SUPER-OBJECT-ORIENTED PROGRAMMING AND MODEL NESTING

Eugene Kindler

University of Ostrava, Faculty of Science,
701 03 Ostrava, 30. dubna 22, Czech Republic

ekindler@centrum.cz

Abstract

While the term object-oriented programming (OOP) settled down the paradigm of classes as encapsulations of data and methods (procedures), specialization of classes (subclasses) and virtuality of the methods (late bindings), there are further offers related to OOP but enhancing it in essential way; they are sometimes called super-object-oriented programming (SOOP). Curiously, SOOP arose with OOP and rooted more tightly in simulation than OOP itself. Nowadays, after 40 years of existence of both paradigms, one can observe essential contributions of SOOP, among which there is simulation of intelligent systems having elements (computers, persons) that create and use their "private" (simulation) models for decision support; that allows deciding with respect to possible future consequences. The properties that enrich SOOP above OOP will be in detail explained (namely: life rules, block structure, classes local in other classes and in blocks, and quasi-parallel control), their relation to general aspects of computer simulation (namely to the process paradigm) will be exposed and existing applications will be presented. The participants will get a free and efficient PC implementation of SIMULA programming language, in which particular examples will be formulated.

Keywords: Object-oriented programming, Super-object-oriented programming, Simula, Intelligent systems simulation.

Presenting Author's biography

Eugene Kindler studied mathematics at Charles University in Prague, concluding with grades of Doctor of philosophy (in logic), Doctor of sciences (in theory of programming) and (from Czechoslovak academy of sciences) Candidate of sciences in physics/ mathematics. At Prague Research Institute of Mathematical Machines (1958-1966), he participated at the design of the first Czechoslovak electronic computers and designed and implemented the first Czechoslovak ALGOL compiler for it. At Biophysical Institute at the Faculty of General Medicine of Charles University (1967-1973), he designed and implemented the first Czechoslovak simulation language and then introduced the object-oriented programming into Czechoslovakia. Nowadays, as professor emeritus of applied mathematics, he teams up with Ostrava University. As visiting or invited professor, he worked at the University in Italian Pisa, at West Virginia University in Morgantown, at the University of South Brittany in French Lorient and at University of Blaise Pascal in French Clermont-Ferrand. His main interest is simulation of systems containing elements that use simulation models.

ISBN 978-3-901608-32-2

1 Prehistory of OOP

1.1 Process-oriented simulation languages

The is a common understanding that programming simulation models can be made easy so that instead of description what should happen in the simulating computer one describes what should happen in the simulated system such a description in a suitable simulation language can be automatically converted into an executable program. And it was not later than in 1961, when the author of simulation language GPSS [1] realized that the dynamics of simulated discrete event systems can be described as a result of parallel ***processes***, each of them having some data (***attributes***) and being subjected to its ***life rules*** that can be expressed by tools for controlling algorithms; processes with similar attributes and life rules belong to common ***classes***. The life rules function in time flow common for all processes; therefore, at a mono-processor programmed computer, they are interpreted as switching one to another according to ***scheduling statements***.

The rather primitive programming tools of GPSS were elevated up to the common state of the 60 years algorithmization practice in 1964 in SOL [2] and especially in 1965 in SIMULA [3]. They can be characterized by means of the following principles:

(A) the classes are formulated as encapsulation of attributes and life rules;

(B) similarly as variables local in blocks on ALGOL 60 [4] or in procedures and in program modules, the attributes have names and types;

(C) in the life rules, the sorts of the statements, which exist in common algorithmic languages (assignments, branchings, cycles, …), are feasible;

(D) scheduling statements can be among the life rules, where they can mean e.g. "hold until the simulated time accesses a given value" or "hold until a signal for continuing comes";

(E) procedures (subroutines, functions) can be declareed in classes so that the life rules of a class can call them;

(F) a class is a source for generating ***instances***; their number is a priori unlimited.

Let a class satisfying (A)-(F) be called ***p-class*** in the present paper.

Let us illustrate the life rules of a p-class *cell* of cells, which passes through three states *S*, *G* and *M*, in each of them remains during a time, the duration of which is a random value of normal distribution with mean value *A* and sigma *B*, and – after leaving state *M* – dies (with a probability *P*) or multiplies (otherwise):

L: into(S); hold(normal(A,B,U)); into(G); hold(normal (A,B,U)); into(M); hold(normal(A,B,U)); if draw(P,U) then begin activate new cell; go to L end;

Note that multiplying of a cell is considered in a form that the multiplying cell generates and activates another cell and repeats its life rules from the start. The states are interpreted as sets and it allows e.g. introducing a model of an appliance tracing the numbers of the cells in various states any time divisible by *K*:

while true do begin hold(K); outint(cardinal(S),5); outint(cardinal(G),5); outint(cardinal(M),5); nextline end;

The languages following the principles (A)-(F) are called ***process-oriented simulation languages***. They differ from the ***event-oriented simulation languages***. If one wishes, he can use a process-oriented language as an event-oriented one, but in the opposite way that is not possible.

1.2 Hoare's Data Structures

One of the authors of SIMULA, O.-J. Dahl, was invited to take a lecture [5] at the NATO Summer School on Programming languages [6]. There he met another lecturer, C. A. R. Hoare who spoke [7] on hierarchical data structures, introducing the following principles:

(G) a data structure is composed of data, identifiable according to their names; any component of such a structure has its type, either a conventional one (real, integer, Boolean, text etc.) or ***reference*** one;

(H) the data structures are ***instances*** of classes, which serve as definitions of the names and types of their components; in a class, any reference component gets its ***qualification***, i.e. a class *Q*; the values of that component have to be either "none" or instances of *Q*;

(I) if *X* is a name of a data structure which has component *Y*, then the ***dot notation*** allows to express "component *Y* of data structure *X*"; dot notation can be iterated; e.g. if *Y* is a reference component qualified into class that has a component *Z*, then *X.Y.Z* is meaningful;

(J) a ***subclass*** *C* of a class *D* can be introduced by explicit formulating that all components of *D* figure as those of *C*, too; while *C* can have other components that have no origins at *D*; an instance of *C* is also an instance of *D*; *D* is called ***prefix*** or ***superclass*** of C forming a subclass of *D* is called ***specialization*** of *D*;

(K) any class is open for any number of its instances and for any number of its specializations.

Let a class satisfying (G)-(K) be called ***d-class***.

2 Step to object-oriented programming

The sets (A)-(F) and (G)-(K).of the principles offer to be included in a certain synthesis of them.

(L) reference attributes can occur among the attributes of p-classes;

(M) the attributes of an instance of a p-class can be identified by means of the dot notation;

ISBN 978-3-901608-32-2

(N) the p-classes can be specialized similarly as the d-classes; beside adding new attributes, new life rules can be subjoined to those of the prefix, either after its last life rule or at a place explicitly marked in it.

According to Dahl's oral statements (and some printed ones, as e.g. in [8-9]), he was due to Hoare and his paper for discovering the principles of OOP. Nowadays, such Dahl's statements reflect more his modesty than the historical development. In fact, the synthesis mentioned above does not lead to what was accepted as an essential component of OOP and what was later called methods.

Note that the character of dot in the don notation does not need to be dot, but e.g. – like in Smalltalk – space.

3 Object-orientation

A new step to OOP was performed by introducing the following two principles:

(O) Procedures mentioned in (E) can be called by using dot notation. Later, such procedures were called ***methods*** and the statements for calls of them were called ***messages***. New methods can be added to subclasses.

It was an essential step to OOP, logically independent on the synthesis mentioned above, but not yet sufficient. The next step concerned ***late bindings***:

(P) The contents of a certain procedure introduced for a class can be (re)formulated in any subclass; such a procedure is called ***virtual***.

Virtuality became the last principle for characterizing what is called ***object-oriented programming***. Summarized, this programming paradigm is based on classes as encapsulations of attributes and methods.

4 Super-object-oriented programming

4.1 Simula 67

The new language, i.e. the old SIMULA enriched by aspects mentioned under (G)-(P) (and by other tools described further as (Q)-(U)) was called SIMULA 67. Although it was the first OOP language, since its first international presentation in [10] SIMULA 67 has offered more than that covered by term OOP; namely life rules and their mutual switching (see principles (C) and (D)) enriched the supply of the OOP tools. Note that they offer SIMULA 67 as an excellent process-oriented simulation language, or – more precisely – as a base to define many more or less independent process-oriented languages. The simulationists who use it do not suffer similarly as those applying standard OOP languages, i.e. they do not need to formalize their models under a paradigm of event-oriented programming (see the end of part 1.2), i.e. to destroy the processes into heaps of events.

The first presentation of SIMULA 67 appeared at a conference on the simulation programming languages [10], but at the same meeting an idea arose that what this language offers was applicable outside simulation, too. After some years, OOP was really accepted as an excellent paradigm of programming in a general sense. Already since 1967, SIMULA independence of simulation has been reflected also in its principle that eliminated the "absolute" importance of the scheduling statements for the switching (i.e. the owning of switching to the simulated time):

(Q) For switching among the life rules, general tools called ***sequencing statements*** are offered, while any scheduling statement can be understood as a procedure defined with use of them; some of such procedures are offered as standard procedures but any SIMULA 67 user can define his own procedures for such a purpose.

Another principle introduced in SIMULA 67 relates to life rules, too:

(R) The goal of a transfer inside the life rules can be virtual: among the life rules of a class, statements can occur that transfer the continuing of the instance "life" to a statement that is not just yet among the life rules of the class but is expected to occur in some subclass.

4.2 Block structure and local classes

Like the old SIMULA, also SIMULA 67 based its algorithmic tools upon those of ALGOL 60 [4], which was a perfect block-oriented language. Soon after 1967, block orientation was condemned by the gurus of programming theory, because it seemed being in contradiction with the paradigm of modular programming, which was modish in the seventies and eighties of the last century. Even after omitting that paradigm, the ignorance of the importance of blocks remained and therefore it was not sooner than in the present century when this importance is slowly penetrating into the programmers' and simulationists' minds, discovering a fascinating synthesis with the object orientation. The synthesis roots in the fact that a declaration of a class has the same context as any other declaration; it can be characterized by the following principles:

(S) similarly as variables and subroutines local in blocks occurring in the former block-oriented languages, SIMULA 67 admit classes to be local in blocks; the true block orientation views entities with the same names but local to different blocks as different entities (their homonymy has no importance);

(T) classes can be declared like attributes for a class; then it is called ***main class*** and the mentioned classes are called ***nested classes***; the instances of a main class represent world viewings, formal theories, models or formal languages, while the nested classes represent concepts or knowledge applied in that world viewings, theories, models and languages; two instances of the same main class can represent two different vies at the same "world", two different models of the same object, judgments pronounced by different observers of the same object, or two theories differing by their parameters;

ISBN 978-3-901608-32-2

(U) the contents of a class can be introduced into a block by prefixing it by the class; such a block is called ***prefixed block***; if the "life" of an instance of *C* enters a block prefixed by a main class the instance becomes a model of something that enters its life phase enriched by an ability of a world viewing (thinking, expressing, modeling) using the contents of the main class; if the lives of more instances of *C* are in such a phase they can apply their own attributes when using the contents of the prefixing class.

Rational synthesis of block orientation and object orientation needs the process orientation. The fruits of this synthesis form a programming paradigm called ***super-object-oriented programming*** (SOOP) [11-12].

Suppose a block *B* occurs among the life rules of class *C* nested inside main class *M*; suppose further a main class μ is local in *B*; an instance *X* of *M* can be viewed as a model of a system *S* described by *M*, and an instance *Y* of *C* can be viewed as an image of a component γ of *S*. When *Y* enters *B* it reflects that γ has got an ability to model (or "think on") system σ, expressed by means of μ. If *Y* generates an instance ξ of μ it reflects that γ manipulates with a model ξ of a certain system σ.

Such a situation is outlined in Fig. 1, where squares represent instances of a main class, circles represent instances of the other classes, a horizontal incise represents the "life" (flow of performing life rules) of the instance represented by the surrounding circles, a rectangle with rounded edges represents a block and the relation of nesting a graphical construct *z* inside another one w represents that the object represented by *z* is local to that represented by *w* (in other words: that the object represented by *z* has access to that represented by *w*). *A > D(E)* represents that *A* is modeled by an instance *D* of class *E*. The circles with digits represent elements existing in the same system *S* where γ exists (or images of their elements, existing in the same model *X* of which *Y* is a component) and the small circles represent elements of system σ or their images, i.e. components of model ξ.

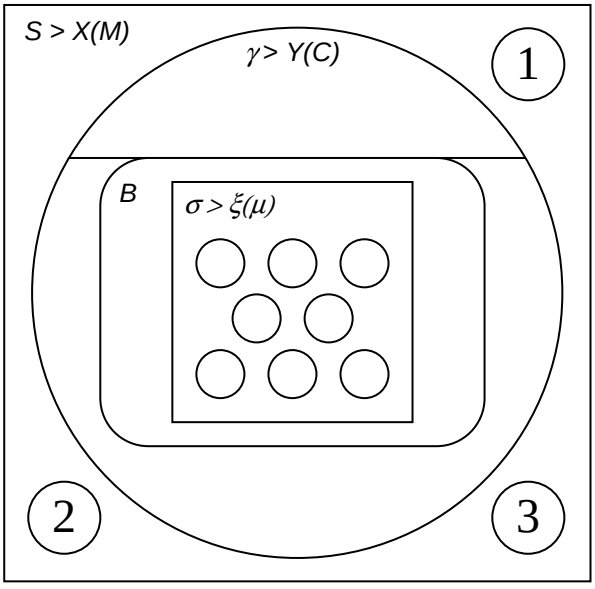

Fig. 1 Nesting of models

In such a case, one says that model ξ is ***nested*** inside *X* and one speaks on ***nested modeling*** or – if *M* and *m* serve for simulation – on ***nested simulation***. If *M* is similar to *m* an instance like *y* can be an image of an element that has ability to observe its environment (i. e. system *S*) in that it is being, to reflect the observed pieces of knowledge in model of the environment of γ, and to apply it e.g. for generating information on the future of *S*. In such a case (i.e. in this special case of nested modeling), one speaks on ***reflective modeling*** or ***reflective simulation***

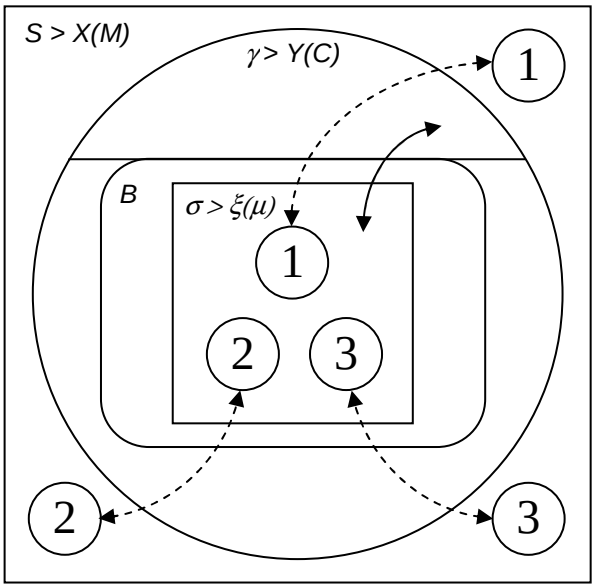

Fig. 2 Reflective modeling

Fig. 2 is a symbolic illustration of such a reflective simulation – when model ξ exists its elements can communicate with corresponding elements of model *X* (see the dashed arrows) and even both the models can communicate (see the full arrow).

There is a small number of programming languages that satisfy some principles leading from OOP to SOOP. For example MODSIM [13] and NEDIS [14] are object-oriented and process-oriented simulation languages, i.e. languages satisfying – among other – (C) and (D), but not (Q) – (U). JAVA is block-oriented but its tools that should draw near principle (O) are rather wooden.

Only SIMULA 67 [15] and Beta [16] appear to satisfy all the mentioned principles. The syntax rules of Beta are rather strange and separate it from current use. When SIMULA 67 became an ISO standard in the eighties of the XX century [17], the increment 67 was refused, as the old simulation language SIMULA fell into oblivion, being replaced by SIMULA 67 at its hitherto users. The syntax rules of SIMULA (67) have followed usual customs and therefore it is suitable as a base for starting with SOOP; another advantage of

SIMULA is that a lot of experiences were obtained with it, which allows solving many obstacles related to the fact that the synthesis of block structure with object orientation leads to something like nesting formal theories (or like formal theories, the elements handled by which can be carriers of other theories) – note that some discoveries how to make something what had been considered as unfeasible, came after tens of years of the language analysis [18]. Another property of SIMULA, which appears especially suitable for programming models, is its complete separation from what could happen inside the used computer: although that makes problems when one would wish to apply SIMULA for programming of some software deeply concerning the computer run (e.g. an operation system), that allows this language to be safely applicable for knowledge representation on such models and for model portability. Last but not least, SIMULA serves well because there are real applications of its highest principles.

5 SIMULA

5.1 Common rules

The fundamental concept of SIMULA is ***block instance***. It is a component of computing process, which has its ***local entities*** and its ***life***. It is described as ***textual block***, which is closed in "brackets" *begin* and *end* and composed of two parts, which describes the local entities and the life. The first part is a sequence of ***declarations***. The following sorts of declaration are important: variable declaration, procedure declaration and class declaration.

Variable declaration has form like *real x, a, ww* which tells that variables *x*, *a* and *ww* can be viewed as carrying real values; in place of *real*, other key words can occur like *integer*, *Boolean*, *character* and *text*. ***Reference declaration*** is another sort of variable declaration; its form is like *ref(C) q, s*, where *C* is a class and the declaration tells that *q* and *s* either may point to an instance of class *C* (or its subclass) or to nothing (identified as *none*)

Procedure declaration is composed of its heading and body. The ***heading*** has a form like *procedure G* or *procedure F(u,w); real w; text u;* introducing procedures called *G* and *F*, *G* being without parameters and *F* with two parameters, specified as text and real ones. Other components can be in the heading, concerning the style of calling. A procedure can be a function (i.e. gives a result and can be called in expressions); then the type of the result is defined in front of key word *procedure* by using the same key words as in the variable declarations. The ***body*** of a procedure is any statement (see further), nevertheless it is often a block.

Class declaration is similar, only in place of *procedure* a key word *class* occurs and some other possibilities are offered for the heading (e.g. specifications of virtual entities). If the introduced class is a subclass of a class the name of that "superclass" is to be put in front of word *class*.

The life of a block is described as a sequence of ***statements***. Examples of statements:

Assignment statement is like *a:=b* for assigning value *b* to variable *a*, or like *a:-b* for assigning reference *b* to *a*. At the right part of such statements, expressions can occur. Common usages for arithmetic expressions are respected, Boolean operations are expressed by key words like *not*, *and*, *or*,... relations serve for computing Boolean values from numerical, character and textual ones, and conditional expression (*if b then x else y*) serves for the opposite conversion. Constants and function calls are permitted, too. A lot of standard functions exist like *sin*, *ln* or *log10*, and among them there are those for generating pseudorandom values.

Procedure statement has a usual form *F(a,b,c)* where an expression of form *R.F* can occur in place of *F*, telling that procedure *F* should be performed by the object pointed by *R* (*R* can a more or less complex reference expression, possibly in brackets).

Branching is expressed by a statement like *if b then S* or *if b then S else T*, where b is a Boolean expression and after instantaneous evaluation of it, statement *S* is performed in case the evaluation gives *true* (logical I). In the opposite case (if the evaluation of *b* gives *false*, i.e. logical O), the second form of the statement permits performing the statement following *else*.

Textual block (see above) is a sort of statement and can occur among the life rules. A text that is similar but contains no declaration after *begin*, is called ***compound statement*** and serves for gathering statements to figure as one statement.

Jumps in the sequence of the life rules is allowed by performing a statement of form *go to L*, where *L* denotes the target of the jump; the place of the target is given by "label" of form *L* followed by a colon. It is not possible to jump inward a statement, therefore neither inward a block.

Other SIMULA sorts of statements (like cycles) exist but their explaining is omitted in this paper. Let us express the general rudders.

The users give names to the entities they introduce, using ***identifiers*** that begin with a letter that can be followed by any letters, digits and sign of underline. The identifiers have to differ from the key words. All names of variables, procedures, classes and labels should follow these rules.

Let *C* be a class declaration. In the statements occurring in it (i.e. in the life rules of the class and in those of any textual block nesting in this declaration (e.g. in procedure declarations), the expression *this C* (called ***local reference***) points to the instance of *C*, which is just performing the statement where the local reference occurs.

ISBN 978-3-901608-32-2

If we introduce a common concept phrase for covering the statements and the declarations, then it is possible to state that the phrases should be separated by semicolons. Note that the statements that are life rules of a textual block should follow the declarations of the same textual block.

5.2 Block orientation

Every block instance has its ***program sequence control*** (PSC), which points to the life rule that is to be performed. PSC is important when another block instance arises or disappears. The dynamics of rise and disparition of block instances respect the following rules.

(1) The program has a form a textual block and the corresponding block instance exists during the whole existence of the corresponding program rum.

(2) When the life rules belonging to a block instance *J* enter a textual block *B* a block instance *K* corresponding to *B* arises and is ***attached*** to *J*. More exactly, *K* is called ***subblock instance***. PSC of *J* is set to the statement that follows *B*.

(3) When the life rules belonging to block instance *J* enter a procedure call, a block instance *H* corresponding to the body of the called procedure arises and is attached to *J*. More exactly: *H* is called ***procedure instance***. PSC of *J* is set to the step that should follow the procedure call.

(4) When the life rules belonging to block instance *J* enter an expression like *new C*, where *C* is a name of a class, a block instance *G* corresponding to the body of the declaration of class *C* arises and is attached to *J*. More exactly, *G* is called ***class instance***. PSC of *J* is set to the step that should follow the generating *G*. A custom exists to speak on ***object*** in place of class instance, on ***attributes*** in place of local variables and on methods in place of procedures.

(5) When a block instance *X* arises, the computing switches to the necessary administrating of it and then it goes on according to the life rules of *X*. Suppose *X* is attached to *Y*. When the "life" of *X* is exhausted the computing switches to place of the life rules of *Y*, pointed by PSC of *Y*.

(6) Only class instances can get names (for example by means of assignment *R:-new C*). The subblock instances and those of procedures become inaccessible as soon as they exhaust their "lives".

(7) In a textual block *A* that is among the life rules of a textual block *B*, the entities accessible in *B* are accessible. The same holds when *A* is a body of a procedure or of a class declared in *B*.

(8) If *A* and *B* are two block instances generated according to the same textual block *T* the sets of their local entities are quite different. In other words, if *X* is declared in *T* then *X* of *A* and *X* of *B* are entities as different as if they would get different names.

(9) Let *T1* and *T2* be two different textual blocks so that in each of them an entity with a certain common name *X* is declared. If *Ai* (*i*=1,2) are block instances of *Ti*, then *X* of *A1* is quite different from *X* of *A2*, even in case *T1* is a part of *T2*.

Let *X(i)* (*i=1,…, k*) be a sequences of block instances so that *X(i+1)* is attached to *X(i)* for *i=1,…k–1*. Then the sequence is called ***operation chain*** with ***head*** *X(1)*.

5.3 Detach and call statements

There is a are statement called *detach*, looking like a standard procedure, which any class instance can perform. Let *X* be such a class instance, attached to a block instance *Z*, and let it be the last element of an operation chain *O1*. During performing its life rules, *X* can enter blocks, call procedures and generate objects and so a more or less long operation chain *O2* with head *X* can develop. *O2* develops as a continuation of *O1*. Suppose that during performing its life rules, the last element *Y* of *O2* meets statement *detach* and determines that it is *X* that should perform it. Than the whole operation chain *O2* is really detached from *O1* and exists as an isolated operation chain. The PSC of *Y* is set after the *detach* statement and the computing returns to the PSC of *Z* (see Fig. 3, left). Although *X* might perform its life rules it becomes "dormant".

Let *V* be the head of operating chain *O3* – in general different from *O1*. Suppose sometimes later the last

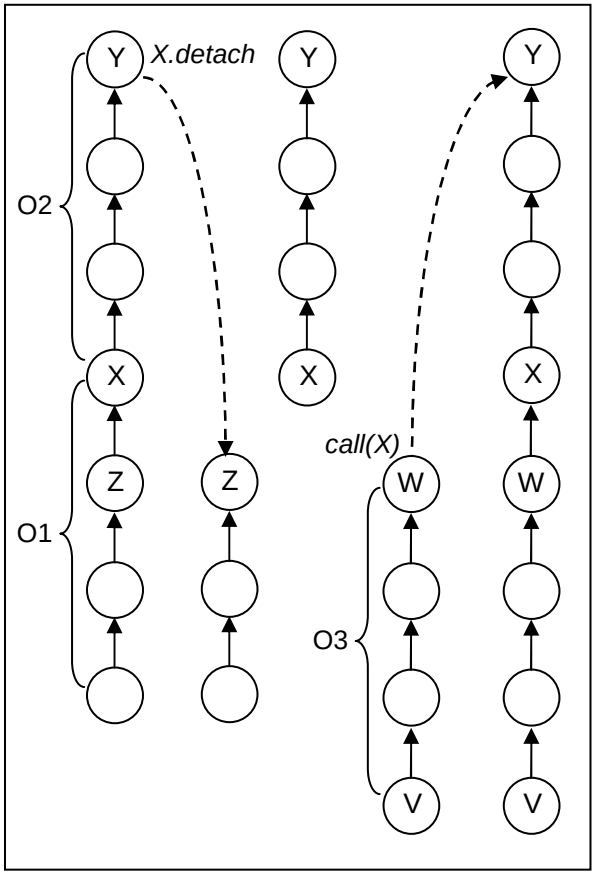

Fig. 3

element *W* of *O3* should perform statement *call(X)*. This statement has also a form of standard procedure with one argument that can point to any class instance. Then *O3* is joined with *O1* so that *X* becomes attached to *W* and the "life" of *V* continues from the PSC of *Y* (see Fig. 3, right).

Statements *detach* and *call* allow to declare a main class called *SIMULAT* that makes SIMULA a simulation language, without neglecting its SOOP tools. *SIMULAT* contains function *time* that gives the value of simulated time, and class *process* that allows every instance of its subclasses to manage its life rules in relation to simulated time. Namely, the following procedures like those in table 1 are for disposal.

5.4 Simulation tools

An integral component of SIMULA is a standard class *SIMULATION*; although it is very suitable for conventional simulation it makes great difficulties for nesting simulation. The obstacles root in the fact that using the mentioned class prohibits models to get names. Therefore serious obstacles arise when one should express that a state of simulation model *A* should be copied as initial state of another model *B*, namely if *B* is nested inside an element of model *A*. Although – after 25 years of working with SIMULA – technique to surmount the obstacles was discovered [18], it demands the users to do rather sophisticated steps. A better implement than *SIMULATION* was discovered (note that that happened not sooner than in 2005). This implement is a main class called *SIMULAT*; it is completely based on the pair of *call/detach* statements, mentioned in the preceding subsection.

In class *SIMULAT*, the switching among the life rules is overviewed by a hidden object *H* that elaborates a "calendar of events", i.e. a list of all elements that demand to work when the simulated time accesses a certain value. *H* scans the calendar and according to the items read at its beginning assigns simulated time and stepwise calls the elements that just came to state of influencing the computation by continuing to perform their life rules. When the life rules of such an element lead to a decision to wait, the element sends the corresponding information to *H* and performs *detach*. The elements that can be governed by *H* in the mentioned manner are called processes and are instances of class called *process* (more perfectly, of subclasses of class *process*). It is to note that the life rules of the simulation model itself behave in the manner as the model would one of the processes (a principle realized already in class *SIMULATION* and in the best discrete-event simulation languages of the history).

The switching among the life rules of the processes in programmed by means of scheduling statements, like:

hold(T) – interrupt until (simulated) time grows up of *T*,

passivate – interrupt until a signal of activation comes,

P.run – activate *P* immediately,

P.run_at(T) – activate *P* at (simulated) time *T*,

P.run_after(Q) – schedule *P* after *Q* performs its scheduled phase,

P.run_before(Q) – schedule *P* before *Q* performs its scheduled phase,

etc.

The scheduling develops dynamically – during the interval between scheduling a process and the performing of its life rules the scheduling can be modifies, e.g. by

cancel(P) – *P* it is passivated even if it has been scheduled for some future (simulated) time,

P.rerun_at(T) – *P* is scheduled for (simulated) time *T*, even in case it was scheduled for some other occasion.

A consequence of the fact that detaching and calling manipulate with the whole operation chain, is that the scheduling statement related to an instance *R* of class *C* can occur in any procedure, subblock and even class nested inside the declaration of *C* (SIMULA allows more but its description would overpass this tutorial).

Let *C* be a subclass of *SIMULAT*. Then *new C* causes the life rules of class *C* to run. They function like to belonging to a certain process called *main*; scheduling statements can be among them and other processes can activate it by using *main.run*… etc. *R:-new C* represents start of a simulation experiment called *R*; it is concluded when the flow of the life rules of *C* leaves *C* (often by accessing *end* of the body of *C*). Nevertheless, the attributes and methods of *R* can be applied.

5.5 Nesting models

Suppose a system *S* is to be modeled and suppose *M* is the main class of the models of *S*. Suppose that in case the models are simulation ones, *M* is formulated as a subclass of *SIMULAT*. Then it is possible to introduce a name *R* of the model by a declaration *ref(M)R*. A creation of the model can be expressed by statement like *R:-new M*. (Note that if SIMULA standard class *SIMULATION* were used in place of *SIMULAT*, such a statement would lead to errors).

Suppose *S* contains elements. Their sorts would be reflected as classes nested in *M*. The elements can be divided into two groups, ***material*** elements (usual in cases of a conventional simulation) and ***thinking*** ones. The last sort of elements can cover e.g. a computer so that (it has certain phases of its "life" during that it processes data obtained at its "environment" in *S* so that the results of computing can be applied in instructions given to the elements of *S*. The computing can be a model ξ of something that could be described by a certain main class μ. Especially, ξ could be a simulation model, possibly influenced by the instantaneous state of *S*. In a special case, ξ could be a model of *S* itself, but a bit different from *X*. One of the differences

can be that ξ does not reflect the thinking element(s) of *S*. Let us concentrate to such a case.

Instead of a computer, a human can represent a thinking element. Human is able to think and to use concepts (i.e. something like classes) when it thinks. Human is able to imagine the future and accordingly to decide. When *S* is automated, such a human can be replaced by a computer, the human's thoughts could be mapped as a modeling phase of the computer and – especially – if the thinking consists in imagining it can be replaced by simulation performed at the same computer.

A similar process comes when such a system *S* governed by person(s) should be simulated. In this case, such a person is represented by an image of a modeling computer, which becomes simulating when reflecting that the person is imagining; note that such a way can be also a test whether (and how) the different abilities of computing technique could replace humans.

Independently of whether the thinking elements are computers or humans, class *M* should contain declarations of the classes of the material elements and a declaration of a class *C* of thinking ones. The life rules of *C* may integrate the instances of *C* into *X*, using all that is available in the declarations of the material classes and e.g. in the life rules of class *M* itself (therefore – among other – the scheduling statements that express a certain existence of the thinking elements during the same time as the material ones, may be of use).

The best way to reflect the phase of modeling consists in a block *B* existing among the life rules of *C*. That block should contain a declaration of class ξ of the models and of a declaration of a name reserved for the models, e.g. like *ref(mu)xi* and the statement like *xi:-new mu*. This statement generates the model and gives it name *xi*. Then the model runs and can detect the properties of the environment of its carrier *Y* (e.g. of the images of the material elements) by means of dot notation where *X* stands before – like *X.car3.place* (i. e. place of element called *car3* and existing in model *X*), or – in a better way – with use of the local reference *this M.car3.place* (see the end of 5.1).

Note that *this mu.time* corresponds to the (simulated) time of the model carried by the given thinking element, while *this M.time* corresponds to the time at which the given thinking element would exist in system *S*.

5.6 Reflective simulation

What was explained in the preceding subsection corresponds in a full way to Fig. 1, where the material elements are represented by circles enumerated by digits. If we attempt to the reflective simulation we should meet something like that presented in Fig. 2. In order to understand the next explication it is suitable to take into account what was expressed in statement (9) of subsection 5.2. In other words, if the main class *M* and μ contain the identical declarations (namely of classes) there is no relation between classes of the same names – they figure as having different names..

That protects SIMULA against an error called ***transplantation***, the substance of which consists in assigning a name reserved inside a main class, to an instance of the class existing in another main class. Also an implicit assigning of a name can be of this sort (e.g. inserting an element of a certain model into a queue belonging to another model) and is automatically rejected in SIMULA programs.

SIMULA allows separate compilation of classes. If such a class is needed in a certain textual block, one puts there its ***external declaration*** (*like external class M*). The compiled program behaves in the same manner like if the detailed declaration of the class would be copied at the place of the external declaration. But SIMULA linker is so organized that when two or more external declarations of the same class *G* occur at the same source text only one object file belonging to *G* is processed and linked, although it behaves like to be particularly linked to each of the mentioned places. That essentially shortens the programs compiled from SIMULA (rather complex models of "intelligent" systems do not need more than 200 KB).

In reflective simulation, *M* and μ may differ only in small details (at least so that μ does not introduce the image of the thinking elements). In such a case, a common superclass *G* can be declared and separately compiled, then called by external declarations at two block levels and at each of them specialized either to *M* or to μ. Note that the specialization to μ can have use of its textual environment that represents specialization of *G* to *M*.

6 Examples of application

A large field for application is logistic, because there are many systems where the transport is planned according to the imagining how different variants could proceed.

So – under support of two projects of European commission [19,20] – the logistics of container yards was simulated, where two tasks of anticipation for ground moving transport tools took place – establishing the shortest paths and testing its security against conflicts and deadlocks in the labyrinth of passages surrounded by containers. The computing of the shortest path was implemented according to the metaphor put down sometimes to Dijkstra and sometimes to Lee, and realized as a simulation model of the metaphor, i.e. of a system of branching, propagating and emulous pulses [5]. The test on the computed path security was implemented by means of the reflective simulation, namely by simulation of what could happen if applying the computed path: in case the simulation discovered that the application of the path passed without obstacles,

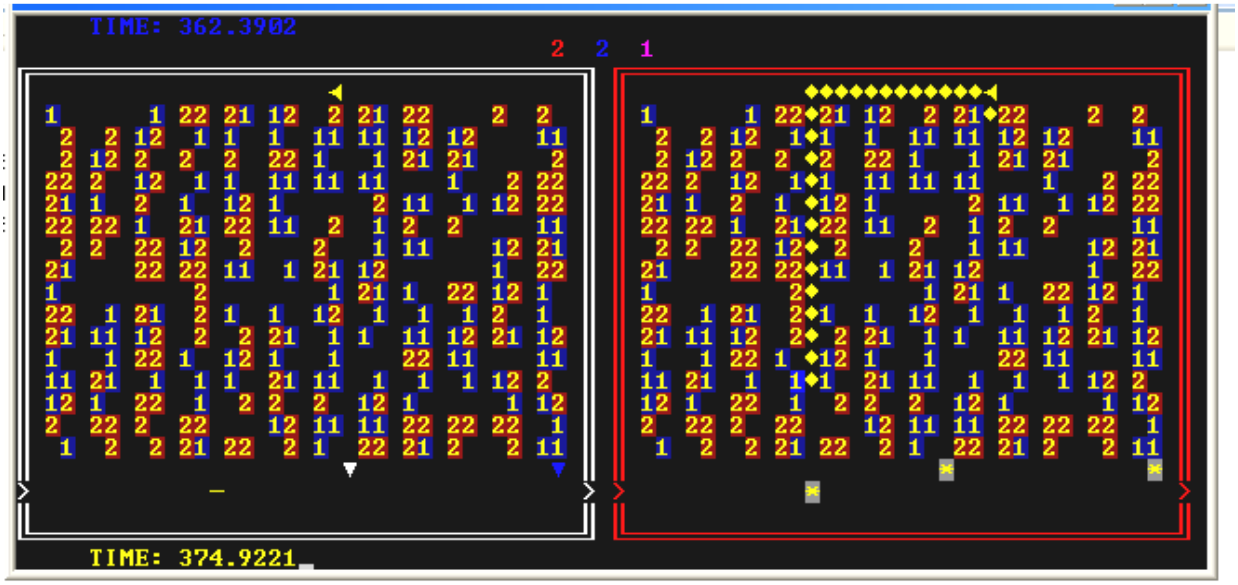

Fig. 4

the path was accepted, in case the simulation discovered that application of the path could lead to a conflict (with another transport tool or with a container meanwhile put at a place belonging to the computed path) the place of the conflict was marked by a fictitious container and a model of the mentioned metaphor was applied; it gave another version of the shortest path, its security was tested by reflective simulation etc., until a secure path was determined. The model is described e.g. in [21,22] and a snapshot of its on-line animation can be seen in Fig. 4 where the simulation of the yard is animated on the left while the tests with the shortest path is on the right, the columns of containers are represented by fields with digits (denoting the number of containers) and the triangles represent ground-moving transport tools.

Another application concerned simulation of circular conveyor tending several working areas and their environment (see Fig. 5). The questions like the following ones arise during the container operation:

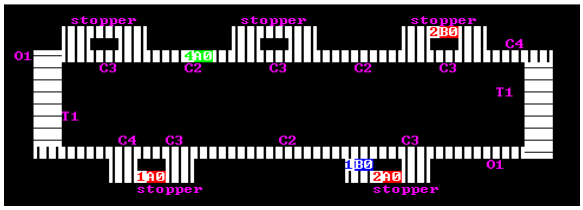

Fig. 5

(i) A failure came to a certain working area; what is better, immediate stop of the system and repairing the error, or going on with a limited number of the working areas?

(ii) If the decision in case of (i) is "to go on", would it be better to change the present technological programs (assigning the work arrays to steps of elaboration), and if so, what change to choose?

(iii) An object comes to the conveyor in order to be processed at some of the working areas; the conveyor is rather occupied and, therefore, if the object immediately enters the conveyor it no working area accepts it, the object performs a full cycle and returns to the place where it entered the conveyor; how long should the object wait out of the conveyor?

(iv) An object should be stepwise processed at several working areas related to the conveyor; having been processed at a certain working area it should come to another one but more working areas can perform the next step of processing; what working area should be chosen for the next step (note that the state of a working area can change during the time when the object is transported to it)?

(v) A similar problem of choice exists for objects that are waiting according to decision mentioned in (iii).

The conveyor was simulated so that the questions (i)-(v) were solved by the help of simulation [23,24].

Other simulation of logistic systems was oriented to public personal transport in a district, with respect to passengers who combine the bus lines according to their imagining of instantaneous possibilities [25]. Relating to it, also demographic development was simulated with respect to consulting centers equipped with simulation models that allow the citizens to anticipate the region development. In the last months, at Ostrava University one approached to a project of reconfigurable information systems design (and therefore of their simulation, too), where four levels exist: one of them introduces fuzzy mathematics, the next one introduces control of simulation studies and related data files, the third one concerns the simulation models and the level of nested simulation models that enable automatically to anticipate the consequences of the intended reconfiguration is at the bottom [26].

ISBN 978-3-901608-32-2

7 References

[1] G. Gordon. A general purpose systems simulation program. In *Proc. 1961 EJCC*, MacMillan, New York, 1961.

[2] D. E. Knuth and J. L. McNealey. SOL – a symbolic language for general purpose systems simulation. *IEEE Trans. Elec. Comp.*, 13:401-410, 1964.

[3] O.-J. Dahl and K. Nygaard. *SIMULA – A Language for Programming and Description of Discrete Event System: Introduction and User's manual*. Norwegian Computing Center, Oslo, 1965 (1th ed), 1967 (5th ed.).

[4] J. W. Backus et al. Report on the Algorithmic Language ALGOL 60. *Numerische Mathematik*, 2:106-136, 1960.

[5] O.-J. Dahl. *Discrete Event Simulation Languages*. Norwegian Computing Center, Oslo, 1968. Reprinted in [6], 349-394.

[6] F. Genuys, editor. *Programming Languages*. Academic Press, London – New York, 1968.

[7] C. A. R. Hoare. *Record Handling*. In [6].

[8] O.-J. Dahl. The Birth of Object Orientation: the Simula Languages. In M. Broy and E. Denert, editors, *Software Pioneers: Contribution to Software Engineering*. Springer, Berlin, 2002. Reprinted in [9].

[9] O. Owe, S. Krogdahl and T. Lyche, editors. *From Object-Orientation to Formal Methods. Essays in Memory of Ole-Johan Dahl*. Lecture Notes in Computer Science, 2635, Springer, Berlin, 2004.

[10] O.-J. Dahl and K. Nygaard. Class and Subclass Declarations. In J. N. Buxton, editor, *Simulation Programming Languages*. Proceedings of the IFIP working conference on simulation programming languages, Oslo, May 1967. North-Holland, Amsterdam 1968.

[11] H. E. Islo. SOOP Corner. *ASU Newsletter*, 22, no 2:22-26, 1994.

[12] E. Kindler. SIMULA and Super-Object-Oriented Programming. In [9].

[13] C. Herring. ModSim: A new object-oriented simulation language. *SCS Multiconference on Object-Oriented Simulation*. The Society for Computer Simulation, San Diego, 1990.

[14] V. M. Glushkov, V. V. Gusev, T. P. Maryanovich and M. A. Sachnyuk: *Programmnye sredstva modelirovaniya nepreryvno-diskretnych sistem* (Programming tools for modeling continuous-discrete systems – in Russian). Naukova Dumka, Kiev, 1975.

[15] O.-J. Dahl, B. Myhrhaug and K. Nygaard. *Common Base Language*. Norsk Regnesentralen, Oslo, 1st edition 1968, 2nd edition 1972, 3rd edition 1982, 4th edition 1984.

[16] O. Madsen, B. Møller-Pedersen and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Harlow – Reading – Menlo Park, 1993

[17] *SIMULA Standard*. SIMULA a.s., Oslo, 1989

[18] E. Kindler. Chance for Simula. In: *Proceedings of the 25th Conference of the ASU – System Modelling Using Object-Oriented Simulation and Analysis*, August 1999 ASU, Kisten, Sweden

[19] E.Blümel et al. *Managing and Controlling Growing Harbour Terminals*. The Society for Computer Simulation International, San Diego, Erlangen, Ghent, Budapest, 1997

[20] E. Blümel and L. Novitsky, editors. *Simulation and Information Systems Design: Applications in Latvian Ports*. JUMI Ltd., Riga, Latvia, 2000

[21] E. Kindler. Nesting Simulation of a Container Terminal Operating With its own Simulation Model. *JORBEL (Belgian Journal of Operations Research, Statistics and Computer Sciences)*, 40: 169-181, 2000.

[22] E. Kindler. Nested Simulation Models Inside Simulation of Container Terminal. In A. G. Bruzzone and E. J. K. Kerkhoffs, editors, *Simulation in Industry*, 8th European Simulation Symposium (ESS 96), Genova, September 1996, Society for Computer Simulation International, San Diego, Volume I.

[23] P. Berruet, T. Coudert and E. Kindler. Conveyors With Rollers as Anticipatory Systems: Their Simulation Models. In D. M. Dubois, editor, *Computing Anticipatory Systems CASYS 2003*. Sixth International Conference, Liege, Belgium, August 2003. American Institute of Physics, Melville, New York, 2004.

[24] E. Kindler, T. Coudert and P. Berruet. Component-Based Simulation for a Reconfiguration Study of Transitic Systems, *SIMULATION*, 80:153-163, 2004.

[25] P. Bulava. *Transport system in Havirov*. In *Proceedings of 28th ASU Conference*, Brno, September 26 – October 1, 2002. FIT, University of Technology, Brno.

[26] E. Kindler, C. Klimeš and I. Křivý, Simulation Study With Deep Block Structuring. In J. Štefan, editor, *MOSIS'07*, Proceedings of 41th Spring International Conference Modelling and Simulation of Systems, Rožnov Pod Radhoštěm, April 2007. MARQ,.Ostrava.

