Aplikace matematiky

Evžen Kindler
Block oriented simulation of combined systems

Aplikace matematiky, Vol. 31 (1986), No. 3, 169–179

Persistent URL: `http://dml.cz/dmlcz/104197`

Terms of use:

© Institute of Mathematics AS CR, 1986

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* `http://dml.cz`

31 (1986) APLIKACE MATEMATIKY No. 3, 169—179

BLOCK ORIENTED SIMULATION OF COMBINED SYSTEMS

EVŽEN KINDLER

(Received September 14, 1982)

Summary. A programming system BOSCOS is described, which permits the computer to transform descriptions of combined systems into corresponding simulation programmes. The combined systems are composed of continuous blocks and discrete processes. The blocks behave according to ordinary differential equations or assignment statements, or represent networks composed of such blocks; the structure of the networks can vary in time. Interactions between blocks and processes are possible. The system has been implemented as a knowledge base (a class of formal definitions) in an object oriented programming language SIMULA.

Keywords:

1. INTRODUCTION

1.1. Essence of combined systems

The influence of digital computers has caused that more and more complicated systems are exactly studied. One aspect of this trend concerns the so called combined discrete-continuous systems, simply called *combined systems*, which can be characterized by the presence of the following attributes:

1.1.1. a lot of parameters which develop continuously during the system time,

1.1.2. plentitude of structural properties which develop discretely during the system time,

1.1.3. mutual influence of the continuous parameter values on the structural properties and vice versa.

In more detail, any combined system has some components (aspect, subsystems, elements, processes etc.) — called continuous components — which are comparable with the so called continuous systems, i.e. systems which can be described by ordinary differential equations where time derivatives occur [22]. Further, any combined system has some components (called discrete components) comparable with what is commonly called discrete event dynamic system, i.e. systems with discrete events in generally non-equidistant time steps, during which not only discontinuous changes of some numerical and non-numerical parameters occur, but also relations among the elements change, new elements enter the system and other elements leave it, as in queuing systems [1]. During the discrete events, the discrete components affect the continuous ones by discrete changes of their parameters and – in case of the systems of combined type 3 (see 1.2.3) – the relations between the continuous components as well. The continuous components affect the discrete ones by causing the so called state events: such discrete events arise when a continuously developing parameter assumes a certain value (eventually in a certain direction). The parameters of continuous components can be distributed, i.e. their development can be describable only by means of partial differential equations, where also other than time derivatives occur, but the ordinary differential equations are commonly considered to be sufficient for the plentitude noted in 1.1.1 [30], [31].

1.2. **Classification of combined systems**

There is a hieararchy of three basic types of combined systems (see [2], [3], [4]):

1.2.1. Systems of combined type 1 (shortly 1-systems) are composed of one continuous component described by a fixed system of differential equations, and one or more discrete components, the number of which can vary in time.

1.2.2. Systems of combined type 2 (shrotly 2-systems) are composed of continuous and discrete components which behave as transactions [3], [4], i.e. their number and mutual relations can change during the system time but no continuous interaction can be present between different continuous components: a continuous component can only cause a state event in a discrete component and that component can further modify a continuous component in a discrete way.

1.2.3. Systems of combined type 3 (shortly 3-systems) whose continuous and discrete component behave as transactions similarly as in 1.2.2 but continuous transactions can mutually interact in a continuous way and so form another continuous transaction, called macro. As its structure can be modified by discrete events, its behaviour would be describable by a system of differential equations of a time-dependent form, including the number of equations; such systems have not been introduced into the language of mathematics.

The hierarchy described is essentially bound with the software for the modelling of combined systems: the programming languages specialized for the 1-systems (the GASP family, SAINT, SIMNON, SIMCOM/F, SMOOTH, SLAM) admit the users to describe the continuous components by means of differential equations which are written similarly as in the non-computer mathematics; the corresponding problems concerning the implementation of the continuous component models belong to numerical mathematics (integration methods, integration step control, state event detection etc.).

The programming languages oriented to the 2-systems (CADSIM, BLOCADSIM, PROSIM, SSL, NGPSS, GSL, CDCSIS – see [4], table 3, or [3], par. 6) offer the users similar facilities but must be combined with the dynamic allocation of data structures; the corresponding problems concerning the implementation of the continuous components overpass the numerical mathematics in the direction of the theory of programming: they are related to the memory economizing and to the language security; namely, the greatest problem is to signalize an error in case a parameter belonging to a continuous component becomes a coefficient in a differential equation describing another continuous component.

The number of the programming languages oriented to the 3-systems is not too great (NEDIS [5], a nameless one [6] and COSY [7], which has not yet been implemented); in addition to some obstacles concerning implementation, there is a great problem of the expressing means, which would enable the user to describe the time dependent structures of the continuous components; commonly the means are based on the analog computing technique language [5], where the continuous systems are considered as networks of simple elements: in order to arrange that language for complex systems, facilities for iterating such a network constructions to define more and more complex "macros" must be available. One therefore starts with the "elementary" blocks (integrators, adders, multipliers etc.), defines more complex ones (e.g. compartments, pools), then defines still more complex ones (e.g. a plate of a destillation colon, then a destillation colon), up to the whole system or continuous component. The creation and destruction of blocks in the structures (networks) is descibed similarly as the creation and destruction of a transaction in the discrete event simulation languages [1], [4], [5], [8]–[12].

1.3. The role of the modern programming languages

The problem of the best and most effective implementation of the programming systems for simulation of combined systems (the so called combined simulation systems) has not been solved. The reasons are in the numerical mathematics, theory of programming and in the applications: numerical mathematics is stimulated by the combined simulation languages and therefore offers new methods of numerical integration and state event detection (and one can expect that new methods will be offered in the future as well); moreover, it is not yet known what is the most suitable algorithm for transforming partial differential equations into arrays of ordinary differential equations, though such an algorithm will be very actual in the nearer future, due to the increasing importance of the systems with distributed parameters and discoverings in the theory of ordinary differential equations [7]; theory of programming owes a lot to the combined simulation languages, beginning with such "trivial" problems as standard outputs and ending with problems which have no analogy, as e.g. automatic modelling of elements which can switch between their inputs and outputs, similarly as electronic circuits or elements; in the applications, no statistics has been constructed informing on such relations as e.g. between linearity of courses of continuous parameters and complexity if arithmetic expressions occuring in the differential equations: such relations would make it possible to prefabricate the most effective configurations of various numerical methods.

In such a state of the art, any combined simulation system must be implemented in such a way to be relatively simply modifiable and extensible, i.e., it must be implemented as a set of modules which correspond to the notions of programming, description of (combined) systems and numerical mathematics. In other words, the implementation would be ideal, if it were a formalized theory, which legibly reflects the notions of system description, programming and numerical methods. Modern universal programming languages, namely SIMULA [9] – [12], reprezent a good tool for this pourpose, admitting to form stepwise richer and richer notions, reflecting both the mathematical languages as well as the natural ones. In the next part, a description of a combined simulation system oriented to the 3-systems (and, naturally to the other combined systems as well), based upon SIMULA, is presented. Compared with the other combined simulation systems based upon SIMULA (e.g. CADSIM [13], COMBINEDSIMULATION [14], SIMKOM/S [15] or DISCO [16]), the presented combined simulation system considers the classes of elements forming the combined systems (i.e. those used by the users) in a similar way as the classes of elements forming the numerical methods (i.e. those used by the implementors); such a technique is not only more effective, but it brings new discoveries in the theory of combined systems.

2. DESCRIPTION OF BOSCOS

BOSCOS (an acronym of Block Oriented Simulation of COmbined Systems) has been developed at Charles University in Prague. It is prepared to be applied, but at the present time it is being enriched by some facilities intended to further improve its function. It has been oriented to the 3-systems.

2.1. The hierarchy of basic BOSCOS concepts

The basic concepts form the following list, ordered according to the increasing content and decreasing extent; the relation between notions is not only suitable from the system theoretical view, reflecting the basic semantical links between the notions, observed already by the ancient Greeks, but also from the viewpoints of implementation, for they directly correspond to SIMULA classes and subclasses.

2.1.1. *cycle* seflects the elements, which iterate some action until they leave the system;

2.1.2. *cycle 1* reflects the elements which enter some list (queue) and at the end of the action mentioned in 2.1.1 resume their action to the next element in the list;

2.1.3. *block* reflects the *cycle 1* which sends a numerical *signal* to other elements;

2.1.4. *dynamic* reflects the simple elements which behave according to ordinary differential equations; with other elements taking part in the implementation, dynamic blocks are joined by means of real functions *function*, *VE*, *VG* and *VH*; they are declared as virtual, i.e. their meaning is defined only for special classes of the dynamic blocks;

2.1.5. *continuous* reflects the dynamic blocks for which the user himself defines the behaviour by means of a virtual procedure *equations*; in this procedure, the mathematical form of the equations is rewritten similarly as in CADSIM [13], i.e. *state* [*i*] means the value of the *i*-th continuous attribute of the block and *rate* [*i*] means its first derivative. If creating any continuous block, two parameters must be introduced: *order* of the equation, i.e. the greatest *i*, and *output*, i.e. the value of *i* for which *state* [*i*] represents the output *signal* of the block

The hierarchy described present abstracts tools which are not intended for the users of BOSCOS, with the exception of *continuous* and *block*; the last one reflects a general function generator, giving *signal* $= f(t)$; e.g. the sine wave generator can be introduced as a subclass of *block* in the following way:

```
block class wave (amplitude, omega, phi);
        real aplitude, omega, phi;
        signal := amplitude × sin(omega × t + phi);
```

2.2. Basic user oriented concepts

Similarly as a block, the users can apply the following concepts of a simple block with inputs:

2.2.1. *simple 1* reflects the simple block with one *input* carrying the output signal from another block;

2.2.2. *simple 2* reflects a block having a *second input* beside the *input* mentioned in 2.2.1; e.g. a multiplier is introduced in the following way:

```
simple 2 class multiplier;
              signal := input . signal × second input . signal;
```

For *simple 1* and *simple 2* the contents of the virtual procedure *set to* has been separately defined so that one can write e.g. *B . set to* (*A*) or *C* . set to (*A*, *B*), stating that *A* becomes respectively *input* of *B* or *C* and *B* becomes *second input* of *C*. After calling *set to*, its last parameter becomes *last* (i.e. it can be considered under the name *last*).

2.2.3. *integrator* reflects all dynamic blocks with the equation *rate* [1] := *state* [1], where *rate* [1] is the value of its input *signal* and its *output* is 1. In order to offer suitable basic concepts to the users, *integrator* is introduced as a special concept (subclass of *dynamic*) behaving similarly as *simple 1* (including *set to*) and not demanding to declare its equations, output and order. Concerning its implementation, see sec. 3.1.

2.3. **Concepts of macro**

2.3.1. *composite* reflects blocks with complicated internal structure; they contain an *output* block, the *signal* of which behaves as the signal of the whole composite block. The internal structure of the classes of "simiias" composite blocks is determined by procedure *form* which is a virtual one for *composite.*

2.3.2. *macro* is a composite block which takes place in the *form* procedure of some "greater" composite block.

2.3.3. *element* is a composite block which does not take place in the *form* procedure of any block; its interactions with the other components of the system are only through discrete events. It has its integration *method* and its integration step called *tstep* (because of SIMULA word **step**, it cannot be called simply *step*).

2.3.3. *macro 1* and *macro 2* are special macros, which are joined with other blocks, from which input signals come; *macro 1* has one input, comming as *signal* from the block denoted *input*; in addition, *macro 2* has its *second input,* comming as *signal* from into the block denoted *second input.*

As an example, we present two equivalently behaving declarations of the concept of block realizing the equation $y' = y^2 - \sin(3t)$:

```
continuous class C; begin procedure equations;
          rate [1] := state [1] ↑ 2 − sin (3 × t); end;
macro class D; begin procedure form; begin
       output: − new integrator; output . set to (new adder);
       last . set to (new wave (3, 1, 0), new multiplier);
       last . set to (output, output) end;
       end:
```

3. IMPLEMENTATION OF BOSCOS

3.1. **Integration methods**

BOSCOS has the following methods for the numerical integration [17]: rectangular (Euler), Simpson, Newton-Cotes, Runge-Kutta 4th order with fixed step, Runge-Kutta 4th order with controlled step (Runge-Kutta-Simpson), Runge-Kutta--Merson, Runge-Kutta-England, 3/8-Runge-Kutta and Shampine-Euler for stiff systems [18]. Each of them is a SIMULA class, i.e. an abstract notion. If one wishes to use one or more of the methods, one creates an instance existing as an individual entity of the simulated system. Such an instance can be used by any *element* as its *method* (see 2.3.3). The parameters of any method with a controlled step are the minimum and maximum value of the step. SIMULA automatically permits an element to change its method during a discrete event.

After being created, every instance of a method creates automatically its phases. Then it only controls their work determining one of them to be the actual *phase* and *possibly* modifying the value t reprezenting the independent variable.

The active step (see 2.1.1) of every dynamic block is composed of the following two simple actions:

3.1.1. the block itself takes name *present*;

3.1.2. the block resumes the control to the *phase*. Then the *phase* performs what is to be done in the actual phase of the integration step; after this, the *phase* should resume the control back to the *present* block, but as it is a rule (see 3.3) that this block resumes the control immediately to its successor, the same action can be done already by the *phase*; thus all phases of integration methods are specializations of the concept called *meth*, which is a subclass of *cycle* (see 2.1.1), enriched by the statement *resume* (*present* . *suc*).

At the end of 2.2, a comparison of integrators and continuous blocks has been presented with a statement that it is convenient to consider the *integrator* as a concept which is not a specialization of the *continuous* block. This is reflected in the implementation of the integration methods, where all the introduced concepts have two distinct versions – one for the integrators and the other for the continuous blocks; it concerns not only the phases of the integration methods, but the common concepts as well; for the mnemonic reasons, the identifiers mentioned above are used for the integrators and those used for the continuous blocks have the prefix *c* (*cpresent*, *cmeth*, *cphase*).

Let us mention that various methods share common integration phases, which makes the implementation economical.

3.2. State event detection

The existing languages for the modelling of combined systems determine the exact time of the state events by the step halving method, which is robust and simple for programming. Nevertheless, other methods exist, which can be very effective in certain situations [7] but can give false results without any warning in other situations; such methods, as e.g. Newton's method or that of inverse Hermite's polynomials, have been recommended for the implementation and are intended in the language COSY [7], though they are complicated for programming: they would be suitable as alternative methods to that of step halving. The SIMULA base of BOSCOS has allowed to abstract all the common aspects of the possible methods for state event detection and join them with the abstract notion of *flag*, which is a specialization of *cycle* (see 2.1.1):

flag contains facilities for determining some value of t between some values t_1 and t_2, and for the iteration where the value of t is assigned to t_1 or t_2 according to certain conditions. The determining of t requires some virtual arithmetics, which is made precise in the specializations of *flag*; BOSCOS has the following of them:

halving (step halving method), *regula falsi* (regula falsi method) and *Hermite.* (method of inverse Hermite's polynomials), but also other ones are admissible, as e.g. Newton's method. The corresponding arithmetics uses virtual functions of dynamic blocks, mentioned in 2.1.4. Every flag has some parameters determining the maximal error for the state event detection, maximal number of iterations during one state event detection, etc.

Every *element* (see 2.3.3) has a virtual procedure *post step* which is performed always after finishing an integration step, i.e. when the following two conditions are satisfied:

the element is just developing continuously,
i.e. its integration step has not been performed inside the iteration for a state event detection;

in this development the integration step just performed has been taken as finished, i.e. the accuracy test has not been negative.

Procedure *post step* can be let empty in case the development of the element should not be disturbed by any state event (e.g. in case of continuous simulation), but for the state event detection it is used to determine whether during the performed integration step a state event has not occurred: in the affirmative case, a flag is activated to perform its work. In [19] it is shown that in every such case *post step* can call one flag existing during the system's whole existence as the only element of class flag, or every state event can have its own *flag*; both the possibilities have their advantages and disadvantages. Naturally, *post step* can be used for quite different purposes, like outputs of results.

3.3. Composite blocks

Let A be a composite block and B a block created in the *form* of A. We say B is a node of A. An entity is called *storage* if it is a dynamic block or if it is a composite block with a node which is a storage. Every composite element has a list *backbone* and in case it is a storage it has another list *stomach*. Procedure *sort*, which is automatically applied to every composite block X, forms its backbone and sorts into it its nodes which are not storages; in case X is a storage, *sort* creates the stomach of X and puts the nodes of X which are storages into this stomach. In the last positions of the lists mentioned, special elements are placed, which return the computation either to the integration method (see 3.1) in case the list in question belongs to an element (see 2.3.3) or to the next element of the corresponding list of the composite block for which X is a node. These special elements belong to certain subclasses of class *cycle* (see 2.1.1).

In case an integration method requires to compute the right hand sides of the corresponding differential equations, it resumes the control of the computation to the first element of the backbone of the element concerned (in the sense of 2.3.3). The action joined with the class *macro* begins by resuming the control to the first element of its backbone. The special elements at the end of the backbone, mentioned above, cause that the whole hierarchy of the backbones of the same element is step-wise activated and after the computation of the right hand side the computing process returns to the integration method. In case an integration method requires to compute something which concerns only storages (namely, to store their signals to some of their auxiliary attributes) it resumes the control of the computing process to the stomach of the element in question and the hierarchical process is performed similarly. Let us mention that every composite storage which is a node of X is placed in the backbone of X and therefore the procedure *sort* applied to X automatically creates a special *card* and places it into the stomach of X in place of the storage itself; the card is an element of a subclass of *cycle 1* (see 2.1.2).

Procedure *set to* (see 2.2) is declared for various classes of macros so that it calls *form* and *sort*. The same procedures are also included at the beginning of the action of every *element*. Procedure *sort* should be called also in case the structure of a composite block is essentially modified during its existence in the system. Calling *sort* during the computation (and not during the compilation) makes BOSCOS a combined simulation system oriented to the 3-systems (and naturally also to the 1- and 2-systems).

In the composite blocks, implicit elements can be used making it possible to form algebraic (implicit) loops (see e.g. [20]); the implicit elements behave as elements of a subclass of class *simple 1* (see 2.2.1) and use Wegstein's algorithm, which is considered as robust [21], [22].

4. CONCLUSIONS

BOSCOS has no standard outputs, because of the absence of experience: the existing combined simulation systems are facilitated only by standard outputs commonly known from continuous system simulation. At present time, the principles mentioned in 3.1 are studied for being adapted for multistep integration methods and for combining implicit integration principles with traditional explicit algorithms, similarly as mentioned in [18]. These tasks could mean an important contribution to the systematic algorithmization of numerical methods: the expressing of structural algorithmical properties of the numerical integration methods is far from being so exact as their analytical treatment concerning their accuracy and stability, and SIMULA seems to be an important tool in this development.

All concepts mentioned in Parts 2 and 3 have been declared as attributes of class BOSCOS. This class has been designed as a subclass of SIMULA standard class SIMULATION, i.e. the definition of BOSCOS has been introduced by the following declaration:

SIMULATION **class** BOSCOS; **begin** ⟨classes of Parts 2.3⟩ **end** ;

This fact enables the users to use in BOSCOS any tool introduced in SIMULATION;

thus one can make use of the list processing tools and of discrete event scheduling on a simulated time axis. Therefore we have not described those facilities in the present paper. Moreover, in BOSCOS one can make use of any subclass of SIMULATION: one can arrange it simply so that instead of the "prefix" SIMULATION one writes another prefix before the mentioned text **class** BOSCOS. The author has made a good experience especially with using prefixes GPSS and TRANSPORT. The first one [23], [24] introduces all common facilities concerning the queuing systems, the second one [25], [26], [27] introduces all tools concerning the material flow systems [28], [29].

Literature

[1] *O.-J. Dahl:* Discrete event simulation languages. Norsk Regnesentralen, Oslo 1966. Reprinted in: F. Genuys (ed.): Programming languages. Academic Press, London, N. York 1968.

[2] *D. A. Fahrland:* Combined discrete event continuous systems simulation. Simulation, 14 (1970), pp. 61—72

[3] *E. Kindler:* Classification of simulation programming languages. Part II: Description of types and individual topology. Elektronische Informationsverarbeitung und Kybernetik, 14 (1978), pp. 575—584.

[4] *E. Kindler:* Simulation programming languages (Czech). SNTL, Praha 1980. Языки моделирования. Атоменергиздат, Москва 1985.

[5] *V. M. Gluškov, V. V. Gusev, T. P. Marjanovič, M. A. Sachňuk:* Programming means for the modelling of discrete-continuous systems (Russian). Naukova Dumka, Kiev 1975.

[6] *P. Medow:* Private letter to the author. York University, Downsview, Canada, 1978.

[7] *F. E. Cellier:* Combined continuous/discrete system simulation by use of digital computers. ADAG, Zurich 1979.

[8] *O.-J. Dahl, B. Myhrhaug, K. Nygard:* Common base language. 4th ed. Norsk Regnesentralen, Oslo 1985. SIMULA 67, универсальный язык программирования. Mir, Moskva 1969.

[9] *G. Lamprecht:* Einführung in die Programmiersprache SIMULA. Vieweg, Braunschweig 1977.

[10] *G. Lamprecht:* Introduction to SIMULA 67. Vieweg, Braunschweig 1981.

[11] *Z. Benda, J. Staudek:* Programming in SIMULA (Czech). SNTL, Praha 1978.

[12] *G. M. Birtwistle et al.:* SIMULA begin. 3. ed. Studentliteratur, Lund 1976.

[13] *R. J. W. Sim:* CADSIM users guide and reference manual. Imperial College, London 1975.

[14] *K. Helsgaun:* COMBINEDSIMULATION — a SIMULA class for combined continuous and discrete simulation. Proceedings Sixth SIMULA Users' Conference, Lisbon 1978, pp. 30—35.

[15] *J. Fischer, G. Schwarze:* SIMKOM-S/80 — A simulation system for combined models. SIMULA Newsletter, Vol. 9, (1981), No. 4, pp. 14—16.

[16] *K. Helsgaun:* DISCO — a SIMULA-based language for combined continuous and discrete simulation. Simulation, 35 (1980), pp. 1—12.

[17] *A. Ralstone:* A first course in numerical analysis. Mc.Graw-Hill, Inc., N. York 1965.

[18] *L. F. Shampine:* Efficient use of implicit formulas with predictor-corrector error estimate. Journal of computational and applied mathematics, 7 (1981), pp. 33—35.

[19] *V. Červenka, E. Kindler:* Detection of state events in combined discrete-continuous dynamic systems. Aplikace matematiky, in print.

[20] *G. Schwarze:* Simulation — kontinuierliche Systeme. Technik, Berlin 1976.

[21] *P. Rechenberg:* Die Simulation kontinuierlicher Prozesse mit Digitialrechnern. Vieweg, Braunschweig, 1972.

[22] *M. J. Shah:* Engineering simulation using small scientific computers. Prentice-Hall, Englewood Cliffs 1976.

[23] *A. Mojka, J. Šplíchal, E. Kindler:* A contribution to the simulation of steel plant — rolling mill. In: Proc. 5th SIMULA users' conference. Norsk Regnesentralen, Oslo 1977, pp. 22—28.

[24] *A. Mojka, J. Janda, J. Šplíchal, E. Kindler:* Simulation models of complex production systems (Czech). ASŘ sešity INORGA, No. 39, pp. 156—160.

[25] *K. Prokop, Š. Chochol:* TRANSPORT. Systems of material flow. Aplikace matematiky, 28, 1983, No. 2, pp. 156—160.

[26] *E. Kindler, K. Prokop, Š. Chochol:* Dynamic modelling of transport in agricultural systems. Elektronische Informationsverarbeitung und Kybernetik, 17 (1981), Heft 11/12, pp. 645—657.

[27] *E. Kindler, K. Prokop, S. Chochol:* Systems of material flow. COMTEX Series on Systems Science and Engineering, New York 1983.

[28] *E. Kindler, S. Chochol, K. Prokop:* Systems of material flow. International journal of General Systems, Vol. 9 (1983), pp. 217—223.

[29] *E. Kindler, S. Chochol, K. Prokop:* Theory of material flows and its use for simulation. In: 27. Internationales wissenschaftliches Kolloquium, Heft 6. Technische Hochschule, Ilmenau 1982, pp. 133—136.

[30] *B. Zeigler:* Theory of modelling and simulation. J. Wiley, New York 1979.

[31] *G. A. Mihram:* Simulation — statistical foundations and methodology. Academic Press, New York 1972.

Souhrn

BLOKOVĚ ORIENTOVANÁ SIMULACE KOMBINOVANÝCH SYSTÉMŮ

EVŽEN KINDLER

V článku je popsán programovací systém BOSCOS, který umožňuje strojový překlad popisů kombinovaných systémů do odpovídajících simulačních programů. Kombinované systémy se skládají z bloků a diskrétních procesů. Bloky se chovají dle obyčejných diferenciálních rovnic nebo dosazovacích příkazů, nebo představují sítě složené z takových bloků. Jsou možné interakce mezi bloky a procesy. Systém byl implementován pomocí báze znalostí (třídy formálních definic) v objektově orientovaném programovacím jazyku SIMULA.

Резюме

СТРУКТУРНО ОРИЕНТИРОВАННОЕ ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ НЕПРЕРЫВНО-ДИСКРЕТНЫХ СИСТЕМ

EVŽEN KINDLER

В статье описана система программирования БОСКОС, которая ориентирована на машинный перевод непрерывно-дискретных систем и их описаний в соответствующие имитационные программы. Непрерывно-дискретные системы конструируются из блоков и дискретных процессов. Динамика блоков описана обыкновенными диференциальными уравнениями или операторами подстановки, или при помощи сети подобных блоков. Существуют интеракции между блоками и процессами. Система была реализована на языке СИМУЛА.

Author's address: RNDr. PhDr. *Evžen Kindler,* CSc., Katedra aplikované matematiky na MFF KU, Malostranské nám. 25, 118 00 Praha 1.