Syst. Anal. Model. Simul. 1 (1984) 1, 55–61

How to Enhance the Robustness of Simulation Software

F. E. Cellier

Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich
ETH-Zentrum, Zürich, Switzerland

This paper describes different means to improve robustness of simulation software (languages, compilers, and run-time systems) as compared to products which are currently available on the software "market". It is shown how these improvements can help to ameliorate the robustness (validity) of models and of their coded counterparts: the simulation programs.

I. 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 continuous simulation system languages (CSSL) were developed (like CSMP-III, CSSL-IV, ACSL, DARE-P). For each of those languages, a users' manual was written showing how beautifully the Van-der-Pol equation and the pilot ejection study can 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 modeling 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 modeling" and "large-scale system modeling" 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 modeling 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 has carried out a study a few years ago in which they compared the cost for debugging programming errors at different stages of an investigation. They found that, given a cost of one money unit for correcting a program error which is detected during programming, the same error would cost 20 money units when removed during the software test phase, and up to 120 money units when removed while the program is already in production.

We feel that the currently available CSSLs are far from meeting the 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 modeling, 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 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.

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

II. Simulation languages

Modeling: Simulation languages should be constructed such 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 modeling 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 modeling 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, 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). 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 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. 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 raises 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 modeling task. Another very useful facility in this respect is the capability of defining models in a hierarchical manner. As it has been shown first in [5], the MACRO facility, as it is offered in most CSSL's, is not really modular. Better concepts are provided for example in DYMOLA, MODEL, and COSY.

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 which enables the software 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 to some negative values. Such additions have been proposed in [6].

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 in [3]. The definition for meta-models can be found in [7].

Programming: As explained in the introduction, errors should be detected as quickly 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 not yet reached most of the simulation software designers, because in todays simulation languages such features are hardly ever offered. This is probably due to the fact that most 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 school examples, declaration of variables even looks awkward because short user programs are a very strong argument, too. Nevertheless, as the art of modeling has matured, also the simulation software must follow.

Into the same category falls the use of LL(1) grammars for the language definition. LL(1) grammars allow to parse programs from left to right by looking just one symbol ahead. No backtracing 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 fully discussed in [1].

Another possible help is to check on model consistency and completeness. Some possible measures have been proposed in [8]. 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 infinit on a CDC 6000 series installation) if such a feature is available.

III. Simulation compilers

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

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. 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 (like the previously proposed dimensional analysis) are feasible.

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 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. It can, moreover, certainly not accept recursive algorithms. Nevertheless, so far FORTRAN seems to be the most appropriate language for coding the simulation run-time system, whereas good candidates for implementation of the preprocessor are either PASCAL or SIMULA-67.

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 to define the language, in general it poses more restrictions to the form of the language (grammar) than to the expressiveness of the language itself. 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 this context, we must either restrict ourselves in the macro facilities the language is to offer, or we must leave the pure LL(1) environment.

The reason, for using LL(1) grammars is explained below: 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. This problem is even more severe for simulation software as compared to general purpose programming languages since there exists a smaller number of users (speak: guinea-pigs) for them. When an arror is detected, 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 backwards 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 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.

IV. Simulation run-time systems

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

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 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.

This can be illustrated at an example. 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, howsoever, 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 tolerance 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 is not easily to be answered by the average user. In future, we should 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 first steps into this direction have been carried out, and are described in [4].

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 "insight 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" which rings 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 in [4].

V. Simulation systems

The term simulation system denotes 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 makes sense to discuss the robustness of a simulation system with respect to its:

Updatability: To make the documentation as easily updatable as possible, it is very conventient if also the documentation is developed by use of the computer. The text itself should (as this paper) 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, COSY, CSMP-III). In [1] 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 [2] 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.

There exist furthermore programs which can draw flow charts directly from the code of a program.

VI. 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 principally never happen. The simulation run-time system must take care to avoid this. Possible means have been discussed under the headings (simulation run-time systems, algorithms).

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 is wrong with his program(!). By restricting ourselves to LL(1) grammars, we can make sure that this shall never happen. This has been discussed under the headings (simulation languages, programming).

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, though. They have mostly to do with the introduction of redundancy which allows to perform different types of consistency checks. Some further thoughts to simulation program verification have been discussed in [9].

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. Further thoughts to model validation can also be looked up in [9].

VII. 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 modeling and ill-defined system modeling the classical CSSL-type language cannot suffice. We tried to list in a systematic manner different suggestions for improvement of future simulation systems which would meet the growing needs of modelers better.

VIII. References

[1] Bongulielmi, A. P.; Cellier, F. E.: On the usefulness of deterministic grammars for simulation languages. Proc. Sorrento Workshop on Internat. Standardization of Simulation Languages (SWISSL), Sorrento, Italy, 1979

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

[3] Cellier, F. E.; Fischlin, A.: Computer-assisted modeling of ill-defined systems. Proc. 5th European Meeting on Cybernetics and Systems Research, Vienna, Austria, 1980

[4] Cellier, F. E.; Moebius, P. J.: Towards robust general purpose simulation software. Proc. ACM SIGNUM Sympos. on Numerical Ordinary Differential Equations, April 3–5, 1979, University of Illinois at Urbana-Champaign, USA, 1979, pp. 18.1–18.5

[5] Elmqvist, H.: Manipulation of continuous models based on equations to assignment statements. Proc. 9th IMACS Congress on Simulation of Systems, Sorrento, Italy (L. Dekker, G. Savastno, G. C. Vansteenkiste, eds.); North-Holland Publishing Company 1979, pp. 15–21

[6] Elzas, M. S.: What is needed for robust simulation. Proc. Symposium on Methodology in Systems Modelling and Simulation, Rehovot, Israel (B. P. Zeigler, M. S. Elzas, G. J. Klir, T. I. Ören, eds.); North-Holland Publishing Company 1978, pp. 57–91

[7] Kleijnen, J. P. C.: Experimentation with models: Statistical design and analysis techniques. Proc. Course associated with SIMULATION '80 on Modeling and Simulation Methodology, Interlaken, Switzerland (F. E. Cellier, ed.); 1980

[8] Richards, C. J.: What's wrong with my model. Proc. 2nd UKSC Conference, England 1978

[9] Sargent, R. G.: Verification and validation of simulation models. Proc. Course associated with SIMULATION '80 on Modeling and Simulation Methodology, Interlaken, Switzerland (F. E. Cellier, ed.); 1980

Received: October 19, 1980

This paper was presented at the International Symposium on Systems Analysis, held in Berlin, German Democratic Republic, September 1980

Author's address: Dr. François E. Cellier,
Institute for Automatic Control, The Swiss Federal
Institute of Technology Zurich, ETH Zentrum
CH-8092 Zürich, Switzerland