Component-based simulation for a reconfiguration study of transitic systems

Eugen Kindler, Thierry Coudert, Pascal Berruet

Component-Based Simulation for a Reconfiguration Study of Transitic Systems

Eugene Kindler
Ostrava University Faculty of Sciences
CZ–701 03 Ostrava, Dvorakova 7, Czech Republic
kindler@ksi.mff.cuni.cz

Thierry Coudert
Pascal Berruet
LESTER FRE 2734, Université de Bretagne Sud
Centre de recherche, BP 92116
56321 LORIENT Cedex, France

This paper is organized as follows. Part A presents the context of reconfiguring transitic systems and the main idea in implementing the decision step. It comprises sections 1 to 3. Section 3 presents an example that illustrates the concepts presented in the next sections. Parts B and C express the models and principles used to simulate transitic systems, the result of which will be helpful for choosing the new configuration. Part B focuses mainly on models. It comprises sections 4 to 6. Part C focuses mainly on simulation principles. It comprises sections 7 to 10.

Keywords: Transitic systems, object-oriented programming, conveyors with rollers, nesting models, reflective simulation

Part A. Context and Global Objectives

The presented work highlights a major project that concerns the reconfiguration of transitic systems.

1. Transitic Systems

Transitic systems are particular manufacturing systems that transport products from different locations with a fast flow and dispatch them to their destinations. They are composed of different types of conveyors, sorters, elevators, consignments, and automated guided vehicles. Conveyors can be linear, curved, or circular and can have pneumatic jacks, mechanical stops, and switchings. The transitic systems (TSs) are characterized by the following different features:

1.1. TSs are specific manufacturing systems in the sense that no transforming subsystem is considered. The product, often named *parcel*, remains unchanged until it leaves the TS.

1.2. TSs can be characterized as discrete event systems in the sense that their control is mainly discrete. Nevertheless, TSs also include a continuous dimension.

1.3. Because of the speed transfer, TSs are often analyzed in terms of flows, capacities, and costs.

1.4. The parcels run over a network of permanent elements, which can be classified into a small number of types. The reusability concept is very important, enabling designers to reuse some validated parts.

1.5. TSs include a flexibility notion. Different paths enable the same area to be reached. The management of the traffic of such systems has to guarantee good performances management, avoiding deadlocks and collisions.

When the TS is being designed, designers have to confront many problems that arise. The complexity of the TS requires modular approaches for the user to decompose very large and complex design problems into simpler ones. It is necessary to obtain the best approximations between functional solutions and material architecture at the earliest stage of design. The competitive markets imply demands for reducing time to design and implement a new TS. Nevertheless, a TS has to be flexible, robust, easy to maintain,

easy to control, modular, and fault tolerant. Often, several solutions with different costs are possible, and then they have to be evaluated so that the optimal one is chosen. In the evaluation process, computer simulation is very important. Computer simulation models are applied as tools that give data for decision support during the real-time control.

Nowadays, another aspect is more and more present in current TS—namely, the reaction toward failures. This is a constraint that influences accepting a concrete TS and therefore also influences the design. TSs not only have to be efficient from the instantaneous performance point of view. In anticipating their behavior, one must take into account that they also have to react to failure occurrences.

2. Reconfiguration

The response to faults comes from the dependability concept. This concept has been investigated, with the notion of fault tolerance, in automatic control systems [1, 2] and in the computer science field [3, 4]. Fault tolerance can be achieved either by passive or active strategies [2]. Passive approaches use robust design strategies to make the process insensitive to faults. Fault detection and isolation (FDI) algorithms perform the active approach. This paper focuses on some means to implement the active approach and, more particularly, a particular fault isolation step: the reconfiguration process.

The reconfiguration process is a response to faults during the exploitation phase. It consists of reorganizing both the material structure of the process and its control part, allowing it to go on with its production after fault occurrence. This problem is quite complex, especially regarding the number of parameters and functions it requests. These functions take part at different levels:

- The parts present in the system first have to be considered and treated [5].
- The same step has to be performed with parts planned by the scheduler that are to be treated later by the plant [6].
- The system has to make sure that production objectives will be completed in terms of delay.
- From the workshop point of view, the resources have to be settled into relevant modes [7].

From a practical point of view, the reconfiguration process first requires localizing the faulty part of the system, analyzing the impact on the rest of the system, deciding on a new system organization, and then applying corrective actions to reach the proposal organization. Within the framework of the reconfiguration as previously described, presented works give some ideas about implementing and validating the decision step. This decision step requires having knowledge of the potentialities of the system and the operating sequences. Previous works using graph models and graph theory enable determining the following [6]:

- if there is a possibility for the system to go on with the current production;
- if some resources have to be set in the production mode;
- if the part can follow a path to complete its logical operating sequence: a sequence of controls from a parcel point of view

But the presented procedures took very few dynamics parameters into account. Another experimental way is to use forward-looking simulation to assist in complex decision making [8].

Today, for more efficiency, this decision has to be performed online. To reduce the time of the system availability, the calculation should be performed even during the evolution of the system. But is it possible to imagine such a thing? Such a process is indeed very difficult to globally analyze and validate. In most cases, simulation is used to check the appropriateness of the procedure. The point, then, is to have models that enable different levels of simulation.

The aim of this paper is to propose a global methodology that enables designers to build models that test the decision step of the reconfiguration process. In particular, this paper focuses on nested simulation and reflective simulation of transitic systems.

3. Example

The transitic system, which is used for illustrating concepts and methods presented in the following, is introduced in this section. Its scheme is presented in Figure 1.

The long horizontal lines represent parts where the transport is performed by motorized rollers. The parcels occurring there can be stopped by mechanical stops (thrusts) or moved out from these parts by pneumatic jacks. The jacks are used to move the parcels along the free rollers, which are represented by vertical lines. There are five working areas in the system. Jacks also perform the transport along the parts of the working areas, which are represented by short horizontal lines. At the stopper, three parcels can occur—namely, the parcel that has entered, the parcel that is being processed, and the parcel that is waiting to return to the motorized part. The synchronization at the system—namely, at the working areas—has to occur with respect to the movements and positions of the jacks, so that the jacks do not enter into mutual accidents. In Figure 1, the scales of the vertical and horizontal dimensions differ a bit to make the scheme more transparent.

The parcels enter the system at the left lower edge and continue according to the marked arrows. They can make a “deviation” to a working area according to their technological program and to the state of the working area. Each parcel can make several cycles until it is processed according to the rules given by its technological program, and then it leaves the system at the same place where it entered.

Part B. Models

This section presents models and techniques used to implement the forward-looking simulation of the TS. The main

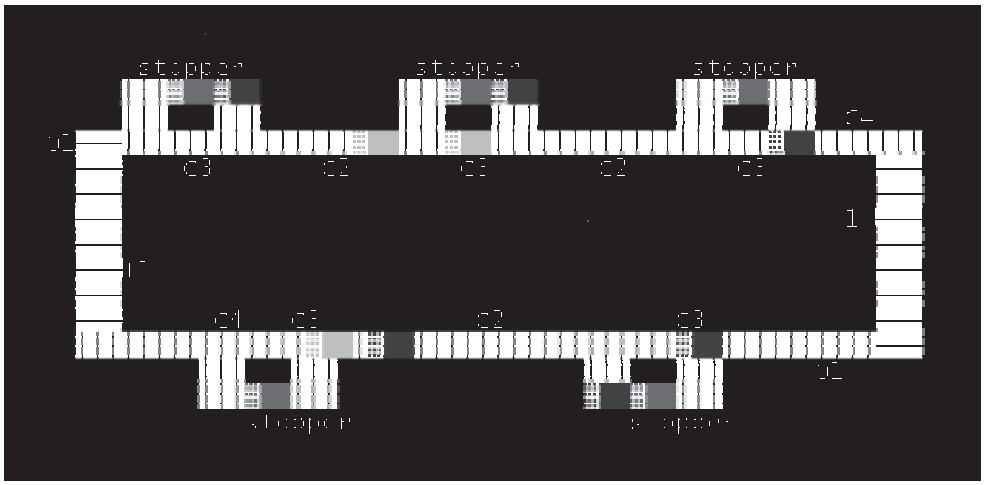

Figure 1. Instantaneous state of a model of the IUP

goal is to build models implementing the simulation of a decision process that is based on the simulation.

4. Component-Based Simulation Models of TS

The mental process of describing a TS can be considered in the following way. The designers have a list of the sorts of components at their disposal. The tendency is to minimize the number of component types. Nevertheless, in principle, there is no limit on the number of components of the same sort that can be introduced into the concrete TS. The chosen components of various sorts are configured into a "base" (i.e., into a structure) over which parcels are moved. The base is considered "pseudo-constant," which means that during a normal operation of the TS, its base does not change and the parcels move over it; only during some exceptional events can the base be modified. A typical example of such an event is a failure occurring to a component of the base; if that event does not cause the TS to collapse, then it is reconfigured to maintain the availability of the TS. During the reconfiguration (or another change of the base), the rules for moving parcels are often different from those applied to the phases when the base is constant.

The programming tools are important because they shorten the transformation of ideas from the human mind into the physical world of the computer. Namely, the appropriate programming tools perform the simulation in a way that is never explicitly formulated: the user is not forced to describe algorithms; he or she can describe the simulated system, and the description is automatically translated into something comprehensible for the applied computer. The help provided by the simulation programming tools is essential, as simulation is generally applied for studying complex systems.

The simulation programming tools offer synthesis in designing the TS. The simulation model M of an individual transitic system S can be constructed as an analogy to building S in the following way:

4.1. A list of component types should be mapped to a list L of classes (abstract knowledge representations). Characteristically, the applied simulation programming tool does not have such classes. Therefore, it is suitable to apply an object-oriented programming tool and to define L in it. This tool allows the user to organize the classes in a hierarchical way that could reflect the hierarchical ordering of the types of components. For example, the general class *head* of lists can be "specialized" into class *tech_program* of the technological programs and into class *trace* of the traces. Tools introduced for *head* (e.g., serving to determine whether the given list is empty, what its first and last elements are, or how many elements it has) can be applied for the instances of *tech_program* and for trace as well. Nevertheless, for any of these classes, one can introduce other tools. An example is a tool that determines whether there is a parcel at a trace or a "reader" of the technological program, which informs what technological step has just been performed for a certain parcel. Class *tech_program* can be further specialized into certain special sorts of the technological programs. Similarly, class *trace* can be specialized by adding tools for its presentation at the computer display (note that, in principle, there are many ways to visualize models).

4.2. Instances $\{I_1, I_2, I_3, \dots\}$ of the classes of L are generated as components of M; this is a process analogous to the design phase of S, when one determines

the components $\{C_1, C_2, C_3, \dots\}$ forming the base B of S.

4.3. The instances $\{I_1, I_2, I_3, \dots\}$ are linked to form a substructure β of M that is a certain homomorphic image of B. The object- oriented programming enables the user to define the classes of L so that the linking is described almost in terms used for the linking of $\{C_1, C_2, C_3, \dots\}$ when S is being described. β can be metaphorically called the "base of M."

4.4. L can be enriched from classes of the parcels. Those classes reflect the sorts of parcels expected in S. In the contents of the classes, the rules for manipulating the parcels can be reflected. Let such rules be called "life rules of the parcels." Suitable object-oriented languages allow the user to formulate the life rules similarly as algorithms. Such languages are not only object oriented but also process oriented or agent oriented. Let them be called languages with two orientations, or *2-O- languages*. The places of B, at which parcels enter S, can be interpreted in β as parcel generators. Such generators can be defined in 2-O-languages without any essential problems.

The list L can be considered as a simulation language defined at the given object-oriented programming tool.

5. Beginning of Implementation

The number of 2-O-languages is not many. We know about Java, Beta [9], Simula [10-12], and Modsim. For certain reasons that will be explained in section 10, we applied Simula. In it, we define the following classes oriented for the components of the base.

Class *trace* reflects the linear parts of the base. Every trace has two ends that are instances of class *junction*. The instances of *junction* serve as connections of traces. Class *junction* corresponds to a connection of two traces. It has two attributes, *source* and *target*, which refer to instances of *trace*. Let J be an instance of *junction*. Then the end of its *source* is joined to the beginning of its *target* through J. Class *junction* is specialized into two classes, *branching* and *inverse_branching*. Each of them has another attribute: *branching* has *second_target*, and *inverse_branching* has *second_source*. If a parcel moves along a trace and then meets a branching at its end, its further movement can continue either along *target* or along *second_target*. To serve the decision, a class *switching* is formulated. Its instances behave similar to the switches used in railway systems. However, the choice between *target* and *second_target* is not bound to switches. Class *inverse_branching* is a certain reflection of a confluence of two traces; a parcel can enter its instance from two traces—*source* and *second_source*—and then it moves along *target*.

Every trace has its *direction* and its *length*, which are expressed in the number of instances of class *place*. These instances represent certain atoms. A place can have only one parcel, but a "voluminous" parcel can be placed at more places contemporaneously. The parcel of a place is pointed out by the attribute *contents* of the place. The places enable the user to discretize the continuous movement in the system. They can also serve as animation: method *show* represents the places on the computer display. The parcels also have such a method; when a place contains a parcel (i.e., when the value of its *contents* differs from *none*) and is requested to perform *show*, the parcel delegates a place to its *contents*. The method *show* enables one to visualize the system from a bird's-eye view. Other attributes of class *place* are integer coordinates x and y, which correspond to the position of the place image at the display.

Traces are considered as lists of their places. In other words, class *trace* is a specialization of the class of lists, which is introduced as a standard class *head* in Simula. A detailed analysis shows that several simplifications were admitted according to the objectives described previously. First, a trace cannot change its direction: a curving trace can be replaced by several traces of different directions. Second, only four basic directions can be introduced: *right*, *left*, *up*, and *down*. These are instances of an abstract class *horiz_vert*, and a binary operation *sub* of angle subtraction exists among them. The third simplification consists of limiting the number of connections of traces to three. A contact of four or more traces can be replaced by two mutually close contacts.

These simplifications enable one to introduce procedures and methods relating to class *trace* and to construct the base simply, transparently, and without errors. The names *last_trace* and *last_but_one_trace* are given to the pair of the last constructed traces. For example, the scheme outlined in Figure 2 can be described as follows (suppose we begin from the place marked by a cross, and the coordinates of the place are <3,4>):

```
new_trace(3,4,down,11);
last_trace.continue_to(right,15);
last_trace.branch_to(up,11,right,12);
last_but_one_trace.continue_to(left,15);
last_but_one_trace.continue_to(up,11);
last_but_one_trace.connect_from(last_trace);
```

The main classes of temporary elements are *colis* and *parcel* (*colis* is a French term that is equivalent to *parcel*). While *parcel* represents an atom, *colis* represents a list of such atoms to enable the modeling of parcels that occupy more than one place of the traces. Class *colis* is enriched by a Boolean attribute *forwards* and by a real attribute *speed*. The parcels have attributes *way* (the trace at which the parcel is placed) and *membership* (the colis of which the parcel is a component). Both classes have methods *move* and *return*, which perform a step of one place in a robust manner (i.e., eliminating all phenomena that could contradict the physical laws). When the classes

colis or *parcel* perform *move* or *return*, that procedure automatically checks whether the change in position does not infringe on the kinematics or statics laws.

It appears suitable to introduce an abstract class called *driver*, which represents an individual algorithm that controls a colis and, among other things, decides on its choice at a branching. The separation of class *driver* (and its subclasses) from the other classes of L enables one to separate the requirements of the universal laws of kinematics from the formulation of manmade control rules. The user of the classes only has to formulate the control rules as "life rules" of a subclass of *driver* while having use of the other classes, which do not need to be specialized. The list L of the mentioned classes really represents a simulation language. Class *driver* is declared as a subclass of Simula standard class *process*, which allows the user to schedule events of different instances of this class at one flow of the simulated Newtonian time. Simula has special excellent properties for robust scheduling that can be applied so that the scheduling of events formulated in methods such as *move* and *return* of a parcel are interpreted by its own driver. In other words, the length of time necessary for the parcel to move is caused a realistic waiting time of the algorithms that rule the parcels.

6. Next Development

The noted aspects of the mentioned classes allow a user to simulate TS with the following properties:

6.1. Every parcel is able to move at its own rate, which may be different from the other parcels.

6.2. The parcels are able to move in both ways; in other words, the instances of *branching* could sometimes behave as those of *inverse_branching* and vice versa.

6.3. The lengths of the parcels can be different.

But the list L is not suitable for the simulation of other TSs. Problems especially appear with the TS when the movement results from the interactions of components that have to be protected against mutual accidents and against accidents with the parcels. This concerns mainly TSs with pneumatic jacks and mechanical stops. Namely, the jacks represent "pseudo-permanent" elements of the system, as they can be seen as static elements from the structural point of view: they are always present in the system (e.g., like traces and unlike parcels), and unlike parcels, each of them is always attached to the same trace. But they also have some dynamic evolution, and a part of their structure can move into the space. They also interact with the parcels and can have conflicts with each other.

The set L had to be enlarged. Nevertheless, it was already too big, and the enlargement would lead to rather large models. But could see that the physics of the mentioned pseudo-permanent components would cause phenomena to work against the aspects noted in 6.1 to 6.3. Therefore, the optimal solution was to reduce the components of L that permitted the mentioned aspects and then to add classes corresponding to the mentioned pseudo-permanent components.

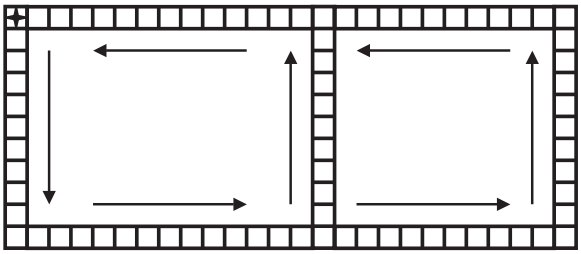

Figure 2. Simple example of a base

Through this method, another list L^* of classes could be constructed, containing class *verin* (representing the jacks) and *bute* (representing mechanical stops). Class *verin* was introduced as a specialization of Simula class *process* and allowed the automatic scheduling of events to include events of the jacks, too.

Lists L and L^* were then enlarged by special tools for animation, using a class called *TERMINAL*. This class has been added to the Simula implementations for PCs under various operating systems (WINDOWS, OS2, LINUX, UNIX, and even MS/DOS and XENIX), although it is not an official component of the ISO standard of Simula. It allows an effective animation, using the surface of the computer display discretized as a matrix of 80 columns and 50 lines. This discretization appears to be sufficient for modelers who wish to watch the events that interest them. Other methods enable modelers to collect data from selected parameters and to display a synthesis of their evolution during the simulation interval. A projection of an instantaneous state of the animation is presented in Figure 2.

Part C. Simulation Principles

7. Internal Simulation: Simulation Adapted to TS Exploitation Problematics

7.1 Examples of Problems

The classes mentioned in sections 3 and 4 can be used to build models of a TS with a rather sophisticated transport substructure. The base of the system (cf. section 3) is rather simple. Nevertheless, that TS illustrates the difficult problems that often occur during its exploitation about the decisions that should be made and applied during the functioning of the TS. The decisions have to be

formulated according to the instantaneous state of the given TS so that their consequences do not negatively influence its operations. The obstacle is that the consequences can be influenced by many elements that operate and move simultaneously, forming a network of events that is so complex, no mathematical formula is able to compute it in a detailed way. Let us present some simple examples:

7.1. At time T, the system is operating (i.e., a rather great number of parcels are in it). A parcel P comes to the IUP and is to be placed at the main cycle. Should it be performed immediately or after some time? Two variants can be admitted.

7.1.1. P enters immediately, and after having moved a rather short path at the main cycle, it enters a working area.

7.1.2. No working area is available, and P will perform a full turn at the main cycle; at time $T+K$, it returns to the place where it entered.

From the viewpoint of P, the situation of 7.1.2 may not be different from the decision to let the parcel wait K time units and only then to place it at the main cycle. But for the global transport at the system, P can present a certain moving barrier—namely, to evacuate the parcels that should return from working areas to the main cycle, which might delay their transport. Letting P wait outside of the IUP until a relevant part of the system is no longer occupied would be better. But how do we anticipate which of the mentioned variants will come?

7.2. A working area should be assigned to a parcel P that is moving along the main cycle. Many different methods are available, two of which are transparent (or mutually extreme) and can be well algorithmized.

7.2.1. The considered working areas are able to perform the next technological step and contain a free place; P is assigned to the considered place it is able to reach in the shortest time.

7.2.2. The considered working areas are able to perform the next technological step and are not currently elaborating any parcel; P is assigned to the nearest free working area.

The variant 7.2.1 may place P at a certain working area that is elaborating another parcel Q. Then, P would have to wait until Q is ready. It is possible that P would be soon elaborated at a working area that is more distant but free. The variant 7.2.2 may place P at another working area that is more distant but free. Nevertheless, before P enters this working area, another parcel may enter it. Then P has to wait; it would be better for it to get a working area according to 7.2.1. Interestingly, computer simulation of the IUP pointed out cases in which a parcel had to make several complete turns along the main cycle; being served according to 7.2.2, it always accessed the assigned free working area too late.

7.3. A fault occurs in a working area, causing the working area to be out of order so that it cannot be used until it is repaired. During that reparation process, the TS main cycle has to be stopped. In this case, should the whole production process along the main cycle be stopped and the system repaired, or should the process continue during some time, using a smaller number of redundant working areas?

7.4. A technological step can be performed at several working areas that, nevertheless, differ in their parameters. For example:

7.4.1. A working area $W1$ has rather better resources, and the elaboration at it takes a shorter time than another admissible working area $W2$. Which of them should be assigned?

7.4.2. Compared with $W1$, a smaller number of technological steps could be performed at $W2$. Should that be respected during the assignment of the working area?

Note that 7.4.1 could imply a similar situation as for 7.4.2: the working area with better technological parameters could be a subject of greater interest.

7.5. Due to product flexibility, some technological steps may be swapped. What is the best choice for the next technological step of P?

The list of the mentioned examples could be enlarged by many other ones, but we hope that, namely, 7.1 to 7.3 are rather instructive for any simulationist; they do not demand special knowledge about the design, control, and configuring of the TS.

7.2 Simulation as a Way to Solve

Today, modern computers are, in principle, able to generate and evaluate many combinations of parameters, the number of which can be far beyond the abilities of the non-computer-oriented person who would like to understand and control complex production/logistic systems. A computer can have a much more detailed and finer insight into what happens in time.

Anticipation of the possible consequences of the decisions and choices among the variants outlined in this section could be obtained through forward-looking simulation. The computer C that controls a given system S can be used to generate the decisions that are physically possible and then simulate the future of S according to the generated decision, so that C can come to quite a good decision.

Let the simulation models used for the mentioned simulation be called the *internal models* of S, and let the experiments with them be called the *internal simulation* of S.

In any case, to have data serving for the control, C must be connected with a detected apparatus that gives it information about the state of S. The same data can be used to feed the internal models. In principle, there is no problem in performing and applying the internal simulation. One only needs to give the computer suitable software, and the internal models. These models can be more easily implemented by component-based simulation techniques outlined in part B (sections 4-6).

8. External Simulation: Simulation Adapted to TS Design Problematics

This section tackles the simulation of designed system S. The reason to simulate such a system is to get support data for the optimal design. Let the simulation models used for such a simulation be called the *external models* of S, and let the application of them be called the *external simulation*. These models have many properties similar to the internal models. Among others, they can use almost the same lists of classes (see L and L^*, introduced in sections 4-6), but they differ by their application. While the simulated time of the external models runs for a rather long interval of weeks and months, that of the internal models runs from seconds to hours. The external simulation often interprets various sets of investments. The variants represented by external models differ by structuring the investments. The internal simulation interprets various decisions using the existing (i.e., constant) set of components (note that a component that is not operating because of its faults does not cease being a component of the system; it only changes the values of its parameters). Therefore, the variants simulated in internal models only differ by the components' parameters.

A universal task for computer simulation is to manipulate models that give true data about the systems that they simulate. The internal simulation should serve to provide data that support the owner of a simulated system in having optimal use of that system. In section 7, we illustrated the benefit that the internal simulation can provide to the operations of the TS. The consequence should be that when the TS is being designed, the internal simulation should also be taken into account. In other words, the external model M during a design system S should reflect a good image J of computer C that is a part of S. "Good image" means that J performs internal simulation like C. If the simulation ability of C were neglected in J (or if even J were neglected in M), the same model M would exhibit two errors in design:

8.1. If the internal simulation were necessary, M would lead the designers to overvalue the dimension of the TS; in other words, M would give false information about S.

8.2. If M gave true information about S in the realization of S, the internal simulation would be superfluous.

Section 7 illustrates that the internal simulation of S often helps in solving decision problems. Thus, the absence of a computer J performing internal simulation is an error.

Let M be constructed without errors. Then it has to contain a (dynamic) structure $\{\Sigma\}$ of images of the components of S and, besides them, J. J should be able to have internal models of S and should have a (dynamic) structure $\{\sigma\}$ similar to $\{\Sigma\}$. Both $\{\Sigma\}$ and $\{\sigma\}$ represent the subsystem of S, which is composed of its components but parameterized a little differently.

Let a simulation of such a model M be called *reflective simulation*.[1]

9. Reflective Simulation Principle

After setting the problem of time in reflective simulation, this section points out the interest in three oriented languages to implement reflective simulation.

9.1 Time in Reflective Simulation

Let us present an overview of system S modeled by M in reflective simulation.

S has hard components and a computer C. All these components exist in a common (real) time. Let that time be called *r-time*. In this time, C sometimes performs a simulation experiment for forward-looking simulation. During this phase, C continues to exist at r-time. When C does not need to simulate, it also continues to exist in r-time. However, its nonsimulation actions can be more or less neglected. Let $[t_1, t_2]$ be the interval of r-time, during which C performs a simulation experiment. Naturally, $t_1 < t_2$, but often, the equality between t_1 and t_2 can be assumed, as the duration of the forward-looking simulation is so short that it can be neglected. During that interval, something happens to C, which is interpreted as an internal model m. m has its own simulated time, *s-time*. Both times have similar properties as they both can be seen as Newtonian ones. But both times also differ, principally in the interpretation of their values: for example, at r-time t^*, the computer C is said to simulate what shall happen at time t^+: $t*$ is r-time, and t^+ is s-time.

If S is modeled by M in the external simulation, its properties described in the preceding paragraph should be reflected in the description of M. Therefore, M is described

1. Note that the following phenomenon is theoretically possible: an external model of a system S contains an internal model of a system s that is completely different from S. It might be a situation in which we simulate an enterprise, which has computers that simulate something for another enterprise. Such a phenomenon is covered by the concept of nested simulation; it can be implemented by the programming technique outlined later. But that is not the case in this paper, for which it is characteristic that both the modeled systems are the same. Therefore, we introduce the term *reflective simulation* for this special case of nested simulation.

by an association of components (instances of classes existing in a certain list L^*; see section 6) and a computer. This computer is seen as an instance J of a certain class Γ that is also in L^*. The classes are related to the same simulated time *rt*, corresponding to r-time. But Γ also has to include phase ϕ of forward-looking simulation (i.e., the internal model m). The description of m is very similar to that of M; only the description of J is omitted there. Nevertheless, there is an important difference between these two descriptions: while the description of M is related to *rt*, that of m should be related to another simulated time *st*, corresponding to s-time; *rt* and *st* cannot be mixed.

9.2 A Solution for Implementation: Languages with Three Orientations

An advantage provided by the simulation programming tools, such as simulation languages and simulation packages, is that their users are not forced to describe what should happen during the simulation experiments. Instead, they describe the simulated system, and the description is automatically translated into something like the corresponding computer program or algorithm. Another good source of help is the automatic handling of the values and events related to the simulated time. Examples concern event scheduling in the discrete event simulation tools or integrating according to the simulated time in the continuous-system simulating tools. But all these simulation tools relate the described system to a unique time. The problem of reflective simulation is that it needs several times, at least *rt* and *st*.

The solution of the problem consists of using programming languages that are object oriented, agent oriented, and block oriented. Let them be called *languages with three orientations*, or *3-O-languages*.

The advantages of object and agent orientation were outlined in sections 4 and 5, in relation to the lists of classes L and L^*. Moreover, one can enrich such lists with tools that reflect the essential properties of Newtonian time—namely, event scheduling.

Block orientation enables the user to include blocks into the life rules of the classes. *Block* is a phase that has "local" entities. These entities are at disposal when the "life" of an instance is inside of the block. Block b can be a subblock of another block B. In such a case, when the life is inside b, the entities of both b and B are accessible.

The concept of a block was first investigated in ALGOL 60 [13], but this programming language was not object oriented. Nevertheless, the first object-oriented language that Simula accepted was the block concept. Like structured programming, block orientation was refused by the world-professional community of programming theoreticians. The blocks slowly gained acceptance after structured programming had been fully replaced by object-oriented programming.

Evidently, the 3-O-languages are 2-O-languages. The reverse implication does not hold, however. Modsim is a 2-O-language that is not a 3-O-language. It seems that only Simula, Beta, and Java are 3-O-languages.

10. Implementation with Simula

This section deals with some mechanisms that are essential for implementing reflective simulation: block orientation, the avoidance of transplantation, and model copying. In this way, advantages provided by Simula are emphasized.

10.1 Implementation Using Block Orientation

Block orientation of a 3-O-language enables a user to introduce blocks that have local classes. These classes are accessible only inside the given block. Let A be a class having block B in the life rules, and let a list L of classes be introduced among the local entities of B. Let R be any instance of class A. When the life of R enters B, R can be viewed as having become a modeler that uses L for viewing its models. When L contains tools describing the Newtonian time, R can be viewed as a simulationist. When the life rules of R omit B, R loses its modeling/simulating ability. Naturally, the life rules can contain a cycle containing B, and then R can be viewed as returning to its "modeling/simulating" activities.

The combination of the three orientations is rather difficult to understand. For that reason, graphical representations were developed, called *Mejtský's diagrams* [14, 15], which use a two-dimensional representation of any block (represented as a rectangle) and an instance of a class (represented as a circle). The local entities can be imagined by being placed inside such a domain. According to the life rules, a life is represented by a horizontal line inside the circle or along the upper edge of the rectangle (see Fig. 3).

The introduction of two time axes is simply made by introducing T-tools into two blocks so that one of them is a subblock of the other one. In Simula, the T-tools can be introduced into a block by a "prefix" of that block. Such a prefix is the name of the set of T-tools. Simula offers a standard class called *SIMULATION*, which represents the T-tools. The "outer" block represents the external simulation. It contains the description M of the external model. When the computing process enters that block, M begins to exist. M exists and operates until the computing process accesses the end of the block. In the block, the list L^* of classes is introduced. The instances of the classes represent the components of the simulated system. The inner block is devoted to the internal simulation.

As previously mentioned, the class Γ belongs to L^*, and one instance of it is the simulating computer C. Among the life rules of that class, the subblock represents the internal simulation (based on model m). The scheme is outlined in Figure 3, in which the blocks having local classes are emphasized with a triple dotted line and rounded edges.

The principle is rather simple. The outer block operates with a list L^*, whereas L is sufficient for the inner one. Note that many Simula implementations offer a possibil-

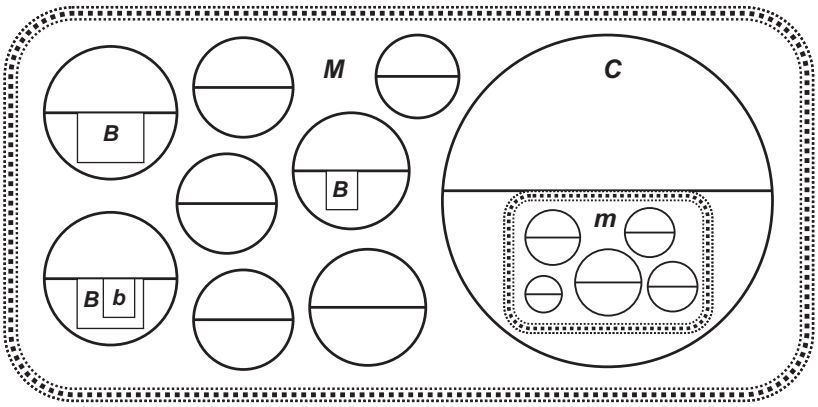

Figure 3. Mejtský's diagram of nested simulation

ity for the system linker to build only one exemplar of L into the executable model, although it should behave as occurring twice in the program: once in the outer block and once—with a completely different interpretation—in the inner block.

Simula has very suitable T-tools, and this is one of the reasons why we use this language.

10.2 Transplantation

The presence of L in two blocks could be a source of serious programming errors, called *transplantation*. Such an error consists of mixing instances belonging to the outer block with those belonging to the inner block. For example, a parcel belonging to the internal model would be described as entering into a queue introduced for the external model. Such an error may not be discovered immediately; a collapse of computing comes after millions of additional computing steps, when the components of both models are very much interchanged. In such a case, the true reasons of the error are difficult to discover.

Java does not prevent transplantation, and Beta has to make many tests (during the simulation run) to prevent it. The tests make the computing rather slow. Simula has syntactical rules that introduce certain limitations, minimizing the occasions of making transplantation errors; therefore, many such errors can be discovered during compilation. Thus, checking possible transplantation during the computing phase is very rare, and the simulation is hardly lengthened by the tests.

That is another reason why we use Simula.

10.3 Principles of Model Copying

In the real world, the internal model m arises during the existence of the real system S. The modeler (a person or an automatic modeler programmed to run on the controlling computer) has to generate m. The modeler overviews the state of S and maps it as the initial state of m. Then, m proceeds automatically, using the classes of list L.

This process has to be reflected in the external model M of S. The image C of the computer should also be a certain image of the modeler. It should be programmed so that it "overviews" the instantaneous state s of M. According to s, it generates the initial state of the internal model m, and then it lets m operate.

Suppose the request for m generation occurs at r-time t. In general, some components of S operated before t. Let one of them be called W. At time t, it may make some continuous action that began at r-time t' and will end at time t''. In general, $t' \leq t \leq t''$, but habitually, $t' < t < t''$. When S is reflected by its external model, M, an image w of W exists in M. Its operation during time interval $[t', t'']$ is often reflected by an "empty" phase: w is programmed as if it held during $[t', t'']$; at time t'', it accepts new values of attributes that, in reality, were continuously changed during $[t', t'']$. For time t'', the "life" of w is scheduled by a certain life rule ρ, which assigns the new values. This is a common practice in discrete event simulation [16], which has existed since the end of the 1950s and is appreciated by simulationists.

Therefore, at s-time t, w is in a "suspended" state (it is not operating at the computer), and it points to ρ with information to perform it when s-time is equal to t''. When the internal model m is generated at the same time as t, the following steps have to be performed concerning w:

10.1. For m, the image w^* of W is created as a copy of w, including all its attributes.

10.2. The next life rule to be performed by w^* is the image ρ^* of ρ.

10.3. The time when ρ^* has to be elaborated corresponds with t'' (i.e., it is scheduled for time $\Delta = t'' - t$, after beginning the operation of m).

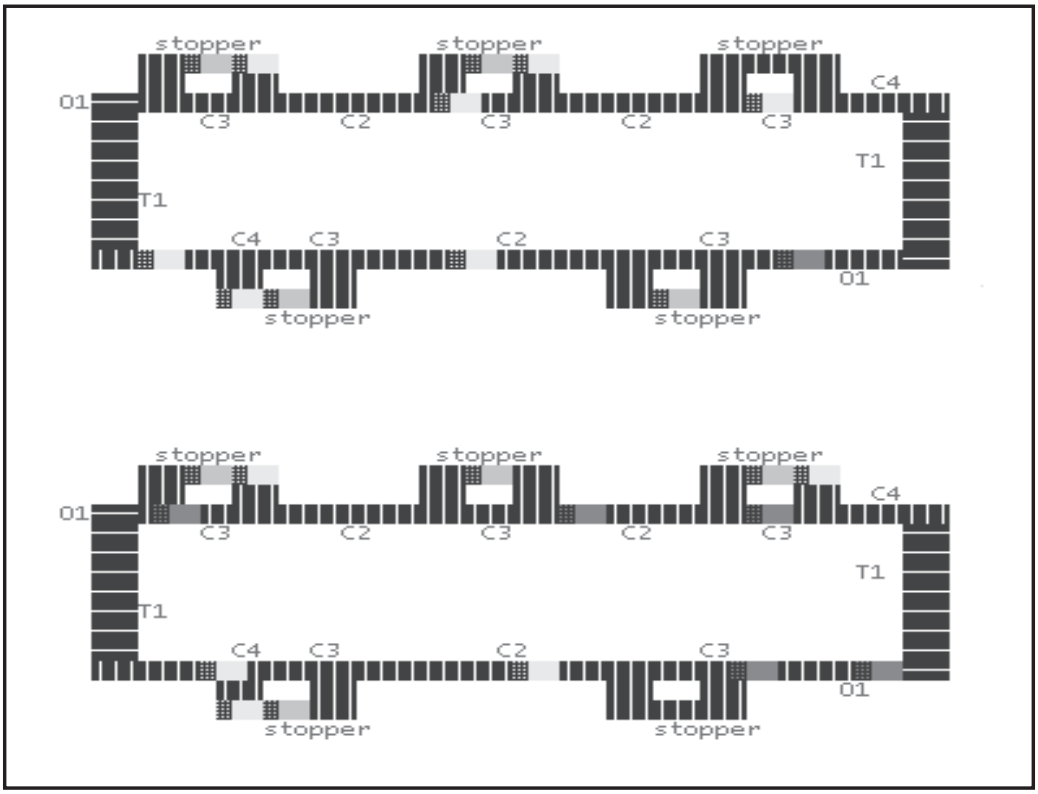

Figure 4. Reflective simulation: The working area on the right-hand side of the upper section is out of order

Simula provides relevant programming methods in performing such steps. A general description of them can be taken from Kindler [17, 18]. Therefore, it was no problem to implement external models of systems that contain controlling computers that use internal simulation models.

An illustration related to the system (see section 3) is presented in Figure 4. It is a model for solving the questions outlined in 7.3. Let the lower part be called the *e-part*, as the external model of the system is shown there. When a failure occurs and causes a certain working area to be inaccessible, the internal simulation model is generated and runs. Its animation is shown in the upper part of the figure, called the *i-part*. The consequences of failure are then examined. For example, respecting the hypothesis that the operation of TS continues, it enables answering whether the queues of the arriving parcels will grow. The impact of different elaborated decisions is analyzed to choose the best adapted decision.

11. Conclusions

The models implemented according to the principles described above operate well. Other models are in progress. This kind of simulation provides data supporting decisions in a reconfiguration context. The interest in this subject is very important as several internal models based on the same external model can easily be elaborated at the same time and can run with different values for their parameters.

Another interesting aspect of reflective simulation is as follows: the models were implemented to react to input data. They concerned the technological aspects (rates of moving and of pneumatic jack operations, frequencies and distribution parameters of entering parcels into the system, the duration of manipulation at the working areas, etc.). But we also introduced something as the rate of the simulating computer. Although one can assume that, in a realistic application, the rate should be accepted as infinite, it is no problem to build the rate into the external model. The designer is asked to enter the duration of a certain step for running the internal model. Zero is the default value, but a positive value can be entered. If the default value is not changed, the observed behavior is that while the i-part is changing, the e-part is frozen. If the default value is changed, some changes can be seen at the e-part in certain moments, while the i-part of the display scene changes rather quickly. This represents a situation in which a slow

computer is simulating something that may be happening in its environment. This experiment is useful for validating whether the decision algorithms can be calculated during the evolution of the real system. The impact will reduce the time the system stops for withdrawal and minimize the tardiness in completing the process. The reactive piloting problematics could then be tackled by this kind of technics.

One can expect that reflective simulation will be an important aspect of modeling the computerized production, ecology, and service system in the near future. The presented results concern but are not restricted to transitic systems. Others works could consider FMS or cooperative systems. A small number of other implementations have been made. The first case was a demonstration model of a bank controlled by a person who simulates possible consequences of his decisions (according to the simulated data, he modifies his decisions; see Kindler [19]). A more important study concerned computing optimal traces in a dynamically changing network, which was applied in modeling container yards [20].

9. References

[1] Patton, R. J., and J. Chen. 1994. Review of parity space approaches to fault diagnosis for aerospace systems. *Journal of Guidance, Control, and Dynamics* 17:278-85.

[2] Frank, P. M. 1996. Analytical and qualitative model-based fault diagnosis: A survey and some new results. *European Journal of Control* 2:6-28.

[3] Laprie, J. C. 1993. Dependability: From concepts to limits. In *Proceedings of the Symposium on Safety of Computer Control Systems (SAFECOMP'93)*, Poznan, Poland, pp. 157-68.

[4] Dugan, J. B., S. J. Bavuso, and M. A. Boyd. 1992. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability* 41 (3):363-77.

[5] Berruet, P., A. K. A. Toguyeni, and E. Craye. 1999. Considering parts in progress in a reconfiguration procedure for FMS. In *Modern applied mathematics techniques in circuits, systems and control*, edited by N. E. Mastorakis, 366-73. Athens: WSES Press Editions.

[6] Berruet, P., A. K. A. Toguyeni, S. El Khattabi, and E. Craye. 2000. Toward an implementation of recovery procedures for FMS supervision. *Computers in Industry* 43:227-36.

[7] Dangoumau, N., S. El Khattabi, and E. Craye. 2000. Modes of management for flexible manufacturing systems. In *Proceedings of the World Automation Congress (WAC 2000)*, Maui, Hawaii.

[8] Tomizuka, M. 2002. Mechatronics: From the 20th to 21st century. *Control Engineering Practice* 10 (8): 877-86.

[9] Madsen, O. L., B. Møller-Pedersen, and K. Nygaard. 1993. *Object-oriented programming in the Beta programming language*. Reading, MA: Addison Wesley.

[10] Dahl, O.-J., and K. Nygaard. 1968. Class and subclass declarations. In *Simulation programming languages*, edited by J. N. Buxton, 158-74. Amsterdam: North Holland.

[11] Dahl, O.-J., B. Myhrhaug, and K. Nygaard. 1984. *Common base language*. 4th ed. Oslo, Norway: Norsk Regnesentralen.

[12] *SIMULA standard*. 1989. Oslo, Norway: Simula.

[13] Backus, J. W., J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauguois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. 1960. Report on the algorithmic language ALGOL 60. *Numerische Mathematik* 2:106-36.

[14] Mejtský, J., and E. Kindler. 1980. Diagrams for quasi-parallel sequencing—Part I. *SIMULA Newsletter* 8 (3): 46-49.

[15] Mejtský, J., and E. Kindler. 1981. Diagrams for quasi-parallel sequencing—Part II. *SIMULA Newsletter* 9 (1): 17-19.

[16] Banks, J. 1990. The simulation of material handling systems. *SIMULATION* 66:261-70.

[17] Kindler, E. 1999. Chance for Simula. In *Proceedings of the 25th Conference of the ASU—System Modelling Using Object-Oriented Simulation and Analysis*, 25-53. Kisten: ASU.

[18] Kindler, E. 2000. Chance for SIMULA. *ASU Newsletter* 26 (1): 2-26.

[19] Kindler, E. 1994. Simulation of systems containing simulating objects. In *Simulation und Integration'94*, edited by P. Lorenz, 65-76. Dortmund: ASIM.

[20] Kindler, E. 2002. Nesting simulation of a container terminal operating with its own simulation model. *JORBEL (Belgian Journal of Operations Research, Statistics and Computer Sciences)* 40 (3-4): 169-81.

Eugene Kindler *is an associate professor at Ostrava University Faculty of Sciences, Dvorakova, Czech Republic.*

Pascal Berruet *is an associate professor at LESTER Laboratory, University of South Brittany, Lorient, France.*

Thierry Coudert *is an associate professor at LESTER Laboratory, University of South Brittany, Lorient, France.*