BETA AS AGENT BASED SIMULATION LANGUAGE

Frantisek Hunka
Department of Computer Science, Institute for Sesearch and Fuzzy Modeling
University of Ostrava
30. dubna 22, Ostrava 701 03, Czech Republic
E-mail: frantisek.hunka@osu.cz

KEYWORDS
Agent based simulation, discrete simulation, BETA.

ABSTRACT

This paper describes possibilities of the BETA language and simulation class BetaSIMULATION essentially enlarging the existing BetaSIM discrete event simulation framework (Kreutzer and Osterbye 1998), (Osterbye and Kreutzer 1999), which enables the BETA language to be used as an agent based simulation language. The BetaSIM framework was designed to support queuing scenarios based on processes, in which a special attention is paid to providing abstractions that allow to specify synchronization between processes in a simulation model. The concepts used in BetaSIMULATION are inherited from Simula and enable thus extension of the original BetaSIM framework towards agent based simulation. Specific intention of BetaSIMULATION is to exploit BetaSIM for more complex simulation tasks such as reflective simulation, described in (Kindler et al. 2003), (Kindler at al. 2004), using in this way many BETA's unique features.

INTRODUCTION

Simulation is used in a wide range of applications today. Simulation framework can provide not only the basic simulation tools but also certain abstraction level for describing simulated reality. In addition, a simulation framework can be also divided into modules or in the view of abstraction into layers that provide certain services or abstraction mechanisms and can be chosen deliberately in regarding the reality. The better modeling and simulation facilities the underlying programming system provides the better potential for simulation framework creation can be used.

The rest of the paper is structured as follows. In the next section we describe specific features of the BETA language in the context of modeling and simulation facilities. The second section is applied to the short description of the BetaSIM simulation framework. The third section contains description of BetaSIMULATION that essentially extends the abilities of BetaSIM simulation framework towards agent based simulation. The forth section is focused on the structure of BetaSIMULATION. In the last section we summarize the results and give some conclusions.

SPECIFIC FEATURES OF THE BETA LANGUAGE

BETA can be characterized as object-oriented, block-oriented and process oriented language that comes from the Scandinavian school of object-orientation. BETA is among others characterized by powerful abstraction mechanisms. This includes the pattern concept, which e.g. unifies class and method. The notion of virtual pattern generalizes virtual procedure (method) and provides a virtual class concept that supports generic classes. 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 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. This causes that the actions (methods) can be further specialized, which can be exploit in many ways e.g. in process (agent) synchronization.

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 specialized. This means that execution will always start at the most general level and will be delegated recursively to more specific levels through use of the INNER statement. This feature can be used with benefit in object oriented modeling where created models can be fluently extended (further specialized) by the growing knowledge of the modeled reality.

BETA 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 can be exploit for nesting the whole simulation model into the other one and is used in nested and reflective simulation described in (Kindler et al. 2003), (Kindler at

Proceedings 20th European Conference on Modelling and Simulation
Wolfgang Borutzky, Alessandra Orsoni, Richard Zobel © ECMS, 2006
ISBN 0-9553018-0-7 / ISBN 0-9553018-1-5 (CD)

al. 2004). In this case the process, which bears nested simulating model can be called simulating agent.

From the simulation perspective BETA itself supports co-routines (or multiple threads), a prerequisite for building agent-oriented models. These co-routines (active objects) differ in BETA from the standard BETA's objects (passive objects) in declaration and there are also some small differences in manipulation (e.g. in storing in the lists). On the other hand BETA offers the possibility of including parameters when calling co-routine, which may be used in the construction of generators. This BETA's feature reduces the need to use globals for communication and leads to better encapsulation and has potential to improve program reliability.

A co-routine executes its own actions defined in an associated *action-part*. This means that there is no need for any special method for running an object but that simply calling the co-routine by name causes its execution. An active object thus defines an independent thread. Co-routines may be executed concurrently or as a co-routine making it possible to model quasi- parallel processes or alternating sequential processes.

In BETA 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. As BETA declares only semi-symmetric co-routines the other co-routine can not be explicitly named to take over control, which is a drawback.

The BETA's co-routines are usually managed by the construct of endless loop *Cycle(# do Imp #),* which is similar to a prefixed block in Simula, where prefixed blocks play a major role in quasi-parallel sequencing. 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 active objects are *attachment* and *suspension* in BETA. 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 classes and semaphores.

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 support seamless integration of the built-in abstractions into the whole application.

BETASIM QUEUING SCENARIOS SIMULATION FRAMEWORK

BetaSIM framework is based on a layered design where the concepts provided at one level may be employed to compose more specialized concepts at next layer. Object-oriented description offers an elegant and powerful metaphor for organizing such layers of knowledge through locality of description. The framework can be extended by adding a new layer or adopting the current layer.

It draws from the Beta's possibility of well integrated additional libraries and in this way introduces high level synchronization abstractions such as re-entrant monitors, server abstractions, interruptible servers and nested servers. The key abstraction mechanism used in creating these abstractions is procedure (method) specialization. All these synchronization issues can be further exploited in making general extension of the framework. This makes the framework more powerful, suitable for more complex modeling and simulation applications.

Each process used in the framework is declared in the more general way that covers the name of the process, numerical value of the instance counter and the basic attributes and methods necessary for queuing scenarios simulation. The class describing these processes can be further specialized in high level of synchronization abstractions.

In the BETA language there is no explicit possibility for taking over control between two co-routines. There is only *suspend* operation that stops co-routine's "life rules" and the control is moved into the "prefixed block" usually created by endless loop. From the loop the next co-routine in turn is chosen to take over. This fact limits the simulation framework BetaSIM only to queuing simulation scenarios where the process of taking over control is managed mostly by the process queue or other queue used in this simulation framework.

The framework also lacks other standard discrete event simulation facilities (known from Simula) such as *main program* process. The process queue does not provide all protocol of methods necessary for standard simulation tasks but it only fulfills queuing scenarios requirements. On the other hand BetaSIM framework offers high level of abstraction mechanisms that are based on methods inheritance, block structure and powerful abstraction mechanisms of the BETA system.

SIMULATION CLASS BETASIMULATION

BetaSIMULATION is designed to essentially enlarge exploiting of the BetaSIM simulation framework towards process and agent oriented simulation. In this way the design of the BetaSIMULATION was inspired by the Simula interface for simulation as it proves its suitability for general simulation purposes. At first it was necessary to modify some classes of the bottom layer and to add some new ones to fulfill new requirements. Most of these activities were connected with the process queue, its structure and functionality. Elements of the process queue were redesign and new methods were added.

The other changes were connected with the possibility of explicit taking over control between processes. BetaSIMULATION closely cooperates with simulation framework. During the work on BetaSIMULATION there was maximal effort to keep all possible functionality of synchronization abstraction layers and other advanced feature of the original framework.

First and the most important task was to manage processes to behave as "symmetric ones", which means introducing explicit naming next active process that takes over control from the suspended one. There is an asymmetry between the calling process and the process being called. The caller explicitly names the process to be called, whereas the called process returns to the caller by executing *suspend* command (with no parameters), which does not name the caller explicitly. The *suspend* command always suspends the active process and not the lexically surrounding one.

There is another kind of active objects, called *symmetric*, which explicitly calls the process to take over control. It does not return to the caller by means of *suspend* command, giving a symmetric relation between the processes. As was mentioned earlier symmetric processes are not implicitly supported in the BETA language. They can be only explicitly modeled using abstract superclass, described in (Madsen at al. 1993). This abstract superclass must be declared for all processes that are taking part in the *symmetric* scheduling and is a part of the most bottom layer.

The abstract superclass that makes it possible for processes to behave as a "symmetric" (explicitly call process to takes over) is implemented in the BetaSIMULATION class and in this way substantially increases simulating functionality. Except for the *resume* method, which function is obvious it was necessary to develop the *Start* method that is used at the beginning of the simulation process and each time when the simulation control is returned back from the *main program* process. In the body of the *Start* method there is a loop, which can be interrupted only at the end of application or when the *main program* process takes over control. It is of course possible for the *main program* process to transfer control to the next process to continue in executing or finishing the application. The code of the method follows:

```
Start:
 (proc enter next: instance Process;
               active: instance Process;
   do while next <> none
       begin active : = next; next:= none;
            act : = true;
            active;   (* start running the process *)
       end;
 proc)
```

The *next* and *act* attributes are declared in the scope of the whole framework to be accessible by all agents.

When the method *resume* is called, control is transferred to the receiver of the *resume* method. Mere *suspend* command suspends the active object, which is the receiver of the *resume* method. For this reason the receiver must be firstly saved into the temporary variable (*next* variable) and then *suspend* command is called. After this the control is moved into the *start* method, which executes the process stored into the temporary variable (*next*). Attribute *act* distinguishes if the *start* method is called for the first time or if it is called from the *main program* process. The code of the *resume* method follows:

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

At the beginning of the application it is necessary for the introductory agent to be sent to the *Start* method and thus starts simulation execution.

Simulation inherited from Simula also relies on *main program* agent that is called in case the control is taken over by the main program. This requirement was added into the BetaSIMULATION class as well as the *CALL* method that is used for transfer control back to the agent that called the *main program*.

In the same way as Simulation class of the Simula language declares essential process managing methods, similar methods were declared and implemented in the BetaSIMULATION class. Interface of the methods is very similar but the implementation differs as it depends on the underlying systems. The following methods *HOLD*, *PASSIVATE*, *WAIT* and *CANCEL* were declared and implemented at the bottom layer of the BetaSimulation class. In the BetaSIM framework there are *HOLD*, *PASSIVATE* and *WAIT* methods but they are not directly accessible to processes and have slightly different code just suitable for queuing simulation scenarios.

The BetaSIM framework also missed methods for direct executing agents as the simulation control was done through queues. Queuing scenarios are based on the fact

that processes take over control only by getting at the top of the (process) queue. Explicit take over control between two processes is not possible. For the bottom layer of the framework a set of methods for running agents was declared and implemented. The set covers direct execution method (*run*), execution with some delay method (*runAt*) and execution before given agent method (*runBefore*) and after a given agent method (*runAfter*).

BetaSIMULATION class also covers generation of new agents directly from agents themselves. A special class was declared and implemented for agents' creation. This class is parameterized by a type of the agent to be created and also ensures unique identification number of a new agent for tracing purposes. This class creates new agents at time with a given probability. One of the BETA's drawbacks is that the initial parameters (arguments) of the newly created agent can not be added directly as other languages allow. Instead the initial parameters can be inserted through the *Init* method, which must be called explicitly when a new agent is created.

BetaSIMULATION class extends the use of simulation framework also for the cases where it is necessary the agent itself has possibilities to create a new agent. It means that in the agent's class there is a method, which enables to create new objects with initial parameters. This can be e.g. used in cellular system simulation.

BETASIMULATION STRUCTURE

BetaSIMULATION is a Beta class that extends the simulation framework and can be used directly as a prefix for a specific application or as a prefix for further specialized layer. Using prefixes is very effective and helpful for creation of the certain level of abstraction. Of course this practice gives the possibility of choosing only those layers that are necessarily needed for the application. BetaSIMULATION class contains:

- simple attributes that hold simple values e.g. status of the processes,
- essential processes declaration,
- methods of the framework,
- classes or singular objects.

BetaSIMULATION class declares and implements three additional agents exploited for simulation itself. They are *main* agent that refers to the main program, *next* and *nextDt* agents that are used in the *resume* operation for keeping reference to the next agent that takes over. The *current* agent remains without changes and refers to the active agent.

BetaSIMULATION class internal methods can be divided into two groups. The first group is represented by those methods that implicitly operate on the *current* (active) agent. This means that the reference of the agent is not explicitly given as it is supposed that the agent is referred by the *current* variable. Example of these methods are methods *HOLD* that holds *current* agent by a given delay, *PASSIVATE* that removes *current* agent from the process queue and cancel the agent reference in the process queue, *WAIT* that puts the active agent into the waiting queue and applies *passivate* method on it.

The second group of the methods provides specific operations on an agent, whose reference must be explicitly given. Example of these methods is a method *CANCEL* that cancels any process, *Start* that starts cycle of simulation, *call* that restart simulation cycle after return from the *main* program, *run* that puts the process at the first place in the process queue, *runAt* that puts the agent at given place in the process queue and other methods for executing agents with different conditions.
BetaSIMULATION extended and modified the class describing agents themselves. The importance of the class is given by the fact that many of the other classes whose objects actively take part in the simulation process are its subclasses. Attribute part and the functional part were extended to meet the higher simulation requirements.

Earlier mentioned *resume* method is used for suspending *current* agent and taking over explicitly named agent. *NextEv* method returns reference to the very next agent in the process queue. This method enables *current* (running) agent to have reference of that agent without necessity to do it through another facility. The *firstEv* method returns a reference of the first agent in the process queue. *pCounter* is an example a singular object that provides each newly created agent with the name of its class and with a sequence number that is unique identifier. This notion is helpful in debugging and tracing operations. The *pCounter* singular object is used when there is no explicit agent generator but the agent creates next agents.

The *EventNotice* class declares elements of the process queue. Its attributes are references to the agent and simulation time for taking over control in the simulation.

BetaSIMULATION class that extends simulation framework contains additional attributes, agent declaration, methods and classes that enable extend the possibilities of the whole simulation framework. The BetaSIMULATION class was tested on experimental simulation models of the cellular systems and queuing systems.

ILLUSTRATIVE EXAMPLE

The illustrative example mainly draws from the fact that block structure, hierarchy of classes and hierarchy of methods are mutual orthogonal in the BETA language. This gives qualifications for the BetaSIMULATION

class to be used in complex simulation tasks such as nested or reflective simulation (Kindler et al. 2003). The whole process that bears nested simulating model (internal simulating model) can be considered simulating agent.

In the illustrative example we try to roughly describe simulation of the hospital working (composed of the patient and bed classes only). The simulation study is prefixed by the BetaSIMULATION class as it will exploit most of the class declaration. Next the classes of *patient* and *bed* are declared at the level suitable both for external and internal simulation model. The external simulation model represents simulation experiment while internal simulation model represents a computer used by external simulation model for a partial simulation. This type of simulation is called reflective as the internal simulation model simulates substation part of its environment (in that case external simulation model).

In the external simulation model (simulation experiment) class *patient* and *bed* are further specialized to fulfill the requirements of this part of simulation model. Inside the simulation experiment there is a simulating agent – internal simulation model (computer) that is exploited by the external simulation model for partial simulation (Kindler at al. 2004). Short description in the BETA languages follows:

```
BetaSIMULATION (#
   patient0: (# … #); bed0: (# … #); … ;
   Experiment: (# patient: patient0 (# … #);
                  bed: bed0 (# … #); …
        Computer: BetaSML(# patient: patient0 (# … #);
                     bed: bed0 (# … #);  … ;
          do <main program computer part>  #);
    do < main program Experiment part > #);
 do < main program specification   #);
```

As the simulation study consists of three parts there are also three *main programs*. The most inner belongs to the "computer" simulation, the middle belongs to the simulation experiment and the outer belongs to the simulation study. The internal simulation model "computer is prefixed by BetaSML, which is the superclass of the BetaSIMULATION class providing only necessary simulation tools that covers time axis and process queue facilities. Bearer of the internal simulation model is an agent that can react on the requests of the external simulation model.

Class identification used for patient and bed are the same both in the "experiment" and the "computer" as they are declared in a different blocks.

CONCLUSIONS

BetaSIM framework and BetaSIMULATION draw from the unique BETA features. BetaSIM framework introduces high level synchronization abstractions arranged in layered design. This design is clearly arranged and enables thus not only adding of the new layers but also making adaptations of the given layer. BetaSIMULATION that inherits from Simula enlarges capabilities of the original framework to become more general. BetaSIMULATION overcomes the drawbacks of the BETA language and BetaSIM framework by declaration and implementation of the additional structure and functionality and enables thus the framework to be used for agent based simulation. BetaSIMULATION was created with great understanding to the original framework, which means that all the sooner designed synchronization abstractions such as re-entrant monitors, server abstractions, interruptible servers and nested servers can be used together with BetaSIMULATION without changes or with small adaptation. BetaSIMULATION can be further extended and structured to independent domain oriented modules using prefixes.
Layered design and BetaSIMULATION extend features of the original framework and makes the framework more powerful and suitable for more complex modeling and simulation purposes.

ACKNOWLEDGEMENTS

The paper was supported by the research scheme of the Institute for Research and Application of Fuzzy Modeling No: MSM6198898701

REFERENCES

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.

Kindler, E., Krivy, I., Tanguy, A.: Formalized World Viewings - a Way to Nested Simulation. In Second International Industrial Simulation Conference. Malaga. Gent : Eurosis, 2004. pp. 20-24.

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

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

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

AUTHOR BIOGRAPHY

FRANTISEK HUNKA studied Cybernetics in Transport and Communication at the Zilina University, Slovakia. He received Ph.D. in modeling of water supply systems. After graduation he worked as a researcher with the Institute for Automation and Management in Industry in Ostrava for ten years. Since 1988 he has worked at University of Ostrava, Computer Science Department. Nowadays he works as associate professor there. His main research interests include object-oriented technologies including simulation and process-oriented approach.