE, NEWSTO, ATR)
      RETURN
      END
```

```
      SUBROUTINE  STATE
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     2E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      CALL  GMODL (XSTONE)
      RETURN
      END
```

```
      SUBROUTINE  MODEL (NMOD, NCOP, NSTAT)
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     1E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      CALL  STONE (HEIGHT, IMP, KM, KO, KR, PHIPUS, PI, THETA, NBRSTO)
      RETURN
      END
```

```
      SUBROUTINE  STONE (HEIGHT,IMP,KM,KO,KR,PHIPUS,PI,THETA,NBRSTO)
      REAL IMP,KM,KO,KR
      INTEGER PHASE,CRASH,DOWN
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM2/ DD(100),DDL(100),DTFUL,DTNOW,ISEES,LFLAG(50),NFLAG,
     1NNEQD,NNEQS,NNEQT,SS(100),SSL(100),TTNEX
      COMMON /MCOM1/ JJDD,JJGF,JJSS,KKATR,MMDOP,NNGOP,NNDD,NNGF,NNMOD,NN
     1SS,NNSTT,NNZZZ(11)
      DIMENSION IIZZZ(11)
      EQUIVALENCE (IIZZZ(1),JJDD)
      DIMENSION ATR(20)
      GO TO (10000,10001,10002,10003,10004,10005), NNSTT
C
C*****INITIAL SECTION
C
10000 CONTINUE
      SS(JJDD+1) = 0.0
      IF (NNDOP-1) 10007,10006,10007
10006 CONTINUE
      SS(JJDD+2) = HEIGHT*IMP/THETA
      GO TO 10008
10007 CONTINUE
      SS(JJDD+2) = KO*ATRIB(8)
10008 CONTINUE
      PHASE = 1
      ATRIB(9) = FLOAT (PHASE)
      CRASH = 0
      ATRIB(10) = FLOAT (CRASH)
      DOWN = 0
      ATRIB(11) = FLOAT (DOWN)
      RETURN
C
C*****CONTINUOUS SECTION
C
10001 CONTINUE
      T = KM*SIN (SS(JJDD+1))
      DD(JJDD+2) = T/THETA
      DD(JJDD+1) = SS(JJDD+2)
      RETURN
C
C*****STATE CONDITION SECTION
C
10002 CONTINUE
      PHASE = IFIX (ATRIB(9))
      GO TO (10009,10010), PHASE
10009 CONTINUE
      CRASH = KROSS (JJDD+1, 0, 0.0, PHIPUS, +1, 1.0E-3)
      ATRIB(10) = FLOAT (CRASH)
      RETURN
10010 CONTINUE
      DOWN = KROSS (JJDD+1, 0, 0.0, PI/2.0, +1, 1.0E-3)
      ATRIB(11) = FLOAT (DOWN)
      RETURN
C
C*****DISCRETE SECTION
C
10003 CONTINUE
      CRASH = IFIX (ATRIB(10))
      IF (CRASH.EQ.0)  GO TO 10011
      ATRIB(10) = 0.0
      PHASE = 2
      ATRIB(9) = FLOAT (PHASE)
      IF (NNDOP.GE.NBRSTO)  GO TO 10012
      NEWSTO = NEW (NNMOD)
      ATR(3) = SS(JJDD+2)
      CALL  CREAT (NNMOD, NEWSTO, ATR)
      SS(JJDD+2) = KR*SS(JJDD+2)
10012 CONTINUE
      RETURN
10011 CONTINUE
      DOWN = IFIX (ATRIB(11))
      IF (DOWN.EQ.0)  RETURN
      ATRIB(11) = 0.0
      CALL  DLETE (NNMOD, NNDOP, ATR)
      RETURN
C
C*****PASSIVE SECTION
C
10004 CONTINUE
      RETURN
C
C*****TERMINAL SECTION
C
10005 CONTINUE
      IF (NNQ(NNMOD).LE.2)  CALL FINIS
      RETURN
      END
```



```
      SUBROUTINE  OTPUT
      COMMON /GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,NCRDR,N
     1NAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,PPARM(50,4),TNOW,TTBEG
     2,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON /GCOM4/ DTPLT(10),HHLOW(25),HHWID(25),IICRD,IITAP(10),JJCEL
     1(500),LLABC(25,2),LLABH(25,2),LLABP(11,2),LLABT(25,2),LLPHI(10),LL
     2PLO(10),LLPLT,LSUP(15),LLSYM(10),MMPTS,NNCEL(25),NNCLT,NNHIS,NNPL
     3T,NNPTS(10),NNSTA,NNVAR(10),PPHI(10),PPLO(10)
      COMMON /GCOM6/ EENQ(100),IINN(100),KKRNK(100),NMAXQ(100),QQTIM(100
     1),SSOBV(25,5),SSTPV(25,6),VVNQ(100)
      COMMON /GCOM13/ DTFIX,DTNEX,EEPS,IIDIS,IIREJ,IIRGA,IIRGB,JJCRD,KKE
     1EP,KKILL,MMCNT,MMETH,NNINT,NNITR,NNREJ,NNSTP
      COMMON /OCOM1/ XXX(3)
      COMMON /UCOM1/ G,M,X,Y,Z,NBRSTO,SIMULT,D,DMOD,HEIGHT,HELP1,HELP2,H
     1ELP3,IMP,KM,KO,KR,PHIPUS,PI,R,STNRM,THETA,V,XSQR,ZSQR,NEWSTO,XSTON
     2E
      REAL M,IMP,KM,KO,KR
      INTEGER SIMULT,XSTONE
      V = D*FLOAT (NBRSTO - 1)/TNOW
      STNRM = FLOAT (NMAXQ(XSTONE) - 1)/50.0
      XXX(1) = D
      XXX(2) = V
      XXX(3) = STNRM
      CALL  CROSS
      WRITE (NPRNT,100)  D, V, STNRM
  100 FORMAT (4H D =,E12.4,4H V =,E12.4,8H STRNM =,E12.4)
      JJCRD = IICRD
      MMCNT = 1
      RETURN
      END
```

```
                                SIMULATION PROJECT NUMBER  60  BY       CELLIER

                                DATE  2/  6/ 1979       RUN NUMBER    1 OF    1
                                LLSUP=0000000000000000   GASP V.2  VERSION 25JAN79

 NCRDR=     5    NPRNT=     6    MMONT=     1    IICRS=     3    IISAV=     4    IITIM=     2
 NNOTP=     3    NNMXS=     0

 NMAMS=0              V                STNRM


 NNCLT=     0    NNSTA=     0    NNHIS=     0    NNPRM=     0    NNPLT=     0    NNSTR=     1    NNTRY=     1
 NNATR=     2    NNFIL=     1    NNSET=     4    NNEQD=     0    NNEQS=     1    NFLAG=     0    NNPOE=    -0

 KKRNK= (   1)

 IINN = (   1)

 IIEVT=     1    LLERR=     0    AAERR=      .1000E-03          RRERR=      .1000E-03
 DTMIN=      .1000E-05          DTMAX=      .2000E+00          DTSAV=      .2000E+00

 NSTOP=     0    JJCLR=     1    JJBEG=     1    IICRD=    13    TTBEG=     0.          TTFIN=    -0.
 JJFIL=     1
 IISEQ=    -0
*****WARNING - NEW VALUE COMPUTED FOR:  NNFIL=     2

*****WARNING - NEW VALUES COMPUTED FOR:  NNATR=    11    NNTRY=    34

*****WARNING - NEW VALUE COMPUTED FOR:  NNSET=   442
*****MAKE SURE YOU HAVE ENOUGH SPACE RESERVED FOR ARRAY QSET.
```

Fig. 9.13: GASP-V program for the DOMINO game.

Subroutine STONE may be precompiled and stored in compiled form for later reuse. In this case, however, we would most probably remove the "CALL FINIS" statement from its TERMINAL section and replace it by setting a parameter which is returned to the user to let him decide what he wants to do.

Fig. 9.14 shows a graphical representation of the chain velocity as well as of the maximum number of concurrently moving stones (divided by 50.0 to obtain compatible scaling factors for both curves) plotted versus the distance between stones.

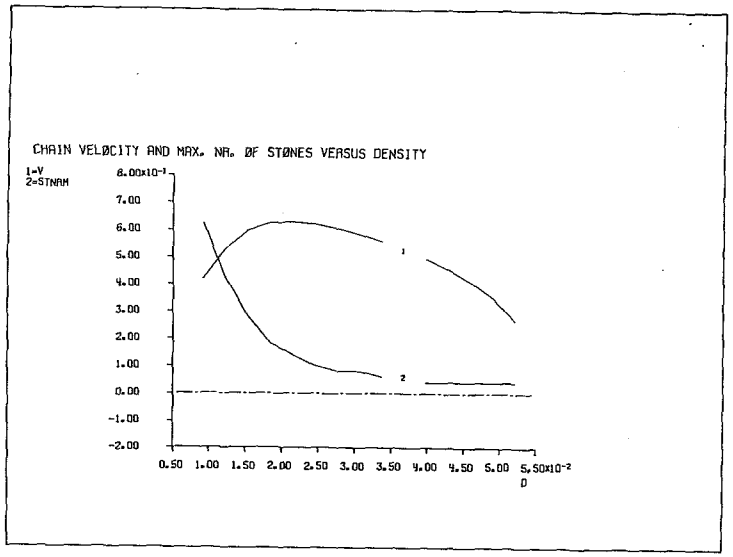

Fig. 9.14: Chain velocity versus distance between stones.

The maximum chain velocity (V) for this problem is 0.6277 m/s. It is reached at a distance (D) of 0.02147 m between stones (measured between centers of neighbouring

stones).

A real experiment would lead to larger values for V. The model, as it has been formulated, does not account for the additional weight originating from stones laying on each other's "back". In a real experiment, one can observe that the chain velocity accelerates during the early stages of the experiment. A more realistic model would, however, be by far more complicated, and, therefore, not very intelligible.

References:

[9.1] A.P.Bongulielmi: (1978) "Definition der allgemeinen Simulationssprache COSY". Semesterwork, Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich. To be obtained on microfiches from: The main library, ETH - Zentrum, CH-8092 Zurich, Switzerland. (Mikr. S637).

[9.2] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[9.3] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker); pp. 391 - 402.

[9.4] F.E.Cellier, Bongulielmi A.P.: (1979) "The COSY Simulation Language". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company.

[9.5] O.J.Dahl, Nygaard K.: (1966) "Simula; A Language for Programming and Description of Discrete Event Systems". Oslo, Norwegian Computing Center.

[9.6] D.A.Fahrland: (1970) "Combined Discrete-Event Continuous System Simulation Language". Simulation vol. 14 no. 2 : February 1970; pp. 61 - 72.

[9.7] D.G.Golden, Schoeffler J.D.: (1973) "GSL - A Combined Continuous and Discrete Simulation Language". Simulation vol. 20 no. 1 : January 1973; pp. 1 - 8.

[9.8] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice Hall.

[9.9] H.Lienhard: (1978) "PORTAL Language Definition". To be ordered from: Landis & Gyr AG, Zug, Switzerland. (Partly in German).

[9.10] H.Lienhard: (1978) "Die Echtzeitprogrammiersprache PORTAL, eine Uebersicht". Landis & Gyr Mitteilungen 25(1978); pp. 2 - 8. (Also available in English.)

[9.11] H.Lienhard, Meyer M., Steinle B., Wehrli P.: (1979) "Simulation and Process-Control with Parallel Processes as Implemented in PORTAL - Experience and Outlook". Proc. of the 9th IMACS Congress on Simulation of Systems, Sorrento, Italy. Published by North-Holland Publishing Company, (Editors: L.Dekker, G.Savastano, G.C.Vansteenkiste).

[9.12] A.A.B.Pritsker: (1974) "The GASP-IV Simulation Language". John Wiley.

[9.13] D.F.Rufer: (1978) "General Purpose Nonlinear Programming Package". Proc. of the 8th IFIP Conference on Optimization Techniques, Wuerzburg, FRG. Published by Springer Verlag, Lecture Notes in Control and Information Science, vol. 7, part 2; pp. 495 - 506.

[9.14] D.F.Rufer: (1978) "Users Guide for NLP - A Subroutine Package to Solve Nonlinear Optimization Problems". Report No: 78-07. Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[9.15] T.J.Schriber: (1974) "Simulation Using GPSS". John Wiley.

[9.16] (1972) "SIMPL/I Program Reference Manual". Form: SH19-5038-0, IBM Corp., New York, U.S.A..

[9.17] (1979) "Preliminary ADA Reference Manual". ACM SIGPLAN Notices, Vol. 14, No. 6, Part A, June 1979.

[9.18] (1979) "Rationale for the Design of the ADA Programming Language". ACM SIGPLAN Notices, Vol. 14, No. 6, Part B, June 1979.

X) INTERACTIVE SIMULATION AND REAL-TIME PROGRAMMING:

When reading the reports of the Technical Committee on Operating Systems (TC8) of the International PURDUE Workshop on Industrial Computer Systems, Purdue Europe [10.3], it is interesting to notice that the language elements which are considered to be useful in programming real-time operating systems are precisely the same as those which are used in discrete process-oriented simulation. Only the terminology varies. For example, in RTOS one does not SEIZE a FACILITY, but rather ALLOCATEs a RESOURCE, etc.. This resemblance is not surprising since these people require list processing (which is obviously application independent), and since their interest is focused on job-shop scheduling which is a closely related field to discrete simulation.

When reading about "High-Speed Block Diagram Languages for Microprocessors and Minicomputers in Instrumentation, Control, and Simulation" [10.1], one can find that also in another field of real-time programming there exists obviously some close relation to simulation. This time, it is the continuous system simulation which is related to real-time programming for applications of digital on-line control. The software which is presented in [10.1] is called MICRODARE-II [10.2], and is a co-product of a whole class of continuous simulation languages (the "DARE family").

Even these two real-time programming tasks are, however, not too different from each other. One can easily imagine situations where a computer has to take continuous measurements from a system which are to be filtered, and also execute distinct actions if, for example, some thresholds are surpassed. In such a situation, one would again require a combined continuous and discrete approach as has been presented in this work. There are only few additional elements required for such purposes:

a) means to communicate data with the real-world environment and,

b) real-time synchronization mechanisms.

In some ultra-fast, real-time applications, parallel computation may be necessary which would require additional synchronization and data protection mechanisms.

In our opinion, this relationship between simulation and real-time programming should be further investigated, and a co-operation between the different standardization committees should be encouraged.

References:

[10.1] G.A.Korn: (1978) "High-Speed Block-Diagram Languages for Microprocessors and Minicomputers in Instrumentation, Control, and Simulation". Private yet unpublished communication. To be ordered from: Department of Electrical Engineering, The University of Arizona at Tucson, Tucson AZ 85721, U.S.A..

[10.2] G.A.Korn: (1978) "MICRODARE-II User's Manual". To be ordered from: Department of Electrical Engineering, The University of Arizona at Tucson, Tucson AZ 85721, U.S.A..

[10.3] (1978) "Up to Date Report - Intermediate Version". International PURDUE Workshop on Industrial Computer Systems, PURDUE Europe, TC8. September 1978. To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

XI) DISCUSSION AND OUTLOOK:

In this thesis we have compiled the multi-faceted theory of simulation software. Starting from the discussion of numerical aspects (simulation run-time systems), we went on to information processing considerations (simulation languages, simulation compilers).

Although the term simulation system has been introduced as well (chapter VI) , we did not pay much attention to it. In future discussions of this topic, this term shall gain importance as we shall subsequently show.

Taken the definition of Korn [11.1] for the term "simulation":

"Simulation is experimentation with data"

it is obvious that simulation systems should provide a high degree of interactivity. This is not the case in current simulation systems. Indeed, one of the major advantages of analog computers over digital computers for the solution of simulation tasks (beside of their higher speed) is the extremely intimate interaction between program and programmer they provide for.

Why does current simulation software not provide such facilities? General purpose simulation programs are relatively complex programs, and they shall become even more complex in the future. They require a relatively large amount of core memory for convenient implementation. Facilities for intensive interactiveness, however, are not generally provided for large computing installations, but, on the contrary, for process computers only. Most of those systems are currently 16 bit machines which allow 32k of core memory to be directly addressed. This is insufficient for reasonably con-

venient simulation software to run on. However, one can find that the first 32 bit process computers became available recently. In a few years only, even relatively inexpensive 32 bit microprocessors shall be on the market. This new generation of hardware does no longer impose the previously mentioned restrictions on the programmer. By that time, complex and highly interactive simulation systems with comfortable graphical input/output facilities shall be offered.

As we have seen in this thesis, simulation is often just a subtask of what has to be performed. We have, for instance, seen that optimization is commonly used in connection with simulation. Moreover, we have assumed so far that a model of the system under investigation has already been found (by any mysterious process). However, the computer may aid in the determination of models as well. Parameter estimation is just one of several possible ways to do so. Often, models which have been derived from physical knowledge of the process under investigation cannot be validated. We then must apply model reduction techniques to these models to obtain simpler and validatable models. Very often, however, a physical knowledge of the process to be investigated is absent. In such a case, we must be able to construct models out of series of measured or wanted input/output relationships. In this way, the econometrical models are generated to mention just one example. Finally, one should be able to transform model representations (e.g. discrete-time models into continuous-time models, and vice-versa). All these tasks can be called operations on models.

Moreover, also operations on data (e.g. comparison of real data with simulated data, Fourier analysis, statistical analysis, etc.) are very important.

For all of these tasks, our future simulation system must provide for a very powerful data base management system in which both structures (models) and data (real and simulated)

can be stored, and which provides for a highly standardized data interface such that several different programs can access this data base to perform operations on the stored data. These programs include nonlinear programming, model manipulation programs, and data manipulation programs beside of the "pure" simulation software.

The background for these programs should be a unified theory of modeling as it has been described by Zeigler [11.3]. A rationale for and some required attributes of such a simulation system have been recently outlined in an excelent paper by Oren and Zeigler [11.2].

References:

[11.1] G.A.Korn, Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice-Hall.

[11.2] T.I.Oren, Zeigler B.P.: (1979) "Concepts for Advanced Simulation Methodology". Simulation, Vol. 32, No. 3, March 1979; pp. 69 - 82.

[11.3] B.P.Zeigler: (1976) "Theory of Modelling and Simulation". John Wiley.

XII) REMARKS CONCERNING NOTATIONS:

The field of combined system simulation, as it has been discussed in this work, is an interdisciplinary subject. Continuous simulation has always been primarily used by people with a background in Automatic Control. These people have a specific terminology which they commonly use, and which has been incorporated in the continuous simulation languages. Discrete simulation has primarily been used by people with a background in Operations Research. They also have a distinct terminology which has been used in the design of discrete simulation languages. Unfortunately, these two fields have many common elements, but use often a quite different terminology.

Common elements use different names. In Automatic Control, for example, one discusses the "Maximum Principle of Pontrjagin", a theory which is useful in Operations Research as well, but which is there referred to as the "Minimum Principle of Pontrjagin". Even the term "discrete system" will easily lead to misunderstandings since researchers in Automatic Control will implicitly assume that a "discrete time system" is meant, that is a system described by sets of difference equations. On the contrary, Operations Research investigators will implicitly assume that one talks of a "discrete event system" since this is much more common to them.

Different elements use identical names. A "process", as it is used in Automatic Control, is certainly different from the use of this term in Operations Research.

For this reason, we were often forced to either use terms in the sense of continuous simulation or in the sense of discrete simulation, or even to invent new terms to avoid misunderstandings. We tried, however, to keep the conflicting

situations at a minimum, and we sincerely hope that we have not added to the Babylonian language confusion by these means.

Another source of confusion is the difference in notation as compared to the common use in Information Science. These investigators use, for instance, the term "MODULE" to mean a SIMULA-67 "CLASS" which is certainly incompatible with our usage of this term. However, the same term "MODULE" is used as well in Formal Algebra and in Reliance Analysis, and has a quite different meaning in all of these fields. Our terminology bases on the Engineering usage of the term. In any event, it is difficult to find a large number of meaningful new terms which should be as close as possible to conversational English.

This situation was not bothersome as long as the different groups were disjoint from each other. However, since there is a strong overlap between these fields, as it has been shown in this thesis, a common terminology whould definitely help to improve the information transfer, and we would, therefore, emphasize to let a standardization committee propose a new terminology which is to include all important terms out of all three fields (Automatic Control, Operations Research, and Information Science).

AUTHOR INDEX

EXAMPLE INDEX

SOFTWARE INDEX

Alphabetical Index

structured	->simulation,system,software,run-time system
	->programming
structuring	->programming,data
	->model,operation
subprogram	->simulation,system,software,language,robustness,modeling
	->programming,structure
	->simulation,system,software,language,structure,segment, declaration
subroutine	->programming,structure,subprogram,classification
	->simulation,system,software,language,structure,segment, declaration,subprogram,classification
synchronization	->simulation,system,software,run-time system,classification, continuous,numerical integration,method
syntax	->simulation,system,software,language
system	->
	->simulation
	->simulation
system engineer	->simulation,system,software,language,extendability
system theory based	->simulation,system,software,language
systems dynamics	->model,application, economy, management, and social sciences
technique	->modeling
terminal	->simulation,system,software,language,structure,segment
terminology	->simulation
text editing	->simulation,system,software,robustness,updatability
theory	->modeling
time-persistent variable	->experimentation,operation on data,collection, statistical quantity
time	->simulation,system,software,language,structure,segment, discrete,event
topological	->modeling,approach,network-oriented,description
topological description	->simulation,system,software,language,structure, network description
transformation	->model,operation
	->simulation,system,software,run-time system,classification, combined,discontinuity function
transparency	->simulation,system,software,language
type	->modeling,approach,network-oriented
unambiguous	->simulation,system,software,language,grammar
updatability	->simulation,system,software,robustness
user	->simulation,system,software,language,extendability
validation	->model
variable	->programming
	->simulation,system,software,language,structure,segment, continuous,memory
	->simulation,system,software,language,structure,segment, declaration
	->simulation,system,software,run-time system,classification, continuous,numerical integration,step-size
variable structure	->model,classification,combined
	->simulation,system,combined
verification	->simulation
well-defined	->system
white-box	->system
3-dim. graphical representation	->experimentation,operation on data,retrieval, state trajectory
	->simulation,system,software,postprocessor

CURRICULUM VITAE

I am born in Zurich, Switzerland on July 30, 1948 as a son of Jean and Anita Cellier-Borchardt. I attended the primary school in Duebendorf/ZH for five years. Thereafter, I entered the preparatory class of the "Freies Gymnasium Zurich" where I passed my final examinations in autumn 1967 with a specialization in the humanistic field (Maturitaets-pruefung, Typus A). Immediately afterwards I started my studies at the Department of Electrical Engineering of the Swiss Federal Institute of Technology Zurich. I was awarded my Diploma degree in Electrical Engineering in spring 1972. Following graduation I spent another two semesters in 1973 attending the postgraduate courses in Automatic Control. Since August 1972 I am employed at the Institute for Automatic Control and Industrial Electronics. During the years 1972 to 1976 I was mainly engaged in an industrial project on simulation, design, and adaptive control of machine tools sponsored by AGIE AG (Losone/TI). Since then I worked as an Assistant. Beside of this work I was granted time to write my Ph.D. thesis. Finally, in spring 1978 I was appointed by the Department of Electrical Engineering as a Lecturer of Simulation Techniques.

Zurich, October 1979

Francois E. Cellier