UNIVERSITY OF OSTRAVA

Institute for Research and applications of Fuzzy Modeling

Anticipation Models in BETA

Frantisek Hunka

Research report No. 80

2005

Submitted/to appear:
CASYS'05 Seventh International Conference on Computing of Anticipating Systems

Supported by:
Institutional research scheme MSM6198898701 of the Czech Ministry of Education, Youth and Sport

University of Ostrava
Institute for Research and Applications of Fuzzy Modeling
30. dubna 22, 701 03 Ostrava 1, Czech Republic

tel.: +420-59-6160221 fax: +420-59-6120478
email: frantisek.hunka@osu.cz

Anticipation Models in BETA

Frantisek Hunka
University of Ostrava
Institute for Research and Application of Fuzzy Modeling
30. dubna 22, 701 03 Ostrava1, Czech Republic
frantisek.hunka@osu.cz
draft

Abstract
Simulation and application (non-simulation) models used in man-made systems are more and more sophisticated and intellectually demanded to comprehension. To cope with the growing complexity at one hand and capability for comprehension on the other hand, it is necessary to exploit powerful facilities, in the sense of modeling and design, to achieve desired goals. This paper deals with the simulation framework and the cluster analysis application that are used by anticipatory systems and that draw from powerful abstraction mechanisms for modeling and design that provides the BETA object oriented language. The paper describes the unique features of the BETA language, demands of the simulation framework and the cluster analysis application and achieved results in the area of anticipation models.

Keywords: Anticipatory systems, simulation models, BETA, cluster analysis, object oriented modeling

1 Introduction

Anticipatory systems can use for its activity simulation models and with the progress of computing technology the application of simulation models will be more frequent. These simulation models are limited in their modeling and simulation capabilities by the underlying programming system they are built on. Our aim is therefore to choose underlying programming system, on which the framework is built, with the great facilities for modeling design and implementation as well. This advancement can greatly enhance the possibilities of the simulation framework and consequently anticipating models as well. The other example described in this paper is the cluster analysis application, which draws from unique features of BETA as well but it is an example of non-simulation model used by anticipatory system.

From our previous experience we highly appreciated programming systems coming from Scandinavian School of object orientation. Scandinavian School of object orientation is foremost aimed at modeling and design facilities. Reusability in this approach is taken as a side-effect of good abstraction mechanisms and good abstractions can be used several times. There are two basic representatives Simula and the BETA object oriented languages. BETA as a newer programming system draws to some extent from Simula experience but has quite new features that differ from Simula. One different feature recognizable from the beginning is that BETA does not have a “Simulation” class (a class enabling description of models of systems existing and developing in Newtonian time) though there are some good prerequisites for it.

2 BETA language characteristics

BETA is a modern object oriented programming language in the Simula tradition. It provides excellent tools foremost for object oriented modeling and design and differs in this way from those languages purely oriented on reusability. From the other view it may be characterized as an object oriented, process (agent) oriented and block oriented language. Native compilers for BETA exist for a number of platforms. Recently work has been going on to port BETA to the bytecode based platforms of SUN Java and Microsoft .NET.

BETA is designed to be used for modeling and design as well as implementation and for this reason it appears to be suitable for modeling anticipatory systems too. The ability for modeling and design comes from powerful abstraction mechanisms. Those mechanisms include the *pattern* concept, which unifies class, method, process, exception and many more giving in this way a uniform treatment. In addition to the pattern, BETA has subpattern, virtual pattern and pattern variable. In order for an abstraction to be reused, it must have been carefully designed. BETA allows procedures as well as classes to be specialized. Syntactically this is achieved by prefixing definition of a procedure body with the name of the "*super procedure*"; i.e. in exactly the same way in which this is done for classes. A pattern *P* is declared in the following way:

```
P: Super
   (# Decl1; Decl2; …; Decln
   enter In
   do Imp1; Imp2; …; Impn
   exit Out
   #)
```

where *Super* defines a possible super-pattern (superclass) for *P*; *Decl1*, *Decl2*, …, *Decln* is a list of declaration of attributes or patterns (which means methods or classes); *In* defines possible input parameters of *P*-object; *Imp1*, *Imp2*, …, *Impn* is a sequence of imperatives associated with *P*-object; *Out* defines possible output parameters of *P*-object.

BETA supports the notion of virtual procedures as well, and as in other object oriented languages virtual procedures are bound dynamically. However, in BETA a virtual procedure cannot be redefined, but only further specialized. This means that execution will always start at the most general level and be delegated recursively to more specific ones through use of the so-called INNER statement. This feature can be used with benefit in object oriented modeling for incremental (gradual) development of the models.
Block structure allows to declare a procedure as local to any other. BETA goes further in that and procedures can be specialized in the same way as classes in other languages. Therefore, procedures can contain local virtual procedures, which can be redefined in procedure specialization.

2.1 Virtual classes

As in BETA there is no technical difference between methods (procedures) and classes, on that account virtual mechanism can be applied both on methods and classes. Virtual classes (virtual class patterns) can be used both for direct qualified virtual classes and for general parameterized classes. The former approach enables declaration of data attributes, which can be further specialized in their subclasses. The later approach is used with connection of the container classes, which means classes that contain other objects e.g. list, stack and so on. In the abstract container class there is a virtual class attribute, which is qualified by the most general class Object. This means that the abstract container class may include instances of all classes. In the subclasses of the abstract class the virtual class attribute can have further binding, which causes further restrictions of the qualification of the attribute. It enables to describe very complicated and intricate relations in the systems to be modeled for example different roles of an object.

2.2 Composition and block structure

Modeling and design require rich means for composition of objects. Composition in BETA includes whole part composition, reference composition (aggregation) and localization. The whole part composition and the reference composition (aggregation) may be combined in so called "reference to part objects", which enables further possibilities for modeling.

BETA is a block structured language. Block structure is a natural and powerful mechanism. In Simula the use of nested classes is limited by certain restrictions. BETA does not suffer from these restrictions and supports block structure of procedures, classes and blocks, which can be arbitrary textually nested. Major advantage of block structure is locality. This makes it possible to restrict the existence of an object and its description to the environment where it has meaning.

Localization is based on the possibility of nested class declaration where outer attributes may be regarded as a global attributes to inner objects attributes. This feature is challenging for exploiting in reflective simulation [5], [6], [7].

As mentioned earlier, BETA supports mechanism associated with part objects. The mechanism of part objects has been recognized as a useful mechanism in order to model the *is-part-of relation* in the field of semantic modeling and databases.

Part objects are normally instances of globally defined classes. In BETA however, locally defined part objects and objects of locally defined classes are natural extensions of the object language concept.

In modeling where wholes consist of parts, part objects are characterized by various aspects and those aspects are defined by other classes. The location mechanism and locally defined objects enhance this possibility considerably. The mechanism of part objects and combination of this and the notion of locally defined classes and objects gives a more powerful notation of part objects and using of this in modeling and design. In addition, locally defined objects and classes are only meaningful in the context of the containing object.

BETA also introduces classless declaration of object (singular object), which is useful when there is only one object of a kind. Applied to part objects it introduces the notion of locally defined part objects.

All these possibilities are shown on an example of modeling house.

```
Bedroom: (# (* class declaration *) #);

HouseA: (#
   theBedroom: @ Bedroom (#  (*  further specialization  of the class Bedroom*)  #)
                                (* locally defined object *)
   specBedroom: Bedroom(#  further specialization  of the class Bedroom #);
                                          (*locally defined class *)
   JaneBedroom, KateBedroom: @ specBedroom; (* locally defined object *)
    JimKitchen:  @specBedroom: (# (* further specialization of the specBedroom object *) #);
           #);
```

In the **HouseA** there are three kinds of bedrooms: **theBedroom** – declared as a locally defined object; **JaneBedroom** and **KateBedroom** – declared as a locally defined object using locally defined class *specBedroom*; **JimBedroom** – declared as further specialized object using locally defined class *specBedroom*; **specBedroom** – declared as a locally defined class.

2.3 Real time facilities - active objects (processes)

A BETA objects may be either active or passive, the distinction is done at declaration of an object. An active object executes its own actions defined in an associated *action-part*. An active object thus defines an independent thread. Active object may be executed concurrently or as a co-routine making it possible to model quasi- parallel processes or alternating sequential processes.
In BETA the co-routines are a well-integrated part of the language. While Simula distinguishes between semi-symmetric and symmetric co-routines, where symmetric co-routines are always scheduled explicitly through *resume* and semi-co-routine must use *detach*. The BETA constructs are simpler and more general than those of Simula. In contrast to Simula, BETA offers the possibility of including parameters when calling co-routines, which reduces the need to use globals for communication. This leads to better encapsulation and has the potential to improve program reliability.

The BETA construct *Cycle(# do Imp #)* is similar to a prefixed block in Simula, where prefixed blocks play a major role in quasi-parallel sequencing. This is not the case in BETA. BETA adds nothing to the basic principles of co-routine sequencing used in Simula. However, the technical details of co-routine sequencing in BETA are much simpler than those of Simula.

Basic operations with co-routines are *attachment* and *suspension*. There is no explicit *resume* operation, but instead the *suspension* operation is used.
Access to shared objects may be synchronized by means of semaphores. Semaphores are, however, mainly used as a primitive for defining higher–level concurrency abstractions. The Mjølner BETA System provides a library with a number of predefined concurrency abstractions corresponding to monitors and Ada-like tasks with rendezvous. These abstractions are all defined using BETA patterns and semaphores.

3 BETA's facilities for simulation

One of the main benefits of BETA is that its linguistic features and light-weight processes can be used to create high level synchronization abstractions. BETA is based on a small number of very general concepts, which makes it highly general and orthogonal. These concepts support a wealth of additional machine and application specific features through suitable libraries. These libraries in BETA support seamless integration of the built in abstractions into the whole application.

From the simulation perspective BETA supports co-routines (or multiple threads), a prerequisite for building process-oriented models. These co-routines (threads) differ in BETA from standard objects in declaration and there are also some small differences in manipulation (e.g. in storing in the lists).

There are no standard simulation facilities in BETA. As mentioned earlier there are only two basic operations for direct control of active objects – *attachment* and *suspension*. If *R* is an active object an imperative like: *R* realizes *attach* operation. Assume that *R* is currently operating active object. The imperative s*uspend* implies that *R* is detached from active state. *R* is said to be *suspended.*

These facilities led to the BETASIM high level framework for discrete event simulation [1], [2]. The framework draw from BETA's abilities for modeling and design but the usage of the

framework is restricted to queuing simulation scenarios. The other extension is limited earlier mentioned operations of *attachment* and *suspension*. Missing *resume* operation could seem to make it impossible to transfer control to other active object except for the next process in a process queue. The framework does not have means for taking control directly between processes as well as does not use the *main program* process and so on.
On the other hand if there is a possibility of describing a reality through queuing scenarios the BETASIM framework could be a good tool.

4 Simulation framework

Computing anticipatory systems apply very often simulation models and BETA appears to be an excellent tool for this purpose. Our intention was to use powerful abstraction mechanisms that provide BETA and create a simulation framework for general use. In the design we were inspired by the Simula interface for simulation as it proves its suitability for general simulation purposes for years. Our aim was to create layered design model as it offers an elegant and powerful metaphor for organizing such layers of knowledge through locality of description. It means restrict the scope of context a programmer needs to consider at any given point of time. Encapsulation ensures that objects can be insulated from a surrounding context by allowing access to internal representations only through well-defined interfaces. Simulation framework is based on a layered design where the concepts provided at one level may be employed to compose more specialized components. Creation of frameworks and layers is well supported in BETA.

The first and the most important task was to introduce some mechanism for co-routines to behave as “symmetric co-routine”. Co-routines in BETA behave like “semi-co-routines”. They are so called because there is an asymmetry between the calling co-routine and the co-routine being called. The caller explicitly names the co-routine to be called, whereas the called co-routine returns to the caller by executing *suspend* command, which does not name the caller explicitly. The *suspend* command always suspends the active co-routine and not the lexically surrounding one.

There is another kind of co-routines, called *symmetric co-routines*, which explicitly calls the co-routine to take over. It does not return to the caller by means of *suspend* command, giving a symmetric relation between the co-routines. This kind of co-routines is not directly supported by the BETA language but there is possibility to create symmetric co-routines support exploiting abstract class [3]. This abstract class must be used as a superclass for all co-routines that are taking part in the *symmetric* scheduling. In the layered design framework it belongs to the bottom layer.

The abstract class that makes it possible for “semi-co-routine” to behave as a “symmetric-co-routine” is implemented as self independent procedure *Start* in the bottom layer and method *resume* in the *Process* class. In the body of the *Start* procedure there is a loop, which can be interrupted only at the end of application or when the *main program* takes over control. It is of course possible for the *main program* to transfer control to the *next* process to continue in executing or finishing the application. The code of this procedure follows:

```
Start:
  (#  next, active: Process;
       act: Boolean;
     enter  next
   do while next <> none
```

```
        begin active : = next; next:= none;
            act : = true;
            active;   (* start running the process *)
        end;
 #)
```

When the method *resume* is called, control is transferred to the receiver of the method. In a more detailed way *this(Process)* is stored into the *next* variable and caller process is suspended. That means that the application continues in the body of the *Start* procedure (actually inside the loop). The code of the *resume* method follows:

```
Resume:
   (#
      do next : = this(Process); SUSPEND;
     #)
```

At the beginning of the application it is necessary for the introductory process to be sent to the *Start* procedure. Variable *act* indicates whether to use *Start* procedure or *resume* method for taking over between two processes.

Simulation framework also enables to transfer control to the *main program process*. It is also possible to transfer control from the *main program process* back after the suspend command. For this reason the *call* procedure was declared.

For standard behaviour the following methods *hold*, *passivate*, *wait* and *cancel* at the bottom layer were declared. All these methods behave in a Simula fashion.

In the bottom layer of the framework a set of methods for running processes is declared. The set covers direct execution (*run*), execution with some delay (*runAt*) or execution before given process (*runBefore*) and execution after a given process (*runAfter*) and so on. The framework can be extended by adding further layers or adopting current layer.

The other layers are designed for more sophisticated abstractions such as server (for client server type of communication), nested servers abstractions and so on similarly like [2].
The design simulation framework enables to describe by simulation parallel processes in a similar way as in the Simulation class in the Simula language. The simulation framework was used for experimental simulation models of the cellular systems and queuing systems.

5 General BETA's facilities using for modeling

Modeling and design facilities of BETA were used for the creation of object oriented model for the cluster analysis application [8]. In this application persistent objects were used for preserving actual state of objects for the next execution of the same or a different application. This model led to the creation of hypothetical classification structure, which was used for anticipating classification of the next examined objects. The application was tested on a real data for anticipating classification of persons in the view of possible occurrence of cardiovascular diseases. The results were used for making diagnosis for newly examined persons.
Persistent objects represent a very good means offered by BETA, which belongs to the group of persistent programming languages. Among other it enables objects to be shared by several systems. Object oriented perspective, modeling and design facilities together with persistence

brought several advantages in the process of clustering. First of all it was possible to work with clusters for all the time of application. Moreover virtual mechanism could be exploited for different methods of cluster analysis, which was very effective. Modeling and design facilities enabled to create more complex clustering structure that gave possibility not only for interactive hierarchical clustering but was used for calculation other clustering characteristics. This advancement enables to anticipate structure of data towards which the results of later cluster analysis will be approaching. The application of cluster analysis is an example of non-simulation models used by anticipatory systems.

6 Conclusions

Simulation framework and cluster analysis application that can be used in anticipation models draw from the unique BETA powerful abstraction mechanisms and thus improve their ability for better modeling and design. Encapsulated and reusable components, which can be specialized and extended through BETA's *virtual* and *inner* mechanisms, make rapid composition of models. Layered design, which was used both for the framework and the cluster analysis application makes it more comprehensible and flexible for further extension. The design is clearly arranged and enables thus not only adding new layers but also making adaptations of current layers. Simulation framework and cluster analysis application enlarge capabilities of the pure BETA language and can be further extended and structured to independent domain oriented modules using prefixes. Layered design and extended features of the programming models make them more powerful and suitable for more complex modeling and simulation purposes within anticipating models.

References

[1] Osterbye, K., Kreutzer, W.: Synchronization abstraction in the BETA programming language. In Computer Languages 25 1999 pp. 165-198

[2] Kreutzer, W., Osterbye, K.: BetaSIM A framework for discrete event modeling and simulation. In Simulation Practice and Theory 6 1998. pp. 573-599

[3] Madsen, O. L., Moller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993

[4] Madsen, O. L., Moller-Pedersen, B.: Part Objects and their Location. Technology of Object-Oriented Languages and Systems TOOLS 7, Dortmund, Prentice Hall 1992

[5] Kindler, E., Krivy, I., Tanguy, A.: Automatisation de la construction de modeles pour la simulation reflective. In Conférence Francophone de Modélisation et Simulation. 2003 Toulouse. Toulouse : SCSI , 2003. pp. 379-384

[6] Kindler, E., Krivy, I., Tanguy, A.: Object-Oriented Sytem Analysis of Anticipatory Systems in Week Sense. *International Journal of Computing Anticipatory Systems*, Vol. 14, 2004, pp. 271-285

[7] Kindler, E.: SIMULA and Super-Object-Oriented Programming. In: Owe, O., Krogdal, S., Lychne, T. (editors): *From Object-Orientation to Formal Methods* [Lecture Notes in Computer Science, vol. 2635]. Springer, Berlin, Heidelberg, New York, 2004, pp 165-182

[8] Hunka, F.: Interactive Clustering Using Object Oriented Perspective. In Conference: Information Systems Modeling. 2003 Roznov pod Radhostem ISM'02. pp. 243-250