HOW TO ENHANCE THE ROBUSTNESS OF SIMULATION SOFTWARE

Francois E. Cellier
Institute for Automatic Control
The Swiss Federal Institute of Technology Zurich
ETH - Zentrum
CH-8092 Zurich
Switzerland

ABSTRACT

This chapter describes different means to improve the robustness of simulation software (languages, compilers, and run-time systems) with respect to products which are currently available on the software "market".

It is shown how these improvements can help to ameliorate the robustness of models and of their coded counterparts: the simulation programs. Model robustness forms a part of the total validity picture, while simulation program robustness partly covers the correctness verification assurance.

This chapter addresses itself primarily to the simulation software designer. It is hoped that these considerations may help future software developers in producing more reliable simulation software.

NATO ASI Series, Vol. F10
Simulation and Model-Based Methodologies:
An Integrative View. Edited by T. I. Ören et al.

1. INTRODUCTION

Digital simulation reaches back to the late fifties of this century. At that time, and in the early sixties, digital (continuous) simulation systems were meant to replace the former analog equipment. Simulation programs were coded by connecting blocks representing the components of analog computers (like integrators, summers, multipliers, etc.) as one used to "wire up" analog patchboards. The coding format of these software systems was extremely rigid, and their capabilities were very limited. About 1965, people started to realize that the digital computer can do much better than to duplicate analog machinery as both types of hardware have their specific advantages and drawbacks. As a result, the well known CSSL's (Strauss 1967) were developed (like CSMP-III (IBM 1972), CSSL-IV (Nilsen 1980), ACSL (Mitchell, Gauthier 1982), DARE-P (Korn, Wait 1978)). For each of those languages, a user manual was written showing how beautifully the Van-der-Pol equation and the pilot ejection study could be formulated by means of the particular software tool(!).

Even in the early seventies, most models under investigation were low order models with a rather limited degree of complexity. Therefore, it was entirely legitimate to conclude that a piece of software being able to handle the Van-der-Pol equation in an elegant manner would be equally acceptable for whatever other application one might dream of.

Meanwhile, this situation has changed drastically. The art of modelling is applied to an ever increasing "clientele" of problems from all kind of different application fields. At the same time, also the number of (differential) equations of the average simulation study is constantly growing. New terms have been created like: "ill-defined system modelling" and "large-scale system modelling" to denote those new classes of problems which were previously not solvable at all, partly due to computer hardware limitations (restrictions of core memory and execution speed), and partly for reasons of lack of knowledge and understanding of the involved processes.

Unfortunately, it cannot be concluded that a piece of software being appropriate for the formulation and simulation of the Van-der-Pol equation is equally fit for the treatment of those new types of problems. With the average model growing in complexity, the average simulation program grows in length, and we must request that the simulation program is able to detect as many modelling and programming errors -- be they of a syntactical or of a logic type -- as possible, and at an as early stage of the investigation as possible.

This aspect of simulation software is called "robustness". Its importance grows at least quadratically with the length of the average simulation program. IBM and Bell Laboratories have carried out studies recently in which they compared the cost for debugging programming errors at different stages of an investigation. They found that, given a cost of one monetary unit for correcting a program error which is detected during programming, the same error would cost 20 monetary units when removed during the software test phase, and up to 120 monetary units when removed while the program is already in production (e.g. Aaronson 1983).

We feel that the currently available CSSL's are far from meeting these robustness requirements, and it is our aim to show how these deficiencies can be overcome in the future.

Possibilities which exist to improve simulation system robustness are manyfold. The robustness of simulation languages and their compilers can be improved with respect to modelling, programming, maintenance, and implementability. Possible measures have mostly to do with the improvement of simulation language definitions and with the introduction of redundancy -- that is with aspects of information processing. Much research has been devoted to these aspects in the development of general purpose programming languages, a knowledge which can be profitably applied to simulation software as well. On the other hand, one can also improve the robustness of the underlaying simulation run-time system mainly through measures involving aspects of numerical mathematics. Again other improvements concern the management of data produced in the simulation study. As we shall show, also this (so far mostly neglected) aspect may increase the model robustness markedly.

The aim of this chapter is to outline and classify in a systematic manner different possible means of improving simulation software robustness.

2. SIMULATION LANGUAGES

2.1 Modelling

Simulation languages should be constructed in such a way that they support their users in the formulation of "valid" models, valid with respect to the tasks they are designed for. We feel that there exist several means by which simulation languages may assist users in their modelling activities.

1) The simulation language can provide for appropriate structures which allow to represent modules of the real system under investigation by corresponding modules of the simulation language.

Precut structures can assist the user in structuring his problem. This shall be illustrated at an example. Several groups of our students were asked to model and simulate a solar energy heating system. For this complex modelling task they had 16 weeks available with an average working load of about 15 hours per week (term project). This is a typical combined continuous/discrete problem with time-events (sunrise, sunset, toggling between night and day service) and state-events (the pump for the circulation of the liquid is either "on" or "off" depending on the temperature at the collector, additional oil heating can be in an "on" or "off" status depending on the temperature in the building). One student used ordinary FORTRAN programming for the task, and called subroutines for numerical integration and report generation from a library of FORTRAN routines. After sixteen weeks, this student ended up with a huge and unstructured program for which he was unable to draw a proper flow chart. He had entirely lost the overview of his program, and it never worked. The program was very badly structured from the beginning. Other groups used the GASP software (Pritsker 1974, Cellier, Blitz 1976) for this modelling task, and found it much easier to construct running (although not necessarily valid(!)) simulation programs. GASP-V is a collection of FORTRAN subroutines, and as such there should principally be no difference between using GASP and using FORTRAN directly. The difference arises from the fact that GASP provides appropriate structures to subdivide models into continuous and discrete portions, and by these means guides the user smoothly through the modelling task.

Hierarchical structures are another important concept in model definition. Such structures are, of course, not implementable at the level of FORTRAN programming (and are thus not available in GASP). Most CSSL's offer a MACRO facility for that purpose, which allows one to specify new operators as a combination of already existing ones. However, as it has been shown first by Elmqvist (1979), the MACRO facility is not really modular either, as the required form of a submodel depends on the environment into which it is embedded. A statement of the form: "U=R*I" may easily be requested to take the form: "I=U/R" in another context. Therefore, sorting of equations alone (as it is offered in most available CSSL's) is insufficient for a true modularity of submodels. More general is the MODULE concept as it is for example porvided in DYMOLA (Elmquist 1981), MODEL (Runge 1979), or COSY (Cellier, Bongulielmi 1979, Cellier, Rimvall, Bongulielmi 1981).

2) The user may be asked to provide dimensions for all variables. The compiler can

then perform an automated dimensional analysis for all equations. The user may also supply ranges for all variables. This enables the run-time system to check whether trajectories leave their allowed range during simulation. This feature is specially valuable in case of large-scale systems where the user will not be able to display all variables on output. He may, for instance, believe in the outcome of a world model study because those variables he looks at seem to take reasonable values whereas somewhere else in the model a population decreases below zero. Such additions have been proposed by Elzas (1979).

3) The future simulation language should contain features which would allow to perform some model manipulations in an automated manner, like generation of a linearized model, generation of a sensitivity model, generation of a meta-model. Such features have been proposed and explained by Cellier and Fischlin (1980). A definition for the term meta-model can be found from Kleijnen (1982).

2.2 Programming

As explained in the introduction, errors should be detected as early and completely as possible. For this purpose, the simulation language definition must contain sufficient 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 obviously not yet reached most of the simulation software designers, as in todays simulation languages such features are hardly ever offered. This is probably due to the fact that most simulation software designers stick too closely to the CSSL-definition which was published before one paid too much attention to questions of software robustness. Clearly, there is no need to have such a feature for the formulation of the Van-der-Pol equation which takes two or three lines only. On the contrary, for such scholastic examples, declaration of variables even looks awkward because short user programs are a very strong argument, too. Nevertheless, as the art of modelling has matured, also the simulation software must follow.

Into the same category falls the use of LL(1) grammars for language definition. LL(1) grammars allow to parse programs from left to right by looking just one symbol ahead. No backtracking is required. Therefore, a symbol which is not foreseen as one

of the legal alternatives at any place in the application program must be detected at once, and can accordingly be reported to the simulation user. The use of LL(1) grammars for simulation language definitions has been thoroughly discussed by Bongulielmi and Cellier (1979).

Another possible help is to check on model consistency and completeness. Some possible means have been proposed by Richards (1978). Specially valuable in this context seems to be the proposed sequence test which helps to make sure that no variable is used before it is defined. It is furthermore useful to preset the load window of the program to some illegal value (like negative infinite on a CDC CYBER installation) if such a feature is available.

3. SIMULATION COMPILERS

Also simulation compilers can be robust in several different ways, namely with respect to programming, implementation, and maintenance.

3.1 Programming

This aspect is very closely related to the previous one. The simulation compiler should perform extensive error testing while parsing the application program.

This must 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 and execution time. GASP-V (Cellier, Blitz 1976) for example, uses 110.000_8 core memory locations on a CDC CYBER. 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 (like the previously proposed dimensional analysis) are feasible.

3.2 Implementation

This aspect of robustness involves insensitivity to alterations in the operating system, the underlying computer hardware, and peripheral equipment. It 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 on many different types of computers. This, again, has its implications with respect to 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 may be somewhat restrictive in the data structuring capabilities it offers, and may also limit the use of recursions (although a more elaborate preprocessor may obviously map most of the data structures available in modern structured languages like PASCAL into FORTRAN structures, and/or may simulate recursions by use of an explicit stack, this at the cost of a somewhat unreadable target code). Until recently, FORTRAN was the most appropriate language for coding the simulation run-time system, whereas good candidates for implementation of the preprocessor were either PASCAL or SIMULA-67. In the future, ADA may be a promising candidate for both tasks (for the preprocessor due to its dynamic data structures and due to its exception handling capabilities, for the run-time system due to its build-in tasking features and due to the feasibility of safe separate compilation of subprograms). It shall, however, still require some time, because full ADA compilers are not yet available on many machines, and because ADA-coded versions of reliable numerical libraries (e.g. for linear algebra, numerical integration, statistical analysis, etc.) still need to be developed.

It is sometimes argued that the preprocessor concept, which we strongly advertize due to its portability, suffers from a major drawback, as input errors are frequently not detected by the preprocessor itself but only at a later stage, that is by either the FORTRAN (or ADA) compiler or the run-time system. Error messages then would not report to the user program, but to a code which the user has hardly any

knowledge of. This problem is, however, a problem of implementation rather than a problem of principle. By use of an LL(1) grammar, it is possible to ensure that no syntactic errors will be proliferated to the next compiler stage, that is: errors detected by the FORTRAN (or ADA) compiler can totally be avoided at the price of a somewhat slower precompilation. Run-time errors (that is: semantic errors) are obviously not totally avoidable, but their reporting can be improved by use of a context file produced by the preprocessor which is passed on to the run-time system. There is no reason why the well-known backtracing mechanisms should not work through two stages of compilation as well as through one. Current simulation systems do not offer such a facility, but this is just a question of the sophistication of the software design and implementation.

3.3 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.

Also for this purpose, it is good use to define the simulation language by an LL(1) grammar. Although this imposes some restrictions on the freedom of how the language is defined, it poses more restrictions as to the form of the language (grammar) than as to the expressiveness of the language. In general, all required features of simulation languages can be easily formulated by LL(1) grammars, possibly with the exception of general purpose macros. In that context, we must either restrict ourselves in the macro facilities the language is to offer, or we must leave the pure LL(1) environment, which is not critical either as general purpose macros must anyway be processed in a separate compilation step. It might even make sense to code the macro handler as a totally independent pre-preprocessor.

The reason for using LL(1) grammars is explained below: It is unavoidable that a complex program like a compiler contains some "bugs" which are not detected until somebody stumbles upon them by chance. This problem is even more severe for simulation software as compared to general purpose programming languages since there exists a smaller number of users (read: guinea-pigs) for them. When an error is detected, it is most likely that the programmer of the compiler 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 often creates unwanted side-effects which are difficult to explain and to correct. Another frequent reason for badly maintainable compilers lies in a careless and unsystematic use of structured data references (like unsystematic use of pointer variables in a PASCAL program). Since LL(1) grammars allow compilers to be written in an almost top-down structure by use of highly systematic data referencing mechanisms, the robustness of such compilers with respect to their maintainability is markedly better than in the case of other types of grammars.

4. SIMULATION RUN TIME SYSTEMS

Also simulation run-time systems can be robust in two entirely different senses:

4.1 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 as much as possible on those factors which only have to do with the statement of his problem, and should not be forced to bother about details 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.

Let us give an example of weakness in procedural robustness. In the early days of digital simulation, the user was asked to provide a step size for the numerical integration algorithm to be used. The user had no information, whatsoever, on how large that could be. He, therefore, was forced to try different step sizes to determine an optimal choice. Meanwhile, algorithms have been developed which determine automatically and "on-line" which step size is to be used. The user must provide the algorithm only with information about his accuracy requirements. This type of information is related to the physics of the problem rather than to the numerical

algorithm, and can usually be provided easily.

Currently, the user is still requested to select the most appropriate integration algorithm out of the available library for the simulation of his task. Again, this is a request which cannot easily be answered by the average user. In future, we can expect that an automated way of determining the most appropriate algorithm for any particular application problem would form an intrinsic part of any simulation software. Some steps into this direction have been carried out, and are described by Cellier and Moebius (1978). Some further results in this field were presented very recently by L.R. Petzold (Hindmarsh 1982). In fact, the needs for such an algorithm were detected quite early. Elzas implemented a first version of such an algorithm as early as 1962 in the DISAR system (for the TR4 machine) (Bos 1962).

The situation gets worse when partial differential equations (PDE's) are to be solved. By use of the method-of-lines approach (which is the most general and most common approach in simulation today), PDE's are decomposed into sets of ordinary differential equations (ODE's) which are then integrated over time by use of any ODE solution technique. There remain three parameters to be user-tuned:

a) the integration method (for integration over time),

b) the method of discretization (of the spatial dimensions), and

c) the grid width of the discretization scheme.

Unfortunately, an optimal choice is here even more important than in the ODE case, as differences in computing time may result which are many orders of magnitude apart from each other (cf. e.g. Rice 1976). There exists an almost infinite number of combinations, and, therefore, a strategy of "blind search" to detect an optimal combination is hopeless. In fact, the three parameters cannot be tuned independently of each other, as the stability properties are determined by combinations of them. First steps towards an automated grid width control have been reported recently by Schiesser (1982). Certainly, more research is still required.

4.2 Algorithms

The run-time software itself must be able to check whether the provided time histories 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 "inside information" into the tool he is using for his 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" to ring as soon as it is unable to properly proceed. Under no circumstances are incorrect results allowed to be displayed to the user.

As an example, numerical instabilities (or possible sources of instabilities) should be detected and reported to the user. How this can be achieved, is also described by Cellier and Moebius (1979).

Again, the situation is particularly awkward in the PDE case also with respect to algorithmic robustness. A bad choice of the applied numerical algorithms does not necessarily lead to an error indication. On the contrary, there will often be produced results 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 scheme, differentiation scheme, and space discretization will lead to a finite difference scheme which is consistent, convergent, and stable. Stability will usually be taken care of 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" results 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).

Automated grid width control, as proposed by Schiesser (1982), may be one answer to this problem, but more research is still required.

What is not taken away yet is the effect of different word-length conventions on different machines. They will lead to a disparety in results, especially in stiff system simulation. It is a good idea to evaluate the machine resolution "on line" by use of a terribly simple algorithm proposed by Rutishauser (unpublished lecture

notes):

```
      EPS = 1.
1     IF ((1.0+EPS).EQ.1.0) GO TO 2
      EPS = EPS*0.5
      GO TO 1
2     EPS = 2.0*EPS
```

to determine the smallest number which is distinguishable from 1.0. Obviously, the result of this little algorithm depends heavily on the available number of bits. Once EPS is computed, all algorithms (and in particular all termination criteria of iteration loops) should be programmed relative to this number EPS to minimize machine dependencies. It is even an excellent idea to provide the user with the facility to enlarge this quantity EPS to simulate a lower precision machine on a higher precision hardware. This requires that, following each floating-point operation, a number of bits (or digits) is chopped by the algorithm upon demand. Such a feature is, for example, provided in MATLAB (Moler 1980).

5. SIMULATION DATA

Many examples can be cited which show that the data handling mechanisms in currently available simulation software systems are insufficient (Cellier 1983). To mention just one of them: It is certainly no unreasonable wish to be able to display two simulation trajectories produced by two different models of the same system on one sheet of paper. None of the currently available simulation software systems would allow to achieve this in a natural and comfortable way. The only meaningful solution to this problem would be to provide for an interface to a relational data base, that is: simulation data can be stored in the data base during simulation, some other data (e.g. parameters, driving functions) may be retrieved from there by the simulation program. Several other programs (e.g. graphical postprocessor, statistical analysis program, etc.) have also interfaces to the same data base, and can, therefore, access the same data. It is important in this respect that models and experiment descriptions are properly separated from each other, as this was demanded by Zeigler (1976). By means of this concept, the above mentioned situation can be handled in a very flexible and natural way by:

a) coding the two models as two separate programs which both store their resulting trajectories in the data base, and

b) by using the graphical postprocessor to retrieve the previously stored data and

to produce meaningful graphs.

In currently available simulation systems, one may well be able to specify both models in the same program and switch between them by means of a logical variable. However, this is by no means an elegant solution, and it would not help to solve some of the other requirements (like representing simulated and measured data on one single sheet of paper). Several of the simulation systems which are currently under development shall provide for such a mechanism, e.g. COSMOS (Kettenis 1983), COSY (Cellier, Bongulielmi 1979, Cellier, Rimvall, Bongulielmi 1981), and SYSMOD (Baker, Smart 1983).

This data base concept can enhance the robustness of simulation software in several ways:

5.1 Modularity

The required language features are spread over several independent programs (simulation program, graphical postprocessor, etc.) which communicate through the data base only. This reduces the number of required keywords in each of these programs (e.g. the only output commands still required in the simulation language are commands to store the data in the data base, some tracing capabilities (for program debugging), some error reporting routines which are entirely unvisible to the user, and possibly a run-time display for interactive and real-time simulation. Many of the currently used language constructs (such as "PAGE MERGE" in CSMP-III) simply disappear altogether. In this way, the required compilers (or interpreters) can be kept smaller and thus better maintainable.

5.2 Documentation

The user manuals can be split into several independent volumes describing the use of the separate programs. Not every user is requested to read all of them. E.g. the manager shall most probably learn to use the graphical postprocessor and some other data retrieval and analysis tools, but certainly not the simulation language (at least not in detail, just enough to familiarize himself with the model structure).

5.3 Flexibility

One and the same graphical postprocessor can be used to analyse whatever data are stored in the data base, independently from how these data got there. In this way, the separate software system modules become more versatile, and are thus more frequently utilized by a larger diversity of users. Therefore, the chance of stumbling upon a "bug" which still remained in the software module despite careful software design and implementation decreases quicker with time.

5.4 Applicability

As the previous example shows, the flexibility of the software increases by use of the data base approach, helping to enhance the robustness of the simulation software with respect to the diversity of problems which can be tackled.

6. SIMULATION SYSTEMS

The term simulation system denotes the union of simulation language, simulation compiler, simulation run-time system, simulation data, and (last but not least) the documentation volume. Experience has shown that program code is updated much faster and easier than documentation material. For this reason, it makes sense to discuss the robustness of a simulation system with respect to its:

6.1 Updatability

To make the documentation as easily updatable as possible, it is very convenient if also the documentation is developed by use of the computer. The text itself should (as this presentation) be composed by use of a powerful text editing system.

The syntactic rules of the language should be documented by means of syntax diagrams. This has been done already for several simulation languages (e.g. SCALE-F (Heppner 1977), COSY (Cellier, Bongulielmi 1979, Cellier, Rimvall, Bongulielmi 1981), CSMP-III (Degen, Grunta 1980), COSMOS (Kettenis 1983), and SYSMOD

(Baker, Smart 1983). Bongulielmi and Cellier (1979) 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) parsability. Another program (Bucher 1977) 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 the 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. The implementers of SYSMOD (Baker, Smart 1983) went even one step further. The EBNF productions form part of the user manual. A special text editor is able to extract these productions (with reference to the respective paragraph as a comment) from the manual, and automatically generates the input to the parser program. In this way, each modification of the language grammar starts by a modification of the user manual, which is an extremely useful feature.

7. ROBUSTNESS OF MODELS

The final aim of all the robustness considerations we have discussed is to ensure that a particular application program (that is: coded version of a conceptual model of a system under investigation) performs its task. We must here distinguish between several types of reasons why this may not be so.

1) Both the model and the simulation program are correct, but the generated time histories are in error. This should never happen in principle. The simulation run-time system must take care to avoid this. Possible means to avoid such situations have been discussed in Section 4.2.

2) The model is correct, but the simulation program has syntactical errors. Unfortunately, it happens often with currently available simulation compilers that they translate correct programs correctly whereas they produce any unpredictable code for incorrect programs without warning the user that something may be wrong with his program(!). By restricting ourselves to LL(1) grammars, we can make sure that this shall never happen. This has been discussed in Section 2.2.

3) The model is correct, but the simulation program has semantic errors. Unfortunately, there is no way to determine all such errors at once. Some possible means to reduce the risk for this situation to happen have been discussed. They have mostly to do with the introduction of redundancy which allows to perform different types of consistency checks. However, some errors may only be detectable during program execution. For such cases, the previously described context files (backtracing technique) may help. Some thoughts to simulation program verification have been discussed by Sargent (1982), some more can be found in (Elzas 1979, Richards 1978).

4) The model itself is incorrect in that restrictions in the kind of experiments which can be performed on it are violated by the actual experiment performed. (A model as such is never valid or invalid. It is the pair (model, experiment) which has validity as its attribute. Any model is able to meet the "null" experiment.) It is extremely difficult to ensure model validity in a general way. Several possible means to improve the probability of a model to meet its requirements have been discussed. A more thorough discussion of model validation was given recently by Sargent (1982, 1983). The principles are also covered by Elzas (1983).

8. CONCLUSIONS

We have shown that the currently available simulation software does no longer meet the ever increasing needs of the average simulation users. Especially for large-scale system modelling and ill-defined system modelling, the classical CSSL-type languages can no longer suffice. We tried to list in a systematic manner different suggestions for improvement of future simulation systems which would meet the growing needs of modellers better.

REFERENCES

Aaronson, S. (1983), "Software and Society: An Interview with Eric Somner", Bell Laboratories Record, February, pp. 10-16.

Baker, N.J.C. and P.J. Smart (1983), The SYSMOD Language and Run Time Facilities Definition, Technical Note 6.82, Royal Aircraft Estalishment, Farnborough, Hampshire, United Kingdom, 196 p.

Bongulielmi, A.P. and F.E. Cellier (1979), "On the Usefulness of Deterministic Grammars for Simulation Languages", in T.I. Oren, (ed.), Sorrento Workshop on International Standardization of Simulation Languages (SWISSL), Sorrento, Italy, to appear in Simuletter.

Bos, G. and M.S. Elzas, (1962), Digitale Simulatie van een analoge rekenmaschine, Afdeling der Technische Natuurkunde, Technische Hogeschool Delft, The Netherlands, 50 p.

Bucher, K.J. (1977), Automatisches Zeichnen von Syntaxdiagrammen, welche in spezieller Backus Naur Form gegeben sind, Users Manual, Institute for Informatics, The Swiss Federal Institute of Technology, ETH - Zentrum, CH-8092 Zurich, Switzerland, 17 p.

Cellier, F.E. (1983), "Simulation Software: Today and Tomorrow", in J. Burger and Y. Jarny, (eds.), IMACS Symposium on Simulation in Engineering Sciences, Nantes, France, pp. 426-442.

Cellier, F.E. and A.E. Blitz (1976), "GASP-V: A Universal Simulation Package", in L. Dekker, (ed.), Simulation of Systems, North-Holland, Amsterdam, pp. 391-402.

Cellier, F.E. and A.P. Bongulielmi (1979), "The COSY Simulation Language", in L. Dekker, G. Savastano and G.C. Vansteenkiste, (eds.), Simulation of Systems, North-Holland, Amsterdam, pp. 271-281.

Cellier, F.E. and A. Fischlin (1980), "Computer-Assisted Modelling of Ill-Defined Systems", in R. Trappl, G.J. Klir and F.R. Pichler, (eds.), Progress in Cybernetics and Systems Research, Vol. VIII, General Systems Methodology, Mathematical Systems Theory, Fuzzy Sets, Hemisphere Publishing Corp., Washington, pp. 417-429.

Cellier, F.E. and P.J. Moebius (1979), "Towards Robust General Purpose Simulation Software", in R.D. Skeel, (ed.), Numerical Ordinary Differential Equations, Dept. of Computer Science, University of Illinois at Urbana-Champaign, pp. 18.1-18.5.

Cellier, F.E., M. Rimvall and A.P. Bongulielmi (1981), "Discrete Processes in COSY", in F. Maceri, (ed.), European Workshop on Simulation Methodology, Cosenza, Italy, also in R.E. Crosbie and F.E. Cellier, (eds.), TC3 IMACS: Simulation Software, 11, Appendix 8, 31 p.

Elmqvist, H., (1979), "Manipulation of Continuous Models Based on Equations to Assignment Statements", in L. Dekker, G. Savastano and G.C. Vansteenkiste, (eds.), Simulation of Systems, North-Holland, Amsterdam, pp. 15-21.

Elzas, M.S. (1979), "What is Needed for Robust Simulation", in B.P. Zeigler, M.S. Elzas, G.J. Klir and T.I. Oren, (eds.), Methodology in Systems Modelling and Simulation, North-Holland, Amsterdam, pp. 57-91.

Elzas, M.S. (1983), Chapter 2 of this book.

Heppner, D. (1977), Beschreibung der Simulationssprache SCALE F, Handbuch F7, Computing Center, Technical University of Braunschweig, FRG, 150 p.

Hindmarsh, A.C. (1982), "Stiff Systems Problems and Solutions at Lawrence Livermore National Laboratory", in R.C. Aiken, (ed.), International Conference on Stiff Computation, to be published.

IBM (1972), Continuous System Modelling Program III (CSMP III), Program Reference Manual, Program Number: 5734-XS9, Form: SH19-7001-2, IBM Canada, Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, 186 p.

Kettenis, D.L. (1983), "The COSMOS Modelling and Simulation Language", in W. Ameling, (ed.), Proceedings of the First European Simulation Congress ESC'83, Springer Verlag, Informatik Fachberichte.

Kleijnen, J.P.C. (1980), "Experimentation with Models: Statistical Design and Analysis Techniques", in F.E. Cellier, (ed.), Progress in Modelling and Simulation, Academic Press, London, pp. 173-185.

Korn, G.A. and J.V. Wait (1978), Digital Continuous System Simulation, Prentice Hall, 212 p.

Mitchell, E.E.L. and J.S. Gauthier (1982), ACSL: Advanced Continuous Simulation Language, User/Guide Reference Manual, Mitchell and Gauthier, Assoc., 1337 Old Marlboro Road, P.O.Box 685, Concord, Mass., 272 p.

Moler, C. (1980), MATLAB, User Guide, Dept. of Computer Science, University of New Mexico, Albuquerque, 60 p.

Nilsen, R.N. (1980), The CSSL IV Simulation Language, User Manual, Simulation Services, 20926 Germain Street, Chatsworth, Calif.

Pritsker, A.A.B. (1974), The GASP IV Simulation Language, John Wiley, New York, 451 p.

Rice J.R. (1976), "Algorithmic Progress in Solving Partial Differential Equations", SIGNUM Journal, 11, 4, pp. 6-10.

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

Runge, T.F. (1977), A Universal Language for Continuous Network Simulation, Form: UIUCDCS-R-77-866, Ph.D. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 153 p.

Sargent, R.G. (1982), "Verification and Validation of Simulation Models", in F.E. Cellier, (ed.), Progress in Modelling and Simulation, Academic Press, London, pp. 159-169.

Sargent, R.G. (1983), Chapter 19 of this book.

Schiesser, W.E. (1982), Some Characteristics of ODE Problems Generated by the Numerical Method of Lines, in R.C. Aiken, (ed.), International Conference on Stiff Computation, to be published.

Strauss, J.C. (1967), "The SCi Continuous System Simulation Language (CSSL)", Simulation, 9, 6, pp. 281-303.

Zeigler, B.P. (1976), "Structuring Principles for Multifaceted System Modelling", in B.P. Zeigler, M.S. Elzas, G.J. Klir and T.I. Oren, (eds.), Methodology in Systems Modelling and Simulation, North-Holland Publishing Company, 1979, pp. 93 - 135.