Dynamic Model Functionality

Dorian Gorgan

August 1995
Dynamic Model Functionality

Dorian Gorgan*
Computer Science Department
Technical University of Cluj-Napoca
26, G. Baritiu St., RO-3400 Cluj-Napoca, Romania
e-mail: gorgan@utcluj.ro

Abstract
The functionality of a graphical Dynamic Model (DYMO) is presented. The model consists of active entities
called objects, variables, flags, and passive entities such as behaviors, trajectories, actions, conditions, and so on. The
active entities have an explicit defined behavior as a spatial and temporal evolution. The behavior is running on a
trajectory, and is defined by a set of parameters, a set of positions and actions which the object executes during its
evolution. The trajectory provides the possibility to direct manipulate the entities of the model and their elements. By
direct manipulation the model developer may define the actions, rules and conditions included in the evolution
associated to an entity position.

The evolution of the model consists of the parallel and concurrent evolution of its active entities. Every entity
such as object, variable, flag, complex object, behavior is managed by a set of light-weight processes called threads. A
set of light-weight processes perform the behavior, the moving, the graphical presentation and the interaction with
external modules of the active entities. The behavior execution is a succession of actions achieved by the current entity
or by delegated entities. The model entities communicate between them by messages. The identification by name of all
the entities is supported.

The model is proper in high-end simulation systems, scientific or engineering visualization systems, rapid
prototyping systems, graphical user interfaces and visual programming.

* The work have been carried out at the Rutherford Appleton Laboratory, UK, and supervised by Prof. David A.
  Duce. The work have been supported by a TEMPUS grant.
CONTENTS

1 INTRODUCTION

2 MODEL INTERFACES

3 FUNCTIONALITY PRINCIPLES

4 COOPERATING PARALLEL PROCESSES
   4.1 Distributed Object Based Programming Systems
      4.1.1 Object Structure in DOBPS
      4.1.2 Object Management
   4.2 Tools for Describing Systems of Cooperating Processes
   4.3 Light-weight Processes. Threads

5 MODEL ENTITIES
   5.1 Objects
   5.2 Variables and Flags
   5.3 Behaviors
   5.4 Trajectories
   5.5 Rules
   5.6 Conditions and Expressions
   5.7 Actions
      5.7.1 Action Modelling
         5.7.1.1 Create Action
         5.7.1.2 Delete Action
         5.7.1.3 Instantiate Action
         5.7.1.4 Append Action
         5.7.1.5 Get and Put Actions
         5.7.1.6 Activate Action
         5.7.1.7 Show Action
         5.7.1.8 Jump Action
         5.7.1.9 Rotate, Translate and Scale Actions

6 COMMUNICATION MECHANISM
   6.1 Communication with messages
   6.2 Message Format

7 MODEL PROCESSORS
   7.1 Behavior Processor
   7.2 Server Processor
   7.3 Display Processor
   7.4 Interaction Processors
   7.5 Function Synchronization

8 MODEL EVOLUTION
   8.1 Condition Evaluation
   8.2 Action Execution
      8.2.1 Create Action Execution
      8.2.2 Delete Action Execution
      8.2.3 Instantiate Action Execution

9 DIALOGUE MODELLING
   9.1 Interface Language
   9.2 User Interface Environments
   9.3 Internal Dialogue. Interactors

10 CONCLUSIONS

11 REFERENCES
1 INTRODUCTION

The static graphical model are not proper for a dynamical application in the field of simulation, computer aided design or graphical animation. The graphical model with an intrinsic behavior description for the active objects would be more flexible and more useful in the application programs. A dynamic model [14] reduces the intern dialogue between application program and the interface objects. Using the dynamic model, the application program has a user oriented control and is more closed to the natural mean of application entities manipulation. In dynamic model the logic of the program is distributed among the program entities, and the model behavior is a combination of the behavior of its parts. This concept provides a simple description of the parallel actions and of a parallel evolution.

Graphical user interfaces use direct manipulation on the interface entities. Graphical models must provide the possibility to manipulate the abstract entities such as: velocity, time, future events, parallel actions, relation between active objects, behavior synchronization, and so on. In visual programming the graphics must provide visual presentations to a better understanding of the links between abstract objects manipulated in applications. Therefore the trajectory is a space-temporal evolution which has a presentation that can be manipulated.

The main purpose of the DYMO model emphasizes on the following experiments further on highlighted in this paper:

• a flexible model that allows visual programming, active object behavior definition by direct manipulation;
• to establish a consistent set of entities that support an active and dynamic evolution: active and passive entities;
• a simple and a consistent structure for active objects, variables, flags, behavior, trajectories, and so on, to support a dynamic functionality, unique identification, interactive applications, concurrent evolution and an abstract space manipulation;
• a set of light-weight processes such as: behavior, server, display and interactions;
• a flexible, dynamic and interactive behavior of active entities;
• to establish the definition, the structure, the consistent modelling of notion such as: behavior, trajectory, trajectory position, rule, condition, expression, action, operation, action parameters;
• to define and execute a space-temporal behavior using the notion of trajectory in space and in time;
• to provide a graphical presentation and a possibility of direct manipulation of the abstract notions;
• a simple set of action that allows complex behaviors and various type of applications;
• a concurrent execution in a quasi-parallel manner using light-weight processes;
• to synchronize the action execution and the access to shared resources: data structures, model entities, light-weight processes;
• action execution by delegation;

DYMO has been experimented in C++, under Unix. The basis of communication in DYMO are the thread functions from the light-weight library of SunOS.

Figure 1. The dynamic model and modules it interacts with.
2 MODEL INTERFACES

The model is a set of active objects which have a asynchronous behavior. The application developer, called with the general term of user, and using a direct manipulation approach constructs the dynamic model. The model developer creates, deletes and instantiates model entities, and defines their behavior. The entity behavior is a set of conditioned actions. All the time the system is running the active objects execute their behavior. The conditions fire the actions according to the actual state of the model: object attributes, positions, values of the variables or flags, and so on. The state of the model can be changed by the evolution of the model itself or by the external events. The external events update the model state through the interfaces between the model and user, and graphical systems, and applications (Figure 1). The application program modifies the model state as a consequence of the application algorithm or as a result of a real process supervising.

Interface events are manipulated by interactors [16], [9]. The interactors are special object agents which supervise signals to/from external modules and send/receive messages to/from dynamic model. We define an event as a changing of a signal value. The interactors transform an asynchronous communication between the model and its external modules into a communication by messages inside the model.

3 FUNCTIONALITY PRINCIPLES

DYMO has a structure and a functionality based on the following principles:

• the fundamental entities of the model are active objects;
• the object has an associated behavior;
• the object behavior, actions, presentation, and motion are executed by light-weight processes;
• the model entities are encapsulated objects which communicate through asynchronous messages;
• all model elements may be direct manipulated and anyone has a name as an unique identification;
• all of the model entities and their elements such as attributes, parameters, components, behavior, rules, positions, etc., may be dynamically changed by the model objects or by the external modules;
• the model has an ordered behavior defined in time and space.

The model has an intrinsic defined evolution. All objects, variables and flags are active entities which have private behaviors. The model developer defines the behavior for every entity rather than defining a global scenarios as an unique controlling program. Any object may be connected to a specific behavior, and can inherit, change or delete own behavior or the behavior of another object.

The evolution of the model consists of the parallel and concurrent evolution of its active entities. Every entity such as object, variable, flag and complex object is managed by a set of light-weight processes called threads.

The evolution of the model is according to the current state and the behavior definition of its objects. The object behavior is a sequence of actions performed by the object itself or by other delegated objects. The actions operate on input elements and generate output elements. The operated elements may be object attributes, object behaviors, the value of a variable, the position from a trajectory, actions associated to a position, and so on. The access to the operated elements is authorized only through the server processor of the accessed entity. The reason for this restriction has been imposed by the synchronization between object behaviors, the multiple access to common resources, and the flexibility of a distributed implementation and execution of the model. The principle of server process delegation is over the object encapsulation principle used in object oriented programming technique.

Communication between objects use asynchronous messages. If an object needs to execute an action on another object or entity it delegates the second to execute this action. The delegation are based on message communication. The same time the message manipulation is according to private evolution.

The identification of model entities is done by an unique name versus the solution of the identification by pointer. This solution has the disadvantage of a slow execution. This dislike can be passed over with a performant machine such as with a parallel or distributed execution of the program. The advantage are more significant and consist of a flexible implementation and execution on distributed resource environment, a portability of application programs, an easy reconfiguration of the connections in the dynamical model and with its external modules, a possibility to change the address and the structure of an entity without searching and changing all references to it. Another main advantage is the possibility to refer to them in user interface, in the object behavior definition, as arguments of actions, and so on.

An object modifies the object attributes, the behavior parameters, and generally the model state when the objects are running their evolution. The changes are dynamic, and other objects may modify the state of the current object - presentation, position, name, behavior, etc.

The behavior is defined in an abstract space where the model will be run. The evolution follows an explicit defined trajectory in the abstract space. The trajectory is a set of successive positions with associated rules. The rules of a position are performed when the object there is on the specified position. The spatial definition provides a natural
manipulation of the abstract notions such as rules, future positions and states, interrelations between elements of entities, actions, and so on. The spacial definition supplies the model with the indispensable characteristics in applications such as graphical user interfaces, visual programming, multimedia presentation, virtual reality, rapid prototyping, and so on.

4 COOPERATING PARALLEL PROCESSES

4.1 Distributed Object Based Programming Systems

Definition of an object, given in [3] is as an entity that encapsulates some private state information or data, a set of associated operations or procedures that manipulate the data, and possibly a thread of control so that collectively they can be treated as a single unit.

The object can be accessed or modified only by making a request or an invocation of a publicly accessible operations of the object. Objects must respect the principle of encapsulation.

There are two types of programming languages which use objects: object based languages that support objects (e.g. Ada, Modula-2), and object oriented languages that support the concept of inheritance (e.g. C++, Smalltalk). Inheritance from a single base class is referred to as single inheritance (e.g. C++, DYMO, MADE [1], HUMANOID [23], GARNET [16], UIDE [22]), whereas inheritance from multiple base classes is referred to as multiple inheritance (PREMO [7], [8], [23], [24]). Another variation of the inheritance scheme is delegation [1], [3], [8]. Delegation is a mechanism that allows an instance object to delegate the execution of an invocation request to another object. The objects can be the instances of the same class and one of them delegates the second for a servicing request the first object is unable to serve. For example the accessing to the state of the second object.

An object based programming system can be defined as a computing environment that supplies both an object based programming language and a system that supports object abstraction at the operating system level. A distributed, object based programming system (DOBPS) [3], provides the features of an object based programming system as well as a decentralized or distributed computing environment.

4.1.1 Object Structure in DOBPS

DOBPS can support different types of object dimension: large-grain, medium-grain and fine-grain. DYMO can be implemented on a DOBPS using the concept of medium-grain objects.

The relationship between the processes and the objects characterizes the composition of the objects. DYMO uses the concept of active object model rather than the passive object model. In the passive object model, the objects and processes are completely separate entities. A process is not bound or restricted to a single object. In the active object model, a few server processes are created and associated to each object to execute its invocation requests. A process is a private resource of an object for which it was created. When an object is destroyed, its processes are destroyed also. DYMO supports the active object model. When an client object makes an action invocation, a server process of the addressed object accepts the request and performs the action of the client. The active object model can be static or dynamic. DYMO supports the dynamic model type, and every object has a different number of server processes. In the dynamic type of the active object model, server processes are dynamically created for an object as required. One disadvantage of the active object model is the deadlock. Deadlock can occur if an object does not have enough server processes to perform the requested services. In dynamic active model deadlock is less a problem. Amoeba [3], [15], CHORUS [18], Eden [3] are DOBPS which use a static number of processes in the active object model. Argus [3] uses a dynamic number of processes, and Clouds [3] and Emerald use a passive object model.

4.1.2 Object Management

In distributed, object based programming systems an important function is to manage the object actions. Actions must have the following properties: serializability, atomicity and permanence. Serializability means that multiple actions that execute concurrently should be scheduled in such a way that the overall effect is as if they have were executed sequentially in some order. Atomicity means an action either successfully completes or has no effect. The permanence means the effects of an action that successfully completes is not lost, except in the event of a catastrophic failure.

According to these properties there are three types of the action managing: requests, transactions and nested transactions. DYMO as Amoeba, CHORUS and Emerald systems supports the request scheme. In this scheme, the user is responsible for deciding when to initiate the committed procedure. The main disadvantage with request scheme is that the guarantee of permanence and atomicity is lost. In DYMO the serializability is supported by the invocation of the single server processor of the associated object and of the single behavior processor of the client object. Atomicity can be improved by the concept of pre- and post-conditions imposed to every action. For example the UIDE [22] supports
An important function of a DOBPS is the synchronization of multiple actions. A mechanism of synchronization is necessary to ensure that all actions have the property of serializability and to protect the integrity of object states. There are two types of synchronization scheme: pessimistic synchronization and optimistic synchronization. The majority of systems have a pessimistic synchronization. Argus, CHORUS, Clouds, Eden, Emerald, TABS/Camelot. Amoeba supports the both schemes of synchronization. In pessimistic synchronization an action that invokes an object is temporarily suspended if it will interfere with another action that is currently being serviced by the object. The most used mechanisms are read and write locks or timestamps, semaphores and monitors. DYMO evolves the pessimistic synchronization using semaphores and monitors. In optimistic synchronization scheme an object does not signal the conflicts while invocation are being processed. The optimistic synchronization scheme allows the maximum degree of concurrence possible. Actions are never suspended. In pessimistic synchronization two actions that modify different parts of the same entity are not able to execute concurrently even when there is no problem of conflict.

The facilities like security and object reliability are not resolved in DYMO. There is the possibility to save in a file the current state of the model. Therefore when the system fails the last saved model state can be reloaded.

Another DOBPS can be mentioned such as Mach (Carnegie Mellon University) that is an operating system for uni- and multi-processors [4]. Mach provides flexible memory management and sharing, multiple threads within a single address space or task for concurrence and parallelism, and a network-transparent communication system.

### 4.2 Tools for Describing Systems of Cooperating Processes

Using the facilities of distributed object based systems let see a few tools used for describing cooperating processes. Are presented MADE environment [1], the MANIFOLD language and PREMO standard proposal [7], [8], [25], [26], related to DYMO. Another programming language can be mentioned such as COOL (Concurrent Object Oriented Language), which was designed to exploit coarse-grained parallelism at the level in shared-memory multiprocessors [2]. COOL was developed at the Stanford University.

The project MADE (Multimedia Application Development Environment) is trying to define and implement a portable object-oriented development environment for multimedia applications. The model emphasizes on two cardinal concepts: active objects and delegation. MADE does not aim at the definition of programming tools to handle video and audio, but it also identifies a set of more general objects that are necessary to develop advanced multimedia applications. For example, interaction objects for advanced user interaction facilities, time or synchronization objects for media synchronization, constraint management, and help, monitoring objects, and so on. Arbab and al. [1] define active objects as having its own virtual processor with its own thread of control. Active objects communicate with one another by sending synchronous messages which are served by the addressed object. For the callers, client objects there is only one type of message communication, named synchronous message passing. The receiver object may use two different forms of message performing methods: the requests are queued or the requests are sampled. The paradigms of message passing and of the control of received messages are well known, as we noticed, in the concurrent programming. The unusual feature of the model from MADE, are sampled messages. But this feature is not unusual in computer graphics.

MADE was implemented on Unix and MS-DOS environments, using C++ language. The general concepts of objects have been mapped onto C++ classes. However, the C++ language has been extended as a language called mC++, which serves as the programming interface of the MADE toolkit. Every object in mC++ has its own thread of control rather than in C++ concepts of thread and object are separate entities. In MADE there are four types of objects: (1) active objects, (2) mutex objects, (3) unprotected objects and (4) C++ objects. The last three types are passive objects and the first three have a class-level behavior. Instances of mC++ objects can be thought of extensions which provide additional behavior to that defined in the prototypes.

MANIFOLD language [19] is another tool for defining complex dynamic interaction techniques that are common in user interfaces. MANIFOLD is a language for describing systems of cooperating parallel processes. Processes in MANIFOLD communicate by means of buffered communication links called streams and by reacting to events raised as asynchronously by other processes. Each activated instance of a manifold definition is an independent process with its own virtual processor. A manifold processor is capable of performing a limited set of actions. This includes a set of primitive actions, plus the primary action of setting up pipelines. A pipeline is an expression defining a couple of streams, represented as a sequence of one or more groups, processors, or ports. An event handler is a labelled block of actions in manifold. Event handlers declare a preemptive change of state in a manifold that observes an event of interest. This is performed by its manifold processor which locates a proper event handler for observer event.

PREMO (Presentation Environments for Multimedia Objects) is the name given to a proposed standardization in the field of presentation of multimedia objects. PREMO is a presentation environment which extends current graphics standards (GKS, PHIGS, etc.). It is an object oriented specification and includes distribution aspects (influence of MADE, ScriptX, GainMomentum, COBRA, COSS, etc.).
The PREMO object model defines the semantics of object and non-object types, and object interactions. A PREMO system consists of a collection of components. Any component is a collection of object types and the model objects are instances of some object type. Objects are conceptually active. Between objects there are supertype/subtype relationships. Multiple supertypes are allowed. There is a set of operations that are actions applied on objects. Each operation has a signature that consists of a name, a list of parameters type and a list of result types. When an operation request is issued, a specific operation implementation is selected based on name and signature. Objects may define their operations as being synchronous, asynchronous, or sampled. Operations are "messages". An object can invoke its own operations without deadlock.

The set of foundation objects contains the following types: EnhancedData - encapsulation of the notion of data to be processed; Controller - a Finite State Machine, with operations like handleEvent; EventHandler, Producers, Porters - active entities to build up processing networks such as fully data flow or fully control flow; Aggregate types - lists, arrays, tables; Objects for synchronization.

4.3 Light-weight Processes. Threads

The functionality of active objects can be achieved by the application programmers by two means: parallelism and concurrence. The parallelism is used when the task is solved using several, physically distinct hardware processors. The concurrence is used when a task can be broken up into a set of collaborating, but functionally independent sub-tasks called processes, threads, or light-weight processes, which exchange information at well defined moments. In these terms parallelism is only a special form of concurrence. The main conceptual differences between concurrence and parallelism are the following: for parallelism the processes are expensive resources, in the case of concurrence the processes are cheap. The assumption of having cheap processes is not in the direction of hardware development rather than with the current trends in the evolution of recent software systems.

A thread is a data type representing a flow of control. The light-weight process mechanism allows threads of control to share the same address space. Threads operate more efficiently than ordinary processes (e.g. SunOS processes), because they communicate via shared memory instead of a file system. The light-weight processes are useful for simulation programs which model concurrent situations. One thread can start other threads. The threads can be scheduled within the process (via threads library facilities), or can be scheduled by the kernel. Threads execute independently. There is no general way to predict (e.g. SunOS) how the execution of instructions by different threads is interleaved. On machines with more than one processor, more than one thread in a process executes simultaneously. Threads make operating system requests independently. Threads can manipulate asynchronous I/O events. Threads exist independently of virtual memory, I/O resources, resource allocation, and other operating system supported objects, but are able to work with these objects.

5 MODEL ENTITIES

The dynamic model is a set of components which define the structure and the functionality of the model. The structure of the model is defined by the structure of its entities and the dynamic connections between them. The functionality is defined by the private functionality of entities. The entities of the model are: objects, variables, flags, behaviors, trajectories, rules, conditions, and actions.

5.1 Objects

Objects are active entities with a private behavior. The evolution implies the execution of an associated behavior on a specific trajectory with a well defined state and parameters.

The formal structure of an object is the following (Figure 2):

```
object (OBJECT
  name,
  position,
  parent_object,
  son_objects,
  attributes,
  active_flag,
  behavior_name,
```
processors (behavior, server, display, interactors)
)

Figure 2. The object structure. An object has a behaviour with a trajectory and rules associated to trajectory positions.

The structure type is called OBJECT. The object name is unique defined. Every object may be a prototype for an instance object which has a different name and a private position. The position denotes the current position in the virtual space. When the object is moving the position is changing according to its evolution on the trajectory. An object having an associated behavior executes its evolution only if the active_flag is true. The visibility_flag specifies the object is visible or not during its evolution.

An object can be simple object or a complex object called aggregate. For simplicity we will use for both types the same common name object. The word object denotes a model entity having the above-mentioned structure rather than the notion of object from the point of view of programming language. The object which includes the current object is named its parent_object or parent. The included objects are called son_objects. The relations between parent and included objects are active when the model is running. An object inherits from its parent object all the characteristics: attributes, active_flag, visibility_flag, behavior, if it has not another explicit definition for them. An explicit element value has a higher priority then the inherited one.

The attributes are graphical attributes or application attributes. Graphical attributes can be for example, the color, line pattern, dimensions, orientation, etc., and the application attributes can be elements used by the application program such as weight, age, temperature, nominal current, etc. An attribute is defined by three elements: name, value type and value. The type specifies the type of attribute value: integer, float or string. Any attribute and its elements may be inquired or modified.

The execution of the model is a combination of the evolution of its elements. The evolution of an object is performed by its associated processors. The managers of the object are light-weight processes called threads. The processors share the same memory and have access to the model resources. An object has four processor types: behavior, server, display and interactor. The behavior processor performs the private object behavior. The server processor performs the actions requested by other objects to the owner object. The display processor creates the presentation (sound, graphics, images) of the owner objects. The interactor processors perform the interface between the object and the external modules like user, application program, or graphical systems. Interactors access the object state according to the value of the interface signals.

Variables and Flags

Variables and flags are active entities with a private evolution. The evolution implies the evaluation of an expression or a condition. The evaluation may be dynamically or statically performed. A dynamic evaluation means a permanent evaluation and a static evaluation means that the evaluation is performed at the explicitly specified moments. The value of a variable can be any value of a model element (integer, float, string), and the value of a flag is a boolean value: true or false.

The formal structure of a variable and of a flag is the following (Figure 3):

\[
\text{variable/ flag ( VARIABLE/ FLAG} \\
\text{ name,} \\
\text{ position,} \\
\text{ type,} \\
\text{ value,} \\
\text{ expression/ condition,} \\
\text{ processors (} \\
\text{ behavior,} \\
\text{ server,} \\
\text{ display,} \\
\text{ interactors} \\
\text{ )} \\
\text{ )}
\]

The structure type for a variable is called VARIABLE and for a flag is called FLAG. There are the same processors as within the object structure, but the definitions of processors are different. For example the behavior of a variable only evaluates the expression. The processor delegates other entities only to supply information about their state. The server processor performs get and put operations indeed.

The variables and flags have positions where they can be presented by the system and direct manipulated by the users.

An improved variation of active model can use a single entity type for variables and flags. The boolean value of the flag may be processed as an integer value.

Behaviors

Every active object has an associated behavior. A behavior is a passive object from the point of view of programming technique rather than an active object from the point of view of active model. The same behavior may be associated to a single object or to a set of object (Figure 4). A behavior has a name as an unique identifier. Its structure contains elements for direct manipulation: name, and position into an abstract coordinates system. The behavior is
characterized by parameters such as the number of moving steps among two explicit positions on the trajectory, other application parameters (delay time, blinking rate, etc.), and a set of trajectory position with associated rules. The behavior may be operated as a model entity, e.g. created, destroyed, associated to an object, modified and so on.

The formal definition of a behavior having the type BEHAVIOR, is the following:

```plaintext
behavior ( BEHAVIOR
    name,
    type,
    steps_number,
    trajectory_set,
    parameters
)
```

The behavior can be linked to different sets of trajectory. Two behaviors having the same trajectory set may differ by behavior parameters. The type of behavior specifies the accessing sequence of the trajectory set, such as polyline, cycle, point, inertial, random, and so on. For example, for cycle type the last trajectory position is infinitely followed by the first explicit position. The same sequence of trajectory positions may be followed in different manners, e.g. the space between successive positions may be followed in a number of steps specified by the parameter steps_number and with a specified delay.

5.4 Trajectories

The trajectory defines the evolution into the abstract space of an associated object. The main goal of trajectory is to provide a concrete representation of abstract notions. The incorporated positions allow a direct manipulation of abstract notions. Using the space representation the notion of successive moments of time are indirect represented as successive explicit positions. The abstract notion of time, as in real world, becomes a notion of space and a notion of the evolution in space. Consequently, the synchronization between objects in different moments of their evolution is equalized with the relations between objects on specified positions.

A formal definition of trajectory is as following:

```plaintext
trajectory ( TRAJECTORY
    position_name,
    position,
    rule
)
```
Every trajectory is a passive object, from the point of view of programming technique. It is characterized by a name, position in abstract space, and a set of rules. The space may be a 2D or 3D coordinates system. While the model is running and the associated object there is on an explicit position of the trajectory the rules associated to this position are evaluated. A position may have none, or a set of rule. Every rule incorporates conditions and associated actions (Figure 5). The rules are evaluated in the explicit defined sequence. The rules evaluating means all set of associated actions fire if the condition is true. After all of rules associated to the current position have been processed the current object starts to the next explicit position on the trajectory. The space between two explicit position is passed on the number of steps that characterized the behavior. Therefore an object may accept requests as delegated actions on any position from this evolution, but performs its private rules only on the explicit position from its trajectory.

5.5 Rules

DYMO is a rule based model. A rule is a named pair of a condition and a set of actions (Figure 6):

```plaintext
rule ( RULE
    name,
    condition,
    action
)
```
In the clausal form it has the following definition: \( \text{condition} \rightarrow \text{action}_1, \ldots, \text{action}_n \). The condition is evaluated and if it is true the actions are performed in a sequential order. Being a named entity the rule can be addressed in a user interface environment.

![Diagram of rule, condition, and actions relationship]

Figure 6. The relation between rules, conditions and actions.

### 5.6 Conditions and Expressions

Conditions are logical expressions which reflect the state of the dynamic model. A condition may be defined in terms of objects, variables, behavior parameters, positions, etc., and their elements. For example, a condition may have the form:

\[(\text{object("R2")},\text{attribute("crt")})+(\text{object("D1")},\text{attribute("current")})<\text{object("R15"),attribute("current")})\]

While the model is running the objects R2, D1 and R15 are inquired about the values of their attributes. The object which performs its behavior computes the condition with the values returned from the delegated entities.

The terms of conditions are elements of the model entities which have name, position and graphical presentation. Therefore the entire syntactic form of conditions may be built using the direct manipulation technique. The condition modelling has pre-fixed Polish form:

\[
\text{<unary_operator> <term> | <binary_operator> <term> <term> | <log_expression> | <str_op> <string_type> <string_type> | <aritm_expr> | <aritm_op> <aritm_expr> <aritm_expr> | <aritm_term> | <integer_type_term> | <float_type_term> | NOT_OP}
\]

where terms may have only three types: integer, float and string. There are three logic operators: not, and, or. The following relational operators: \(< > = <= >= !=\) and the following arithmetic operators: \(* / + -\) are used.

The following simple determine grammar rules have been used to build and to interpret the conditions and the expressions from the actual experimented model:

\[
\text{<condition>} ::= \text{<not_op> <condition>} \\
| \text{<log_op> <condition> <condition> } \\
| \text{<log_expression>}
\]

\[
\text{<log_expression>} ::= \text{<rel_op> <aritm_expr> <aritm_expr> } \\
| \text{<str_op> <string_type> <string_type> } \\
| \text{<aritm_expr>}
\]

\[
\text{<aritm_expr>} ::= \text{<aritm_op> <aritm_expr> <aritm_expr> } \\
| \text{<aritm_term>}
\]

\[
\text{<aritm_term>} ::= \text{<integer_type_term> | <float_type_term>}
\]

\[
\text{<not_op>} ::= \text{NOT_OP}
\]
5.7 Actions

In [14] there are defined the following set of 12 actions used in the dynamic model: create, delete, instantiate, append, get, put, activate, show, jump, rotate, translate, and scale. The first four actions control the creation and destruction of model entities. Get and put actions inquire and modify the content of the model entities. The activate and show actions control the motion and the visibility of the model entities. The action jump controls the sequence of trajectory positions. There are also three graphical actions such as rotate, translate and scale which control the graphical attributes of objects and their presentation.

5.7.1 Action Modelling

Every action is an object, from the point of view of programming technique, having the following general form:

```
action ( ACTION
  operator,
  input_entities,
  output_entities,
  error
)
```

The operator acts on the input_entities and provides the output_entities. Input and output entities are defined in terms of model entities and their elements. For the reason of a pessimistic synchronization and for respecting the property of serializability, any operation on the model elements must be performed only by delegation. An action is defined in BNF formal language as followings:

```
<action> ::= <create_act> | <delete_act | <instantiate_act | <append_act
  | <get_act | <put_act |
  | <activate_act | <show_act |
  | <jump_act |
  | <rotate_act | <translate_act | <scale_act
```

5.7.1.1 Create Action

The create action starts from a nil entity and create a new entity with a private name, type, and position. The create action operates on the following entities: (a) objects, (b) behaviors, (c) variables, and (d) flags. The formal definition of the create action modelling is such as:

```
<create_act> ::= <create_obj> | <create_bhv> | <create_var> | <create_flag>
<create_obj> ::= OP_CREATE EN_OBJECT <name> OBJ_POSITION <x> <y>
<create_bhv> ::= OP_CREATE EN_BEHAVIOR <name> BH_TYPE <trj_type>
<create_var> ::= OP_CREATE EN_VARIABLE <name> VAR_TYPE <var_type>
  VAR_POSITION <x> <y>
<create_flag> ::= OP_CREATE EN_FLAG <name> FLG_TYPE <var_type>
  FLG_POSITION <x> <y>
<trj_type> ::= TRJ_INHERITED | TRJ_POLYLINE | TRJ_POINT | TRJ_CYCLIC
  | TRJ_INERTIAL | TRJ_RANDOM | TRJ_GRAPH
<var_type> ::= VAR_DYNAMIC | VAR_STATIC | VAR_APP
<flag_type> ::= FLG_DYNAMIC | FLG_STATIC | FLG_APP
```
The create action is performed by the current object itself on the specified entity. The entity creation is performed without delegation since it not violate the encapsulation object concept.

5.7.1.2 Delete Action

The delete action destroys an yet created entity or its elements. The delete action operates on the following entities: (a) objects, (b) behaviors, (c) variables, (d) flags and (e) their elements.

The formal definition of the delete action modelling is the following:

```
<delete_act> ::= <delete_obj> | <delete_bhv> | <delete_var> | <delete_flag>
<delete_obj> ::= OP_DELETE EN_OBJECT <name>
                | OP_DELETE EN_OBJECT <name> <obj_element>
<delete_bhv>  ::= OP_DELETE EN_BEHAVIOR <name>
                | OP_DELETE EN_BEHAVIOR <name> <bhv_element>
<delete_var> ::= OP_DELETE EN_VARIABLE <name>
<delete_flag> ::= OP_DELETE EN_FLAG <name>
<obj_element> ::= OBJ_BEHAVIOR | OBJ_SONS <name> | OBJ_ATTRIBUTE <name>
<bhv_element> ::= BH_TRAJECTORY
                 | BH_POSITION <name>
                 | BH_PARAMETER <name>
                 | BH_POSITION <name> BH_RULE <name>
                 | BH_POSITION <name> BH_RULE
```

The delete action may be performed by the entity itself if it is delegated to delete one of its elements. If the delegated delete action specifies to delete the owner active object the action is aborted. An object may delete any other entity.

5.7.1.3 Instantiate Action

The instantiate action creates a copy of an entity with new name and position. All other characteristics of the entity are keeping - attributes, components, behavior, processors, etc. The instantiate action operates on the following entities: (a) objects, (b) behaviors, (c) variables, and (d) flags.

The formal definition of the instantiate action modelling is the following:

```
<instantiate_act> ::= <inst_obj> | <inst_bhv> | <inst_var> | <inst_flag>
<inst_obj>  ::= OP_INSTANTIATE EN_OBJECT <name>
                 OBJ_NAME <name> OBJ_POSITION <x> <y>
<inst_bhv>  ::= OP_INSTANTIATE EN_BEHAVIOR <name> BH_NAME <name>
<inst_var> ::= OP_INSTANTIATE EN_VARIABLE <name>
                 VAR_NAME <name> VAR_POSITION <x> <y>
<inst_flag> ::= OP_INSTANTIATE EN_FLAG <name>
                 FLG_NAME <name> FLG_POSITION <x> <y>
<trj_type> ::= TRJ_INHERITED | TRJ_POLYLINE | TRJ_POINT | TRJ_CYCLIC
              | TRJ_INERTIAL | TRJ_RANDOM | TRJ_GRAPH
<var_type> ::= VAR_DYNAMIC | VAR_STATIC | VAR_APP
<flag_type> ::= FLG_DYNAMIC | FLG_STATIC | FLG_APP
```
The instantiate action is performed with delegation to the prototype entity.

5.7.1.4 Append Action

The **append** action adds to an existent object or behavior a new element. A large set of operation can be performed through assign action (get and put), or append action. We choose the way of assign action for the reason of homogeneous processing. When an element is assigned the processor adds the new element any way - the element there is or is not.

The formal definition:

$$<append_act> ::= <append_obj> | <append_bhv>$$

$$<append_obj> ::= OP_APPEND EN_OBJECT <name> EN_OBJECT <name>$$

$$<append_bhv> ::= OP_APPEND EN_BEHAVIOR <name> <bhv_append_2>$$

$$<bhv_append_2> ::= TRJ_POS_NAME <name> | BH_POSITION <name> TRJ_POS_NAME <name>$$

5.7.1.5 Get and Put Actions

Get and put actions read and write the value of an entity element. The formal definition is the following:

$$<get_act> ::= OP_GET <input_entity>$$

$$<put_act> ::= OP_PUT <output_entity>$$

in the user interface it can be added the **assign** action as a couple of get and put actions.

$$<assign_act> ::= OP_ASSIGN <output_entity> <input_entity>$$

The entity elements are the same for input and for output unless the **input_entity** may be a constant.

$$<input_entity> ::= <object_element> | <bhv_element> | <var_element>$$

$$<flag_element> | <app_function> | <constant>$$

in an extended model version the **input_entity** may be a condition, a logic expression or an arithmetic expression. In a particular case the expression may be a single element. In the case of assign action the type of operand must be compatible.

$$<input_entity> ::= <condition> | ...$$

$$<output_entity> ::= <object_element> | <bhv_element> | <var_element>$$

$$<flag_element> | <app_function>$$

In graphical applications the point or the pixel may be considered as an abstract object having only attributes like position and color. The elements of a point can be read (position, color) or written (color).

$$<input_entity> ::= <point_element> | ...$$

$$<output_entity> ::= <point_color> | ...$$

$$<object_element> ::= EN_OBJECT <name> <obj_elem_2>$$

$$<obj_elem_2> ::= OBJ_NAME | OBJ_PARENT | OBJ_POSITIONX | OBJ_POSITIONY$$

$$| OBJ_BEHAVIOR | OBJ_ATTRIBUTE <name>$$

$$<bhv_element> ::= EN_OBJECT <name> <bhv_elem_2>$$

$$<bhv_elem_2> ::= BH_NAME | BH_TYPE | BH_STEPS | BH_TRAJECTORY$$

$$| BH_PARAMETER <name>$$
5.7.1.6 Activate Action

The activate action modifies the flag which specify the entity is engaged in evolution executing. The flag called active_flag may be set or deleted any time. When the active_flag is false the entity evolution is stopped. The necessary condition to start or to continue the entity evolution is entity has an associated behavior and its active flag is set on true. The formal definition is such as:

<active_act> ::= OP_ACTIVATE EN_OBJECT <name> <active_flag_val>
<active_flag_val> ::= TRUE | FALSE

5.7.1.7 Show Action

The show action modifies the flag which specify the entity is visible or not. The flag called visibility_flag may be set or deleted any time. The formal definition is such as:

<show_act> ::= OP_SHOW EN_OBJECT <name> <visible_flag_val>
<visible_flag_val> ::= TRUE | FALSE

The actions activate and show could be performed through assign action, considering active_flag and visibility_flag as attributes. We choose the approach with two distinct attribute for a faster processing.

5.7.1.8 Jump Action

The jump action controls the position of the object on the trajectory. The position controlling may be implicit defined by the type of trajectory or explicit through the jump action. The action may be performed by the current object requested from its evolution or from the evolution of another object. The current object performs the jump action by delegation. If this is the case, it is the responsibility of the model developer to specify a valid new position name. If the jump action refers to another object it must be resolved the potential inconsistency between the associated behavior and the next position. Implicitly the new position must be in the set of trajectory associated to the delegated object.

The formal definition allows both variants - the same and another object:

<jump_act> ::= OP_JUMP EN_OBJECT <name> BH_POSITION <name>
5.7.1.9 Rotate, Translate and Scale Actions

The actions rotate, translate and scale set or modify the graphical attributes of an object. The formal definitions are the following:

\[
\begin{align*}
\langle \text{rotate\_act} \rangle & \quad ::= \text{OP\_ROTATE EN\_OBJECT } \langle \text{name} \rangle \text{ OBJ\_ANGLE } \langle \text{angle} \rangle \\
\langle \text{translate\_act} \rangle & \quad ::= \text{OP\_TRANSLATE EN\_OBJECT } \langle \text{name} \rangle \text{ } \langle \text{translate\_elem} \rangle \\
\langle \text{translate\_elem} \rangle & \quad ::= \text{OBJ\_TRANSLATEX } \langle \text{tx} \rangle \text{ | OBJ\_TRANSLATEY } \langle \text{ty} \rangle \\
\langle \text{scale\_act} \rangle & \quad ::= \text{OP\_SCALE EN\_OBJECT } \langle \text{name} \rangle \text{ } \langle \text{scale\_elem} \rangle \\
\langle \text{scale\_elem} \rangle & \quad ::= \text{OBJ\_SCALEX } \langle \text{sx} \rangle \text{ | OBJ\_SCALEY } \langle \text{sy} \rangle
\end{align*}
\]

The parameter angle designates the angle of orientation, tx and ty denote the translation relative to the computed current position on the trajectory, and sx and sy are the graphical attributes specifying the modification of the dimension on x and y axes. All parameters of rotate, translate and scale actions are interpreted just in the graphical presentation process. In an homogeneous implementation all three actions may be performed by assign actions.

![Diagram of active entities communicating by messages](image)

Figure 7. Two active entities communicate by messages. A pair of a sender and a receiver thread controls the message passing.

6 COMMUNICATION MECHANISM

6.1 Communication with messages

Active objects change information using messages. As many concurrent environments like Amoeba, Mach, MANIFOLD, MADE, PREMO, and so on, DYMO handles asynchronous message passing.

The basis of communication in DYMO are the thread functions from the light-weight library of SunOS. DYMO has been implemented in C++, under Unix. The library contains functions like message_send and message_receive which achieve the communication protocol. To initiate a communication (Figure 7), the sender thread builds the message as a data structure and calls the message_send function. The sender specifies the following parameters for the message_send function: the receiver thread, the address and dimension of the sent message, and the address and dimension of the received message. The received message is a data structure too. The same time the receiver thread is waiting the message from a precise thread or from an unknown thread. When the message is arriving the receiver identifies the sender thread and receives the addresses and dimensions of the message data structures. All the time the receiver processes the received message and builds the answer, the sender thread is waiting the delegated
thread to reply. When the receiver thread completes its operations it discharges the connection calling a reply function from the light-weight library. From this point sender and receiver threads resume their executions.

All active entities from DYMO have associated a sender thread and a receiver thread which perform a communication. The same thread may be sender or receiver according to the model evolution.

6.2 Message Format

The messages represent the information exchanged between model entities. The message designates a command from a current active entity, and the answer from the delegated entity.

The sent message is a command specifying an action that the sender entity can not execute it and delegates another entity to perform it. The formal structure of the sent message is the following:

```
sent_message ( MESSAGE
    action address,
    sender entity name,
)
```

The action address allows the receiver entity to perform the delegated action without any other information incorporated in message. All necessary information are embodied in the structure of the specified action.

In the experimented version of DYMO the action is an object from the point of view of programming language. It contains parameters describing the action and private methods to access these parameters. The action object complies with the principle of encapsulation. This approach provide the possibility of executing different actions with various structure and content. A delegated action can be in a defined behavior or can be dynamically created. The length of the message is constant and does not depend on type and content of the action.

The received message denotes the answer after the delegated action has been completed. It is a reply from the delegated entity to the sender. The message has the following formal structure:

```
received_message ( MESSAGE
    value type,
    returned value,
)
```

The significance of information from reply message is related to the performed action. The value type gives information about the correctness of action execution. If an error has arisen the returned value is an error code. Error code provides information about the nature of action failure. In DYMO have been experimented three type of returned values: integer, floating point and string. The string type can be used as an unique solution.

The experience with DYMO has demonstrated the structure and the content of selected messages are consistent and supports knowledge for a good concurrent and distributed evolution of the active object model. Another important conclusion is the message structure is independent of the type of delegated action and answer.

7 MODEL PROCESSORS

DYMO uses the concept of active object model rather than the passive object model. In the passive object model, the active entities and processes are completely separate entities. In the active object model, a few server processes are created and associated to each active entity to execute its invocation requests. A process is a private resource of an active entity for which it was created. When an active entity is destroyed, its processes are destroyed also. The active object model can be static or dynamic. DYMO supports the dynamic model type, and every object has a fixed number of processes: behavior, server, display processes and a different number of interaction processes. A process is a light-weight process controlled by thread functions.

A better version of DYMO should create associated processes only if they are necessary. The behavior processor must be created only if the entity has associated a private behavior. When the behavior of an object is deleted its behavior process will be destroyed also. It is essential the set of thread to be reduced to increase the execution speed. The display processor should be created only for visible entities. The interaction process is created only for interaction entities. An internal entity which does not communicate with external modules: input/output devices, user interface, or application component.

7.1 Behavior Processor
Behavior process executes the behavior of an active entity. The active entity of the dynamic model are objects, variables and flags. For an active object the behavior process interprets the rules of the current explicit defined position and computes the next position.

The functionality of such a processor is as followings:

```
THREAD behavior_processor {
    repeat forever {
        if object is active {
            if current position is an explicit defined position {
                computes the next explicit position;
                mark the current position as a blocked element;
                /* executes the associated rules; */
                for every rule from a position {
                    if condition {
                        executes the set of actions;
                    }
                }
                computes the next position;
            }
        }
    }
}
```

The defined behavior consists of a set of trajectory position. These positions are called explicit defined positions or explicit positions. The explicit position may have associated rules. An object can be delegated to execute actions on any position, but only on explicit position interprets its private behavior. To interpret the associated behavior means to execute all the rules from a current explicit position. Every rule consists of a condition and a set of actions. The condition is defined in terms of elements of model entities. Therefore a condition evaluation means to interrogate model entities about their values, using get actions. The conclusion is the rule execution means a sequence of actions executed by delegated model entities.

Before starting a current position rules execution the current position data structure have to be blocked against any modification performed by the own server processor.

![Diagram of explicit positions](image)

**Figure 8.** The explicit positions (Pce and Pne), current position (Pc), and the next position (Pn) on the active entity behaviour.

While the object is active it is moving from one explicit position to the next explicit position in the sequence the positions have been defined (Figure 8). The next explicit position is decided according to the type of trajectory (polyline, cyclic, point, random, etc.), the sequence of explicit defined position, or the jump action executed on the current explicit position. The object covers the distance between two consecutive explicit positions by a set of computed intermediate positions. They are computed by a type of interpolation defined as a behavior parameter. The speed on the trajectory is evolved from a delay on every position defined as a behavior parameter also.

The variables and flags do not have trajectory positions. The behavior only consists in an expression or a condition. If the type of variable or flag is dynamic it is evaluated by the behavior processor all the time with a system dependent rate. If the entity type is static the behavior definition is evaluated only when the entity type is enquired.

### 7.2 Server Processor
Any interrogation or modification of an entity structure is achieved by the server processor. All the operations of the server are initiated by a message encoding an action. The owner or another entity delegates the server processor to achieve the action.

The functionality of the server processor is as follows:

```plaintext
THREAD server_processor {
    repeat forever {
        receive message;
        switch(delegated action) {
            case OP_GET: execute get action;
            . . .
        }
        reply to sender entity;
    }
}
```

The server may access any element of owner entity but may not modify the blocked elements. For instance is not possible to modify a trajectory position used by the behavior processor.

### 7.3 Display Processor

If the model entity is visible it has a graphical presentation. Display processor is a light-weight processor which builds the graphical presentation according to the graphical attributes of entity. Attributes like color, line pattern, fill pattern, and so on, define the current appearance on the screen. Attributes such as orientation angle, translation parameters, and scale on x and y parameters builds a shape related to the current position of the entity in the abstract coordinates space.

For the reason of homogeneous and synchronized access to the graphical attributes of the entity the display processor have to delegate the server processor rather than to directly access them. Actual experiment we have chosen the direct access approach.

In an extended version of the active object model the display processors should use the facilities of a graphical system. The graphical system can built graphical presentation in different manner for the same active model. For example, an unique already created model can be watched using various view parameters: window and viewport definitions, observer positions, observer moving, hidden objects, fotorealistic images, and so on.

### 7.4 Interaction Processors

Interaction processors are threads which can manipulate asynchronous I/O events. The state of the model can be changed by the evolution of the model itself or by the external events. The external events update the model state through the interfaces between the model and user, and graphical systems, and applications.

Entities with interaction processes models the interactors. The interactors are special object agents which supervise signals of external modules and use messages to communicate with dynamic model. When an interface signal arises specific interaction process updates the associated attribute of the owner entity. Any modification of the entity attribute is translated in a modification of an output signal.

The attributes may be interrogated by the dynamic model using the communication mechanism.

### 7.5 Function Synchronization

An important function of dynamic model is the synchronization of multiple actions. A mechanism of synchronization is necessary to ensure that all actions have the property of serializability and to protect the integrity of entity states. The main problem is to avoid race conditions [24]. There are two types of synchronization scheme: pessimistic synchronization and optimistic synchronization. The majority of systems have a pessimistic synchronization. In pessimistic synchronization an action that invokes an active entity is temporarily suspended if it will interfere with another action that is currently being serviced by the active entity. The most used mechanisms are read and write locks or timestamps, semaphores and monitors. DYMO evolves the pessimistic synchronization using semaphores and monitors. In pessimistic synchronization two actions that modify different parts of the same entity are not able to execute concurrently even when there is no problem of conflict. In optimistic synchronization scheme an object does not signal the conflicts while invocation are being processed. The optimistic synchronization scheme allows the maximum degree of concurrence possible and actions are never suspended.
The method of synchronization is according to the scheduling mechanism of light-weight processes and the nature of action achieved by them. Another important consideration is the grain of accessed data structures.

Display and interaction processes perform actions on the entity attributes and do not generate conflicts. For example it is not important if the color of an entity is modified at the current or at the next position, if the moving step is small or the displaying speed is high. Also the modification speed by model is higher than modifications through external signals, for instance a mouse moving.

If any entity has only one behavior process and one server process is necessary a synchronization of actions to the same data structures or a so called mutual exclusion (mutex). DYMO uses semaphores and monitors over the fragmented data structures. The grained data structures provide a better concurrence since it is reduced critical sections. For example, while a behavior process accesses an explicit position only this position is blocked rather than blocking entire behavior or owner active entity.

Another consideration is the nature of operation. There is not necessary a mutual exclusion between two inquiring operations. If one operation is a modification of an element this element must be blocked by the active processes and other processes has not access. We consider the better solution is to use monitors rather than mutex semaphores since unfortunately semaphores do not resolve the problem of deadlock. DYMO uses monitors to implement semaphores. The monitors approach is supported in SunOS which has a light-weight processes library.

Figure 9. An example of condition structure.

The same solution with semaphores implemented by monitors is useful to build an active objects model with a variable number of server processes for every entity. This implementation supports an entity may execute a few delegated actions the same time. When a new delegated action is requested the delegated entity creates a new server process and destroys it when the action is completed.

8 MODEL EVOLUTION

Model evolution is a quasi-parallel evolution of model active entities. The model active entities have associated behaviors which define the sequence of action executed by every entity. The entity behavior is interpreted and executed by light-weight processes. Actions are initiated by model entities and operate in a concurrent manner on the same set of model entities.
The main characteristic of the model is the state of model and the behavior of model entities may be dynamical modified by the model itself and by the external modules such as user, application, and graphical systems. An active entity can be dynamically associated with a behavior that can be dynamically modified also. The model can change the behavior parameters and the trajectory positions: positions set, position name, coordinates and position rules.

8.1 Condition Evaluation

The conditions and expressions as they have been above-stated, are deterministic grammar rules. For the reason of speed execution a condition is a data structure rather than a string of characters that have to be compiled. A condition data structure is a tree of objects (Figure 9), from the point of view of programming language. The structure of the object consists of an operator and two terms, and methods to evaluate the object as tree node. The operator may be a logical, relational, string, or arithmetic operator. The term may be another tree node such as another object indeed, or a get action. Therefore finally the condition evaluation consists in a get action execution.

![Figure 9](image9.png)

Active entity
- behaviour processor
- server processor

Behaviour definition

Message action

![Figure 10](image10.png)

Figure 10. To execute a current action the behaviour processor sends a message to the own or to another server processor.

The condition and the expression are evaluated by the behavior processor for an object, a variable or a flag. The condition evaluation gives the value for a flag entity. The expression evaluation gives the value for a variable. Conditions and expressions are components of active object behavior. The experimented model has condition and expression evaluation in rule structure only rather than an extended variant of the model with expression as action terms. For example the x coordinate of a new position in a create action could be an arithmetic expression.

8.2 Action Execution
The behavior processor executes the behavior of active entities. The consequence of behavior interpretation is a sequence of actions. For variables and flags the behavior execution consists of a sequence of get actions to evaluate conditions and expressions which define the entity values. For active objects there is a above-mentioned set of actions. All actions are executed by delegation. To delegate another entity to execute an action the first entity sends a message to the second entity, called delegated entity. As it was presented the message specifies an action object the delegated entity executes it (Figure 10).

Any action is an object from the point of view of the programming language (i.e. C++). The object obeys the principle of encapsulation. It consists of action parameters and methods to access the action structure. The formal structure of an action object is as follows:

```plaintext
object_action {
    operation,
    delegated entity,
    entity name,
    specific action parameters,
    object methods
}
```

There are two circumstances: (1) the action data structure is already defined by the model developer through user interface, and (2) the action data structure have to be built while the model evolution is running. The second situation occurs when an action invocation is a consequence of a main requested action. For instance, the instantiate action needs get and put actions. After the delegated action is completed the action data structure is destroyed.

The actions such as append, get and put, activate, show, jump, rotate, translate and scale, just change an attribute or a parameter from an entity structure. Therefore the action execution is simple, as it was already mentioned. The execution of create, delete and instantiate actions arises a few difficulties. Therefore they are presented further on.

### 8.2.1 Create Action Execution

The create action is an exception of action execution. The create action is performed by the owner behavior processor without delegation since it does not need information about other model entities. The create action structure specifies the type and the position of the entity that have to be created also.

### 8.2.2 Delete Action Execution

An active object executes the delete action by delegation. If delegates the own server processor to delete an object element. It is forbidden to delete the owner object itself. An delegated server processor may delete the owner object only if the object is not engaged into a current action.

For respecting the atomicity of the model any entity destruction must be performed only if the specified entity has not an active delegation. An active delegation means the entity has delegated another entity or it has been delegated by another. If the delegated object is engaged in an delegated action the delete action will be completed after the object finishes the engagement.

Also, to respect the consistency of the model the deleting of an entity which is a reference must be resolved as an error and the delete action must abort. For instance is not allowed to delete a behavior linked with at least one active object.

Another useful characteristic of the model would be protected entities. A protected entity is controlled by an entity attribute and specifies the entity deletion is forbidden.

In the case of complex objects the delete action is a recursive action, since have to delete all components objects. Let be an object B which has the parent object A and the component objects C1, C2,..,Cn. If an object M have to delete the object B the delete action includes the following actions:

1. object M delegates the object B to execute the delete action;
2. object B stops its behavior;
3. object B delegates object A to delete object B from its components;
4. object B delegates all components C1,..,Cn to execute the delete action;
5. object B replies to delete action message and performs the delete operation.

### 8.2.3 Instantiate Action Execution

In the case of complex objects the delete action is a recursive action, since have to delete all components objects. Let be an object B which has the parent object A and the component objects C1, C2,..,Cn. If an object M have to delete the object B the delete action includes the following actions:
Instantiate action is executed by delegation. The current active object delegates the prototype entity to create own instance, excepting the case of behavior instantiating. The instance for a behavior entity is created by an active object. Unlike the previous version of DYMO, last experimented version creates any instance as an independent entity. There is not any connection between prototype and its instance.

The instance entity retains the same structure, behavior, attributes and processors. Any case the instance entity gets new name and position. In the case of behavior entity the instance just gets a new name. Instance action is a complex action which consists of a set of assign actions (get and put actions). The get and put actions copy the characteristics of prototype to the instance entity. The prototype entity replies to the instantiate action message only when all assign actions are completed.

9 DIALOGUE MODELLING

The active model provides a solid interaction. The model contains consistent information that allow access to all entities and their elements through user interface by external dialogue. All model elements are identified by unique names and positions.

The paper just highlights these characteristics without to detail them. The DYMO experiment does not yet contains an interface implementation.

9.1 Interface Language

*Interface language* consists of symbols and rules used in the *dialogue* between active model and external modules, especially the user and model developer. The interface language is the language used to construct the *external dialogue* between model and user.

Interface language denotes also, the language which describes the model saved into an external file and which allows to reload the model into system memory. This kind language is not the purpose of this paper.

The model entities and their elements are identified by *unique names*. For example objects, variables, flags, behaviors, positions, rules, attributes, parameters, have unique names that identifies entities and their elements in interface language and internal encoding. Interface language includes *symbolic constants* like OP_CREATE, EN_OBJECT, VAR_DYNAMIC, BH_POSITION, INTEGER_VALUE, and so on, that identify entity and element types. They are used to build the deterministic structures like rules, conditions, actions, objects, attributes, and so on.

In a menu based manner and using the direct manipulation the user selects, builds, operates on model elements. The model contains consistent information that allow access to all entities and their elements through user interface by interface language.

One of the significant facility is to edit behavior rules. The editing process includes operation such as condition editing, action editing and position specification. The grammar rules that define the interface language are simple and deterministic. Therefore the syntactic forms may be built in a linear manner, from left to right side. The syntactic form is not a problem for a textual editing but it have to allow the direct manipulation approach.

Model positions are defined in a *normalized abstract coordinates space*. The interface language may use an user coordinates system and user measurement units. The user interface environment must provide the graphical presentation of the model entities. It is essential to have graphical presentation and to manipulate the set of trajectory positions. That provides the possibility to manipulate the *abstract entities* such as: velocity, time, future events, parallel actions, relation between active objects, behavior synchronization, and so on. In visual programming the interface have to provide visual presentations to a better understanding of the links between abstract notions manipulated in applications. Therefore the trajectory is a space-temporal evolution. It has a graphical presentation with elements that can be manipulated.

The direct manipulation of model entities inside the abstract space is similar with the direct manipulation of application objects inside the virtual space of application.

9.2 User Interface Environments

An active model application consists of the kernel that is an active object model linked with external modules such as user interface, graphical system, and application component. The *User Interface Environment (UIE)* should consist of the tools and the methodology to develop active model applications. UIE allows visual programming. UIE should provide operations such as:

- defining and developing model entities;
- editing by direct manipulation the behavior of model active entities;
saving and loading the active model;
* defining the links between active model and external modules;
* developing active model as an independent component of application systems;
* an incremental improvement of the model without consequences on external connected modules;
* connecting the already built active model to various graphical systems, user interfaces, and application components.

UIE emphasizes on active model features rather than User Interface Management Systems (UIDE) that emphasizes on dialogue and application component developing. UIDE provides tools for dialogue developing, application component programming, and software application improvement.

9.3 Internal Dialogue. Interactors

The active model resolves one of the most difficult problem of the software applications that is the separation between application component and interface component. That problem is known also as the dialogue independence. The active model approach provides a user oriented developing of applications where the same model contains the application entities and interface objects. Model entities are application entities and the behavior of the model evolves the application algorithm. Interactors or active objects with interaction processors implements the interface objects and their behavior. A very flexible and interactive approach is available to define the model entities behavior.

Internal dialogue has an homogeneous solution by interactors that implement the interface between model and external modules. The variables of internal dialogue between application component and dialogue component are represented by the application attributes of model active entities.

10 CONCLUSIONS

The experimented active model has proved a dynamic model implementation is reasonable on a sequential machine. Indubitable a machine with a special parallel structure offers a better but a more expensive parallelism. However a concurrent software approach provides a good solution for a quasi-parallel evolution of the active object model.

The experiment with DYMO model emphasizes the following conclusions:
* the experimented set of model entities is consistent and support an active and dynamic evolution. New experiments are necessary to establish a flexible set of interactors that support interfaces with various user interface, application components, graphical systems, under various software platforms and operating systems;
* the structures for active objects, variables, flags, behavior, trajectories, and so on, are simple and allow an homogeneous manipulation. The entities structures support a dynamic functionality, unique identification, interactive applications, concurrent evolution and abstract space manipulations. The unique identification by names allows the implementation on distributed object based programming systems;
* the light-weight processes or threads such as: behavior, server, display and interactions, prove a cheap, quasi-parallel, concurrent and reasonable fast execution;
* a flexible, dynamic and interactive behavior for active entities has been defined;
* a consistent modelling of notion such as: behavior, trajectory, trajectory position, rule, condition, expression, action, operation, and action parameters support an interactive, incremental and dynamical development of active object model.
* the model concepts emphasize the notion of trajectory which provides a space-temporal behavior definition and the possibility of direct manipulation of the abstract notions.
* has been verified a simple set of 12 actions that allows complex behaviors and various type of applications.
* action execution by delegation has been experimented on the conceptual defined model.
* the experiment has proved the active object model is a considerable solution for visual programming environments.

The extension of the DYMO experiment should consist on the following:
* to develop a direct manipulation based user interface language;
* to define and experiment a User Interface Environment that allows the developing and execution of active object model;
* to experiment the possibility of an incremental and independent development of active object models.

11 REFERENCES


[27] ERCIM Constraints management Workshop Report. Edited by David A. Duce, Rutherford Appleton Laboratory. Abingdon, United Kingdom, 24-25 April, 1995.