Software for Simulation of Anticipatory Production Systems

Ivan Krivy*, Eugene Kindler†, Alain Tanguy‡

* Department of Computer Science, Faculty of Sciences,
Ostrava University, CZ – 701 03 Ostrava 1, 30. dubna 22, Czech Republic
Fax: +420-69-6120 478; e-mail: krivy@osu.cz

† Department of Mathematics, Faculty of Sciences,
Ostrava University, CZ – 701 03 Ostrava 1, 30. dubna 22, Czech Republic
Fax: +420-2-2191-4323; e-mail: kindler@ksi.ms.mff.cuni.cz

‡ LIMOS CNRS FRE 2239, Université de Clermont-Ferrand II
Complexe scientifique des Cézeaux, F – 63177 Aubière Cedex, France
e-mail: tanguy@isima.fr

Abstract
Two modes of weak anticipation are applied to production systems. When such a system is being designed both the modes interact. If simulation support is applied, the interaction causes that a system with simulating elements is simulated, i.e. one meets nesting simulation models. In other words, when we anticipate the system existence we should take into account the fact that anticipation will exist in the system during its operation stage. Although the essential problems related to the nesting of simulation models have been solved, some obstacles remain. They are rather of a psychological character and they can exist during the design of any system. For the branch of production systems, they are diminished by a simulation system REFLECTIVE QNOP, the principles of which are described in the present paper.
Keywords: Anticipation of anticipating systems, Simulation nesting, Object-oriented programming, Anticipation in technology, QNOP.

1 Development of technology - Decomposition of Abilities

1.1 Things as Elements of Systems

The systems are composed of ***elements***. An element can be present in the system during its whole existence; such elements are called ***permanent elements*** or ***stations***. The other elements are called ***transactions***; they can enter a system after the start of its existence and leave it before the conclusion of its existence. It is natural to view the elements as images of the physically identifiable things that have their portions of "materia prima", their volume, their instantaneous position etc.

Although many persons who have use of system viewing in their professions do not explicitly express what we have just stated and although the above mentioned "Aristotelian" aspects of the elements of the systems often disappear from the conscious abstraction of such persons, many technical, social and biomedical branches are based on the described conception and only in special boundary cases the elements of systems are viewed as images of entities that are not material things. Maybe physics made the greatest step out from that manner, but the engineers can complain that the more distant is that abstraction from the Aristotelian point of view the lesser relation between the results and possible applications of them exists.

1.2 Things as Configurations of Abilities

We can observe that the physically identifiable things have ***abilities***. The things are able to do something. For instance, a laboratory desk has an ability *A1* to sustain some things on its surface, an ability *A2* to contain some things in its inner and an ability *A3* to occupy a certain portion of space. An ironing plate has abilities *A1* and *A3* and fails to have *A2*, but has another ability *A4* to serve for ironing. A bookcase has abilities *A2* and *A3* but has neither *A1* nor *A4*. (Note that the system viewing is applied in professional activities – for example by designers of flat interiors or by heads of laboratories – and such variants lie to iron at a laboratory desk or to perform chemical experiments at an ironing plate are out of their abstraction, contrary to the fact that they are possible.)

Other examples of abilities are those of humans – writing, speaking, walking, eating, beating, thinking etc. Humans have no abilities like flying or producing honey, though certain animals have them. Note the animals have some abilities similar to those of humans – eating or walking. Since the old times humans have used tools to get more and better abilities. In a certain way they have formed structures $S = <P,T>$ (where P was a person and T a tool) that operated in the same environment and context as P should have to operate, but with abilities of T, which were better than those of P or which P did not possess. A couple man-horse or man-cudgel illustrates this matter. Instead of a single tool, a set of several tools can be present (man-horse1-horse2-carriage, man-sword-shield-armor).

1.3 Abilities of Technology Products

Yesterday the tools were things existing in the nature – non-living materials, plants and animals – and their configurations (for instance caves and lakes). The technology progress can be characterized by their replacement by artificial objects. Nevertheless those objects have an essential property – their designers tried to produce them as optimal structures of such things carrying the demanded abilities.

Evidently, an instrument S composed of things R_i ($i = 1, \ldots, n$), R_i having abilities $A_{i,j}$ ($j = 1, \ldots, k_i$) is often constructed for offering a rather small subset of these abilities. Moreover, some abilities are applied only in some situation but must always be managed and paid. An example is a truck, viewed as a pair of abilities to move (*H1*) and to carry material (*H2*). If it is loaded or unloaded its ability to move exists but one has no use of it.

In such a case the technology tends to produce a greater number of "simple" tools, i.e. with small numbers of abilities and offers them to the users, expecting them to combine the simple tools in configurations suitable to certain tasks and situations. The mentioned truck example can serve as an illustration of that development: its ability to carry material was transferred to container and its ability to move was transferred to container carrier. If a container waits to be loaded or unloaded the carrier can be used to move and carry other containers.

The decomposition can be characterized as an essential aspect of the modern development of technology. It can be met so far from logistics as in programming: as an example we can present a decomposition of *block*: it was originally considered as having two abilities – that to have "local" entities (variables and subroutines accessible only inside it) and that of being "nested" (included) in another block (Backus et al., 1960; Naur, 1963); but then the abilities were separated so that the first one was connected with *process* (Buxton, 1968) and the second one with *compound statement*.

2 Abilities and Object-Oriented Programming

2.1 About Object-Oriented Programming

Object-oriented programming (further OOP) of computers is based on knowledge representation and their use in producing program products. Its essential properties are as follows:

2.1.1 general concepts are represented as ***classes***,

2.1.2 a class has its name, its ***attributes*** and its ***methods***, and – in some OOP tools like Simula (Dahl, Myhrhaug, and Nygaard, 1968; Simula Standard, 1989), Beta (Madsen, Møller-Pedersen, and Nygaard, 1993), Java (Eckel, 2000) and ModSim (Herring, 1990) – its ***life rules***; the methods and the life rules have a form of an algorithm (a subroutine);

2.1.3 individuals that can influence the computing process (i.e. models of things that realize the concepts) are called ***objects*** and are generated as ***instances*** of classes,

2.1.4 any instance of a class has its own attributes as they have been introduced in the class; they reflect the properties of the instance;

2.1.5 any instance of a class is able to perform any method introduced for the class, in case it is demanded by an object to do it; the demand is called ***message***. It contains the object that is demanded (called ***addressee***), the name of the method that the addressee should perform (called ***selector***) and sometimes parameters;

2.1.6 in case a class contains life rules they are performed by an instance of the class immediately after it is generated;

2.1.7 the instructions to generate a new instance of a class can be present in the formulations of the methods and of the life rules; according to the instructions, new instances can be generated by any object when it performs such a method or life rules;

2.1.8 if life rules are admitted by an OOP tool, they can obtain messages; thus an object can turn to itself to do a method (in that case the object uses the method as its own subroutine) or it can turn to another object;

2.1.9 if life rules are admitted by an OOP tool, they can interrupt their performing in the computing process and transfer the computer run to the life rules of other objects – so a parallel existence of things can be modeled at a monoprocessor computer; it has an essential importance for computer simulation; the instruction for interrupting the performing of the life rules is called ***sequencing statement***; the set of objects that can mutually transfer the computing control one to the other by using sequencing statements is called ***quasi-parallel system***;

2.1.10 the sequencing statements can be present in the methods too; suppose an object *J* performs its life rules, sends a message to an object *K* to perform a method *M*; and suppose that *K*, performing *M*, meets a sequencing statement; in such a case, an object (named *K*) causes the interruption of the "active phase" of another object (named *J*);

2.1.11 a class *C* can be used as a ***prefix*** for another class *D*; that means the relation "*D* is *C*" (e.g. "dog is animal", "lathe is machine", "car is a transport tool") and is interpreted so that the instances of *D* "inherit" all attributes, methods and possibly life rules introduced for *C*; *D* is called ***subclass*** of *C*, or ***specialization*** of *C*;

2.1.12 it is possible to add any attribute and method when a subclass *D* of a class *C* is introduced; the instances of *D* have these attributes (together with those introduced for *C*) and are able to perform any method introduced for class *D* or *C*; nevertheless:

2.1.13 in a subclass, it is possible to "redeclare" the meaning of a method introduced in its prefix; such a method is called ***virtual***;

2.1.14 a subclass *D* of class *C* can be used as a prefix for introducing another class *E*; so we can introduce "trees" of classes – similarly as in Liné's classification of living organisms; if *E* is a subclass of *D*, *D* is a subclass of *C* etc., then the sequence {*E*, *D*, *C*, ...} is called ***prefix sequence*** of *E*; the meaning of a virtual method can be redeclared several times in such a prefix sequence; the consequence is that when an object sends a message to perform a virtual method, the addressee self decides what it should do: it looks for the nearest (re)declaration of the method in its prefix sequence.

The possibility mentioned in 2.1.11 enables to compose very complex knowledge systems and to apply them to model very complex situations. It can be popularly said that the specializations can be cumulated so that from such a moment only the computer knows what attributes and methods belong to particular objects. The possibility mentioned in 2.1.10 enables to simulate very complex systems but carries problems discussed in section 4.

2.2. Programming Languages With Three Orientations

The principles of OOP started by the proposal presented in (Dahl and Nygaard, 1968) and by the implementation of it in the language of Simula (Dahl, Myhrhaug, and Nygaard, 1968). The name object-oriented programming did not exist. It was introduced in the 80-ies, when OOP was accepted by the world programming community and used as the best method of making software products; so it has been used and preferred until nowadays. The reasons of that situation are that – beside the possibility to model very complex systems – the OOP enables to implement specialized programming languages tailored to application branches, i.e. to certain classes of programming tasks; such a language can be made so that it uses a similar terminology as the specialists of the given branch use outside computer programming.

The programming tools (languages) inspired by Simula - among them the most popular SmallTalk, C++ and Eiffel (Meyer, 1989) and several tens of languages derived from them and from LISP - did not allow their users to include life rules and quasi-parallel systems. Therefore the life rules and quasi-parallel systems are in general not accepted as necessary aspects of the OOP: they relate to the concept of ***agent***, introduced into the programming practice at the end of the 90-ies as a certain component that has its own existence during that it can "watch its environment (containing other agents), evaluate it, decide about its own reactions to the instantaneous situation and realize the decision". So we can say that Simula, Beta, Java and ModSim are not only object-oriented languages but ***agent-oriented*** ones, too.

The concept of block, mentioned at the end of section 2.1, has been already known since the end of the 50-ies and perfectly introduced into the programming language ALGOL 60 designed by an international commission (Backus et al., 1960). In 1966 it was accepted into the first design of Simula and did not abandon it. Simula is therefore also a ***block-oriented*** language. At the beginning of the 70-ies, the concept of block was criticized by the theoreticians of the structured programming (programming technology that seemed to be good in that time). Later appeared that the blocks make no obstacle for the structured programming. Moreover, the structured programming was abandoned because of the qualities of the OOP (Meyer, 1989). Nevertheless the condemnation of the blocks entered so deeply into the means of the designers of programming languages that the concept of block offered to the users of new programming languages returned only in Beta and Java (ModSim is not block-oriented).

Let the programming languages that are simultaneously object-oriented, agent-oriented and block-oriented be called ***languages with three orientations***, shortly ***o3-languages***. Simula, Beta and Java belong to them. The three orientations imply that in the life rules of a class C a block B can occur in which classes are introduced as its local entities. When an instance K of C influences the computation according to the life rules of C and enters block B it presents a model of a being that "thinks", using the abstract concepts represented by the classes local in B. This "thinking" phase of the K's life exists during the computing points into B. As soon as the computing leaves B, K forgets all what could be expressed by the concepts represented by the classes local in B. The thinking phase of K can also be interpreted as a modeling phase – using the instances of the classes local in B, K handles a certain model which exists and which can be developed when the computing does not leave B. So K can be viewed as a human having a computer on which he handles models and according to them makes decisions about its future behavior. Similarly, K can reflect a machine facilitated with a computer that in certain phases uses models and according to them controls the machine. Shortly said, when *K* enters *B* it becomes a modeling agent.

2.3 The First Experiences With Object-Oriented Programming and Abilities

One can meet rather recent definitions of agents (Schmidt, 2000; Savall et al., 2001). They surprise so that they do not distinguish between computer software components and something that exists independently of them (e.g. agents that are computers in a computer network) or even independently of computers (e.g. in a factory, machines and workers are viewed as agents). From one point of view it reflects the contemporary programming practice in which one tries to describe the modeled reality and hopes the description to be automatically translated into the corresponding computer model. But from another point of view, one can see the definitions as conserving the abstraction that a system element (in our case an agent) exists as a physically identifiable thing (see the beginning of section 1.1). Sometimes it is very explicitly expressed by emphasizing the mobility of agents (Schmidt, 2000).

Although the OOP offers tools to express the properties of such viewed things it is not limited to it. By means of the same tools it is possible to express the properties of abilities and their configurations. The first attempt oriented to a commercial application was met in relation to a simulation of material flow systems in agricultural farms (Kindler, 1983).

At the beginning we met systems with permanent elements like stores, fields and roads and with transaction like trucks, sowing machines, combine harvesters and belt conveyors. We began to simulate such systems, using Simula that appeared an OOP tool suitable for simulation. But after a lot of simulation studies we met systems using more modern tools, like containers and trucks with special unloading facilities (dumping facility, manure dispenser). We discovered that the concepts of the new tools had many common properties with similar concepts we used before, but sometimes the concepts were surprisingly limited. As an example we can present a container as a concept similar to that of a store, but without its connections to access roads, or a container carrier as a truck but without its own ability to carry material (and with its special ability to carry a container). Nevertheless, sometimes the properties were "inherited" from more than one of the original concepts, like in the trucks with dumping facility or with manure dispenser: to the properties of a conventional truck an unloading ability entered and was similar to that of belt conveyor for the damping facility (the time of unloading depends on its rate and on the size of the material placed in the truck) and to that of sowing machine in case of the dumping facility (the time of the unloading is either a constant or a random value, anyway independent of the size of the unloading material, the rate is meaningless).

We discovered what is expressed at the end of section 1.2: the traditional tools (viewed as physically identifiable things) are configurations of abilities and the modern tools can be seen as new configurations of the same (or of similar) abilities. As an example, we can present the table 1 similar to that published in (Kindler, Chochol, and Prokop, 1983); it demonstrates the use of some abilities to compose structures corresponding to physically identifiable things. The heading in the bold letters presents four names of abilities. *Moving* is the ability to change place, *Serving* is the ability to load or unload a container or a truck, *Box* is the ability to contain material and *Waiting* is the ability to wait in queues. Note that although the connection of words *waiting* and *ability* may sound a bit ironical, the waiting in queues is to be considered in that way because the data of waiting in queues are often important information on the use of resources and only computer simulation can give them.

Table 1: structures of abilities – an example

Abilities / **Structures**	**Moving** Trace, Place, MakeStep,...	**Serving** Portion, Tserving, Regeneration,...	**Box** Volume, Capacity, Empty, Full,...	**Queuing** Tstart, Tspent, Enter, Leave,...
Conventional truck	1	~	1	2
Special truck	1	1	1	1
Container	~	~	1	3
Container carrier	1	~	~	1
Storage	~	~	1	~
Belt conveyor	~	1	~	1
Combine harvester	~	1	~	1
Sowing machine	~	1	~	1

Usually a conventional truck has two of such abilities because it can wait for loading and for unloading and the purpose of simulation demand to get separate data for each of the cases. A usual container has even three abilities to wait in a queue: besides the waiting for loading and unloading it can wait to be carried. The usual combine harvesters wait only for being unloaded, the usual sowing machines wait only for being loaded and the belt conveyors are commonly viewed as waiting for any work that is no further structured (but it is clear that the "unusual" belt conveyors can be very simply considered by adding another ability to wait).

An opinion can arise that the abilities can be represented as methods. An experience tells that such a case is rather rare. The abilities are composed of attributes and methods and sometimes have life rules. It can be observed at the heading of Table 1. For example, the ability of moving has several attributes, among them a list *Trace* of places, along which a truck has to move, and a certain "reader" of the trace, called *Place*, because it really identifies where an element with the given moving ability is being. The ability *Box* has two attributes – *Volume* of the material actually placed in the element having that ability, and *Capacity* (maximum possible value of *Volume*), and two virtual methods *Full* and *Empty* telling whether the element is full/empty. Note that in a commercial case *Full* can mean "almost full" (to wait for loading into the remaining empty space would not be economical), and – similarly – *Empty* can mean "almost empty" and both the fuzzy expressions must be algorithmically formulated. The ability to wait in a queue has two attributes *Tspent* (cumulated time spent in the queues) and *Tstart* (for storing the moment of entering a queue) and two methods *Enter* and *Leave* that correspond to entering a queue and leaving it, but they must handle the mentioned attributes in order to compute the cumulated waiting time.

From the other point of view, although the abilities don't need to be identical with physically identifiable things their configurations must be so and therefore some abilities are to be considered as carrying some "materia prima". Let us call them ***materially supported abilities***. They are meaningfully connected with the other abilities, because they must form elements that we view as physically identifiable things. Therefore there must be ***communication abilities*** that realize that. Other communication abilities realize a communication that we see among the physically identifiable things. In some methods of the communication abilities, certain laws must be respected, which correspond to general principles and laws of logic, mathematics and natural sciences. Let us call them (rather metaphorically) ***logic rules***.

In the computer simulation studies oriented to material flow systems, which were mentioned in the present section, it appeared that a differentiation among the materially supported abilities and the communication ones is not necessary. Nevertheless in the recent times the differentiation appeared is very suitable, as it will be presented in the next section.

3 Simulation and General Viewing to Production/Logistic Systems

3.1 A Systematic Handling With Abilities

What was described in 2.3 can serve as an illustration of the "prehistory" of the ability analysis and as simple and transparent example that could introduce into the studied matter. The old attempts were not systematic and rather chaotic; one of the reasons of that situation was that things of different character have very different abilities – for instance the abilities of a human considered as an object of a medical care are in general very different from those of a car with a dumping facility or from those of the economic development of a continent.

The systematic research is possible when it is limited to a certain branch that is sufficiently reduced to systems composed of elements with similar abilities; naturally there is another pressure to the research, namely from the commercial application: it demands the branch to be as large as possible in order the results would be applicable in a rather great cases.

The best fruits in that direction were accessed by analysis of production/logistic systems (further PLS): the results were presented e.g. in (Gourgand and Kellert, 1991; Gourgand, Grangeon, and Norre, 2001; Tchernev, 1997). They show that it is convenient and effective to view any PLS as a system composed of three subsystems:

the physical one, the logical one and the decision one. Shortly said, the physical subsystem contains the materially supported abilities, the logical subsystem contains the communication abilities liable to the logic rules, and the decision subsystem contains abilities for expressing the intended decisions. If a system *S* is being designed the abilities of the decision subsystems of the variants of *S* can be more or less independently introduced in the variants (e.g. in order to discover the optimal variant). In the analysis (and in the simulation, too) of the variants, the separation of decision subsystem from the logical one allows to separate the material base of *S*, which is often supposed not to be modified, from the control of it. (Note that the optimal control is often the main task of the analysis and simulation of a PLS.)

3.2 Simulation System QNOP

The viewing on the PLS mentioned in 3.1 was applied in several software products (programming environments for analysis and modeling of PLS). A QNOP simulation system is among them (Tanguy, 1993). It was implemented as a set of classes written in Simula. Simula itself allows putting classes into another class (called ***main class***) and therefore the mentioned set was put into a main class called QNOP.

Although the classes contained in class QNOP may seem very rich and complex, after a certain analysis one can discover that the global structure of them is rather simple: they perform certain cycles composed of couples of one complex activity and the following waiting; it corresponds to the discrete event interpretation of the real world process (Kindler, 2001) and to the fact that all QNOP classes represent permanent elements. The elaborated objects are not permanent elements but they have no life rules. The permanent elements of PLS are investments (machines, conveyors, stores,

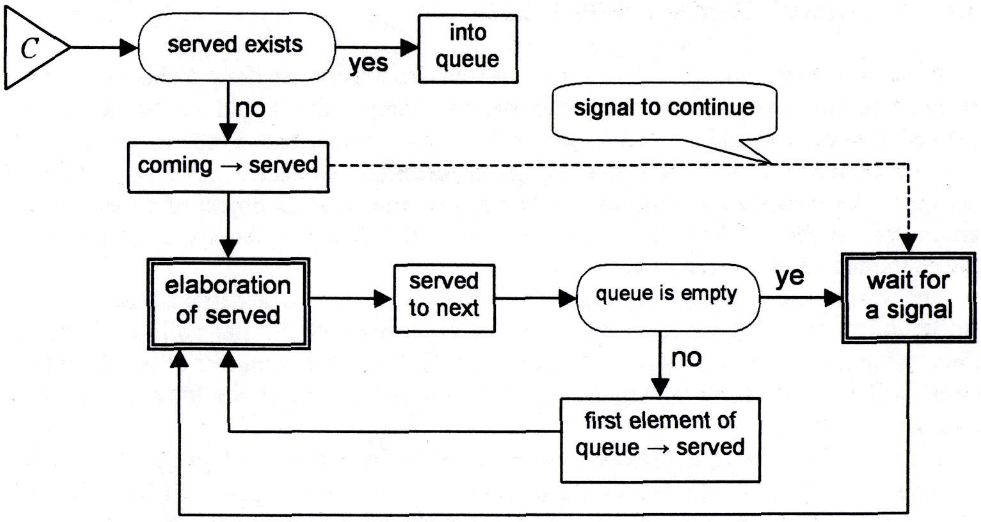

Fig. 1: an example – simple machine

signalizing tools, etc.) and their abilities. One can accept without any professional analysis the idea that such an element has some working cycle. Let us present a short example in Fig. 1.

A simple machine *M* works in the following way. It handles an object called *served*. Its elaboration takes some time. After *served* has been elaborated it is sent to *next*, i.e. to an investment (machine, store, transport etc.) performing its further elaboration and the first object waiting in the queue is taken to be elaborated – therefore it becomes *served* and the cycle is repeated. When the queue is empty *M* must wait. *C* represents the event when an object is sent to *M* from another investment. In case *M* waits (i.e. if *served* does not exist) it is activated to continue, if *M* works (i.e. if *served* exists) the coming object must wait in the queue. The life rules of *M* can be described as an algorithm. We can see that it contains only two waiting phases; they are marked by double-lined boundary in Fig. 1.

Note that the simple global structure of the life rules cycles has an important role in the solving of the problem, which the following section is dedicated to.

4 Psychological Problems of Reflective Simulation

4.1 Reflective Simulation

Computer simulation is a widely used tool for weak anticipation of the behavior of systems whose states change in time that is considered as in Newtonian physics. By use of simulation, it is possible to anticipate on a system that is designed (let us speak about anticipation of the first type, shortly ***anticipation-1***) or on a behavior of a system that already exist (let us speak about anticipation of the second type, shortly ***anticipation-2***).

Simulation applied in anticipation-1 could be called ***simulation-1***; it tells the designers what the designed system will do; simulation-1 is able to anticipate about the behavior of different variants of the designed system (differing e.g. by some numerical parameters, or by machine number and/or configurations, or by control rules and/or algorithms, etc.). The anticipation can concern a design of a totally new thing or a thing developed from an existing thing by some essential intervention (adding/removing/ changing machines or other investments, adding a new interface, changing the structure of transport and/or the capacities of stores, changing control rules etc. The anticipatory system is the designer team possessing and using the simulation model.

Simulation applied in anticipation-2 could be called ***simulation-2***; it is applicable when we would like to anticipate the consequences of a certain decision on the behavior of a system that exists. It can be used e.g. for anticipating the results of a proposed medical therapy or for anticipating consequences of decisions generated in production/ logistic systems by non-simulation methods of operation research, like shortest path computing or heuristical algorithms for production planning. Another example can be to anticipate financial flows after a certain decision of financial resource handling.

Nowadays, almost every system *S* designed by humans contains computers that help it to run. Often the computers use models for simulation-2. Let us call them ***internal models*** of *S*. When such a system is designed often simulation-1 is applied. It uses a simulation model that exists independently of the simulated system; let us call it ***external model*** of *S*.

In (Kindler, 2000), it was shown that in such a case the internal simulation models should be nested inside the external one (otherwise either the external model gives bad data for anticipating on *S*, or the simulation-2 is not necessary for the anticipation-2 in *S* – but the experience shows that the second variant is not realistic). As the external model reflects the same thing as the internal one (and often both the models are described with the same formal language) we speak about ***reflective simulation***.

4.2 Psychological Problems of Reflective Simulation - Introduction

In the same paper (Kindler, 2000) and in (Kindler, 2001) the logical and programming problems of nesting models and reflective simulation have been presented as well as their solutions. Nevertheless there is another problem, and its nature is more psychological than technical. Let us introduce it.

In 2.1.9 it was mentioned that quasi-parallel systems enable the modeling of a parallel world at a monoprocessor computer. It is the only suitable way for that purpose. In section 3.2 it was shown that in simulation programming the scheduling statements of quasi-parallel systems have a form of interruption that could be expressed as statements "interrupt the performing of the life rules until the simulated time accesses a certain value" or "interrupt the performing of the life rules until a signal comes telling to go on". The first variant is ***called imperative (scheduling) statement*** and the other variant is called ***interrogative (scheduling) statement*** (both the variants are illustrated in Fig. 1).

Both the scheduling statements cause some complications when the internal model should arise and run inside the external one. Let us outline them by means of an illustration in Fig. 2.

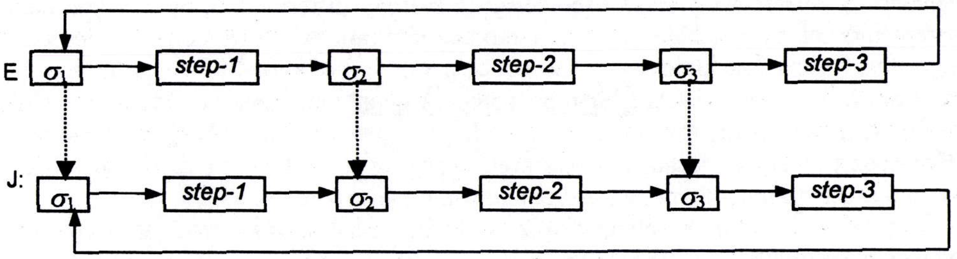

Fig. 2: correspondence between the states of the mutually nested models

Suppose *E* is a life cycle of a component *C* (e.g. a machine or a conveyor etc.). Suppose σ_i are scheduling statements and *step-i* are phases of the life rules without interruption by a scheduling statement. Suppose a signal arises in the external model to generate and use an internal model *M*. In that moment *C* obviously must be in some interruption phase, i.e. performing the scheduling statement σ_k. When *M* is generated it must have an image D of C. The image has similar life rules; in Fig. 2 they are represented by J. When the internal model starts to work D must be prepared to start from the phase *step-k*. That correspondence is represented by dotted lines in Fig. 2.

Suppose C started from phase *step-1* when the external model starts. It does not imply D to start from *step-1*: M starts from σ_2. Moreover, in one internal model the value of k can differ from that existing in an internal model generated later. The "starting index" k changes dynamically for different internal models, namely according to the state of C in the external model.

The last phrase offers a solution: every component C of the external model, which is liable to some life rules, must carry its proper attribute – called e.g. *state* – and before any scheduling statement the life rules must cause to assign a certain value for it. Every

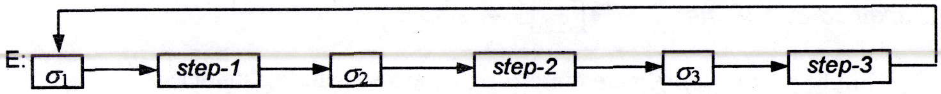

Fig. 3: assignments for states in external model

scheduling statement must own its proper value of the *state*. (In our example the value of *state* can be for instance the index i of the following scheduling statement.) It is outlined in Fig. 3, where σ_i* represents unordered pair of computing activities:

<assign i for state, perform scheduling statement σ_i>

The life rules (e.g. J) occurring in the internal model are a bit different from those (e.g. E) occurring in the external model. They do not need to assign for *state* but they have a jump to *step-k* according to the value of *step* of the corresponding element C of the external model. The jump does not belong to the life cycle (see Fig. 4).

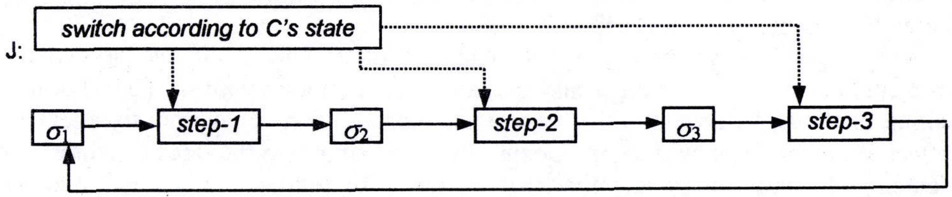

Fig. 4: application of the state value

4.3 Psychological Problems of Reflective Simulation – Essence

Note that the life rules presented as examples at the preceding three figures are very simple. A person who implements a reflective simulation model tends to describe life rules with tens of phases separated by tens of scheduling statements. Moreover, he tends to have the greatest use of the offered programming tools and thus he can apply calling methods in the life rules and simultaneously he can include scheduling statement into them. Let us present an example in Fig. 5.

An element *N* is able to perform a method *F*. Among the steps of *F* there is sending a message to an element *Q*, demanding it to perform a method *G*. A part of this method is to apply a scheduling statement to the element that functions as "current", i.e. that is performing its life rules. Suppose an element *K* sends a message to an element *N*, demanding it to perform a method *F* (see an illustration in the first line of Fig. 5). Then

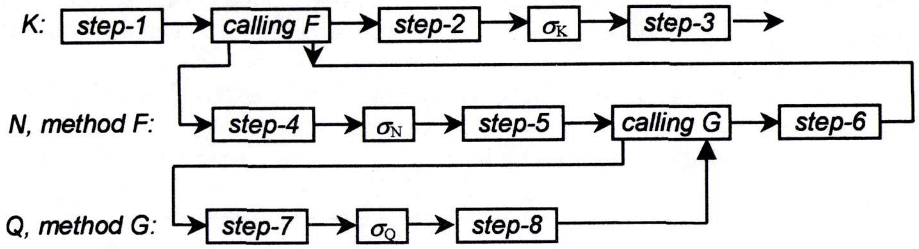

Fig.5: a more complicated interruption by a scheduling statement

N performs *F* (see the second line of Fig. 5) but it meets the calling of the method *G*, performed by *Q* (see the third line of Fig. 5). There the computing meets the scheduling statement σ_Q and it causes an interruption of performing the life rules of *K* (not of *Q*, because it could have been interrupted before and at another place than σ_Q inside *G*).

Assume *K* to be interrupted as it was just explained, and a necessity comes to generate and let run an internal model. We see that the image of *K* in that model should start its run from *step-8* (i.e. from the inside of method *G*), then to continue by returning from *G* into method *F*, there to continue by *step-6* and finally to return to its own life rules and continue by *step-2*. Therefore *K* should have not only one value of its state but a stack (last-in-first-out list) of states, because it must make evidence in all its hierarchical components it is using.

And in good programming of the manipulation of this stack, the psychological problem lies. Every programmer may prepare external models with such a use of nested calling methods, he does it with pleasure and then he must be enormously attentive to reflect all the callings by the appropriate filling and emptying the stack of almost every element. He must be so attentive that he does not follow it and repeats to make programming errors: they are hardly identifiable (and so hardly reparable, too).

4.4 Solution – Enlarging QNOP

Theoretically, there are several ways to solve the described problem. One idea could be to recommend avoiding the use of the object-oriented languages. But almost all programmers and – especially the advanced ones – use them. (Note that people who would like to implement reflective simulation models are advanced programmers.) Another idea could recommend a rule prohibiting the calling of methods, even in case the programming system admits it. The programmers would not be satisfied because they feel it is illogical to prohibit the use of tools that are at disposal. Note that similar problems are also caused by subblocks (Kindler, 2000).

The solution of this problem is to offer the user something, which would be so rich that he would not desire to use such sophisticated programming steps like the one mentioned above. In this situation we appreciated QNOP as an excellent tool for a start to develop a programming environment applicable at least in the array of reflective simulation of production/logistic systems. QNOP offers to view such systems in a rather natural manner (see section 3.2; for the conventional – i.e. non-reflective – simulation it was applied in a lot of commercial cases) and instead of calling methods offers a cooperation of processes with rather simple life rules. Moreover, QNOP is implemented using Simula, i.e. on the base of a good o3-language, and therefore it can be well enriched by tools for reflective simulation.

Thus an idea to develop a REFLECTIVE QNOP arose. After elaborating the first roots (Kindler, Krivy, and Tanguy, 2001a; Kindler, Krivy, and Tanguy, 2001b) it appeared that Simula and QNOP admit to implement a programming environment that could enable an automatic generating of the internal model from the external one.

5 Conclusion

The authors are interested in the simulation of the production and/or logistic systems that are anticipatory in a weak sense, using their own simulation models during their existence. The simulation of such anticipatory systems introduces reflective simulation and reflective anticipation. This subject comes not from a theory of modeling or from software engineering but from industrial applications of computing technique and may introduce new tasks into the science on anticipatory systems. Nevertheless, it makes problems in the implementation of computer models. The authors are engaged to prepare software that enables the wide community of engineers to implement and apply the reflective simulation models of production/logistic systems.

Acknowledgments: this research was performed under the Barrande programme (bilateral cooperation between French and Czech universities) and also supported by the institutional research scheme of the University of Ostrava.

References

Backus, J. W. et al. (1960) Report on the algorithmic language ALGOL 60. Numerische Mathematik, **2**: pp. 106-136.

Buxton, J. N. (1968) Simulation Programming Languages – Proceedings of the IFIP Working Conference on Simulation Programming Languages edited by J. N. Buxton. North-Holland Publishing Company, Amsterdam, 464 pp.

Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968) SIMULA Common Base Language (1st ed.). Norsk Regnesentralen, Oslo. 1972 (2nd ed.), 1982 (3rd ed.), 1984 (4th ed.).

Dahl, O.-J. and Nygaard, K. (1968) Class and Subclass Declarations. In (Buxton, 1968), pp. 158-171.

Eckel, B. (2000) Thinking in Java, Revision 10a, electronic version. http://www.Eckel Objects.com.

Gourgand, M., Grangeon, N., and Norre, S. (2001) Modèle Générique Orienté Objet du Flow-Shop Hybride Hiérarchisé. MOSIM'01 – Actes de la Troisième Conférence Francophone de Modélisation et SIMulation edited by A. Dolgui and F. Vernadat, Published by Society for Computer Simulation International, San Diego, Erlangen, Ghent, Delft, pp. 583-590.

Gourgand, M. and Kellert, P. (1991) Conception d'un environnement de modélisation des systèmes de production. 3ème Congrès International de Génie Industriel, Tours, France, pp. 191-203.

Herring, C. (1990) ModSim: A new Object-Oriented Simulation Language. SCS Multiconference on Object-Oriented Simulation. Published by The Society for Computer Simulation, San Diego.

Kindler, E., et al. (1983) Algorithmization of material flow systems abilities (in Czech). Algoritmy'83. Published by Slovak Mathematical Society, Bratislava, pp. 131-136.

Kindler, E. (2000) Chance for Simula. ASU Newsletter, **26**, No. 1: pp. 2-26.

Kindler, E. (2001) Computer Models of Systems Containing Simulating Elements. Computing Anticipatory Systems: CASYS'2000 – Fourth International Conference . Edited by D. M. Dubois, Published by American Institute of Physics, Melville, New York, AIP Conference Proceedings 573, pp. 390-399.

Kindler, E., Krivy, I., and Tanguy, A. (2001) Tentative de simulation réflective des systèmes de production et logistiques. MOSIM'01: Actes de la troisième conférence francophone de Modélisation et SIMulation – Conception, analyse et gestion des systèmes industriels. Edited by A. Dolgui and F. Vernadat, Published by Society for Computer Simulation International, San Diego, Vol. 1, pp. 427-434.

Kindler, E., Krivy, I., and Tanguy, A. (2001b) Special approach to reflective simulation. Modelling and simulation of systems: MOSIS'01 – Proceedings of the 35th spring International conference. Edited by J. Stefan, Published by MARQ, Ostrava (Czech Republic), pp. 59-66.

Kindler, E., Chochol, S., and Prokop, K. (1983) Systems of material flow. International Journal of General Systems, **9**, No. 2: pp. 217-224.

Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. (1993) Object-Oriented Programming in the Beta Programming Language. Addison Wesley, Harlow – Reading – Menlo Park.

Meyer, B. (1989) From Structured Programming to Object-Oriented Design: The Road to Eiffel. Structured Programming, **10**, No. 1: pp. 19-39.

Naur P. (ed.), (1963) Revised Report on the Algorithmic Language ALGOL 60. Communications of the Association of Computing Machinery, **6**, No. 1: pp. 1-17.

Savall, M. et al. (2001) YAMAM – un Modèle d'Organisation pour les Systèmes Multi-Agents. Implémentation dans la Plate-Forme Phoenix. MOSIM'01 – Actes de la Troisième Conférence Francophone de Modélisation et SIMulation. Edited by A. Dolgui and F. Vernadat, Published by Society for Computer Simulation International, San Diego, Erlangen, Ghent, Delft, pp. 1037-1043.

Schmidt, B. (1989) Preface. Workshop 2000 – Agent-Based Simulation. Edited by Ch. Urban, Published by Society for Computer Simulation-Europe, Ghent, 2000, pp. 1-4.
Simula Standard (1989) Oslo, SIMULA a.s., 246 pp.
Tanguy, A. (1993) Implementation and application of a modelling environment for manufacturing systems. Application of Distributed & Graphical Simulation [Proceedings of 19th Conference of the ASU]. Published by University of Aberdeen King's College, Aberdeen (Scotland), pp. B-2-1 – B-2-12.
Tchernev, N. (1997) Modélisation du processus logistique dans les systèmes flexibles de production, doctoral thesis, University Blaise Pascal, Clermont-Ferrand, France.