Buletinul Stiintific al Universitatii "Politehnica" din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.49 (63), 2004, ISSN 1224-600X

Programming Control Structures in Active Objects Model

Dorian Gorgan

Computer Science Department, Technical University of Cluj-Napoca, 15 C. Daicoviciu St, 400020 Cluj-Napoca, Romania
Phone: +40 264 401478, Fax: +40 264 194491, E-Mail: dorian.gorgan@cs.utcluj.ro, WWW: http://users.utcluj.ro/~gorgan

Abstract **– *Complex applications based on parallelism, cooperation, synchronization and distribution are quite difficult to be designed, developed, and maintained. A potential effective solution is the active objects model. The Active Objects Model (AOM) has been developed and experimented in order to provide to software development the visual programming techniques. This paper concerns on the appropriate set of programming control structures that support the visually manipulated techniques to develop complex software functionalities and interactions.***

Keywords: ***active object, visual programming, control structures, behavior, trajectory, action.***

I. INTRODUCTION

The actual programs have an evolution along the linear address space. Each high level program instruction, and its expended set of low level instructions, is located on such an address position. The program position is stored into the program counter register of the processing unit. For that reason too, the tools such as debugger, tracer and viewer can navigate along a linear trajectory of addresses. Even so, the current tools do not support efficency the case of n-dimensional parallel evolution, concurrency, cooperation, and adaptive behavior.

Consequently, the Active Objects Model (AOM) introduces the new concepts of trajectory and evolution throughout the virtual space [3]. The trajectory is the mapping of the progression around the space of application values (e.g. program addresses, states of application object, color space, animation space, attribute values space, etc).

Another difficult trend in application development and particularly in software development is the visual programming. The visual technique, based on direct manipulation, is still a difficult purpose mainly for ineffective graphical presentation and manipulation of the programming concepts such as data structures, complex entities, dynamic relationships, and control structures. Moreover, the complex applications based on parallelism, cooperation, synchronization and distribution are absolutely difficult to be designed, developed, tested and maintained.

II. ACTIVE OBJECTS MODEL

The AOM model provides the development of visual programming techniques with the virtual location associated to the model entities: active objects, static and dynamic variables, behaviors, trajectories, virtual positions, actions, rules and expressions. The active objects achieve their functionality by a set of associated processors such as behavior, server, presenter, and interactor. The model evolution and dynamism is synchronized by message based communication and bounding technique [1].

The overall research studies the AOM concepts and AOML language (Active Objects Modeling Language) in order to experiment the implementation of the AOM platform layered on the Basic Software Functionality (i.e. Operating System, Java Environment, .Net, etc.). The Basic Software Functionality (Fig. 1) provides to the AOM platform the support for communication, cooperation, evolution, distribution, parallelism, hyperstructures, visualization, navigation, visual programming techniques and animation.

The developer describes the application either as AOML program or as AOM model in the operative memory. The program describes by AOML language the specific structures and evolutions and then is loaded and executed as AOM model. The model consists of instances of active objects, behaviors, trajectories, actions, positions, rules, etc. Furthermore, the executable model can be saved in the AOML form as well. The AOM platform implements the entity structure and functionality (i.e. active objects, behaviors, positions, etc) and mechanisms for message based communication, synchronization, bounding, etc.

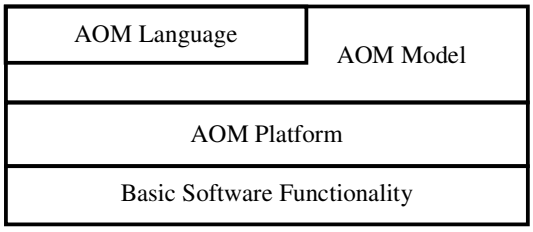

Fig. 1. AOM functional levels

III. VISUAL PROGRAMMING

A visual language manipulates visual information, supports visual interaction, and allows programming by visual expressions. The visual programming uses especially the visual expressions (i.e. graphics, drawing, animations and icons) throughout the programming process. The visual programming environments support the spatial grammar to built up a program.

The AOML language supports also the visual programming paradigms: task modeling, data flow, programming control structures, rule based programming, object orientation, and data structure definition [2].

The data flow diagram involves control structures such as repetition (*while-do*, *repeat-until*), iteration (*for-do*), branches (*if-then-else*) and procedure call. The programming control structures can be mapped onto the AOM concepts through various modes, which are to be exemplified in the next sections.

The paper mainly concerns on:

- what flow control structures can be directly manipulated in visual programming;
- what is the minimum set of the flow control structures that can support the complex and flexible functional description;
- what flow control structures should be implemented at the basic and physical level, and moreover at the high and formal language level.

IV. CONTROL STRUCTURES

The structural programming involves three fundamental control structures: *sequential*, *alternative* and *repetitive*, excluding the *call* and *jump*. In AOM the control structures may be developed at different levels of the model. For instance, the *if-then-else* structure can be created at the level of rules of one trajectory position (i.e. ETP – Explicit Trajectory Position), at the level of trajectory by some ETP positions, at the level of behavior, and at the level of active object and variable entities. The level depends on the program functionality, developer's options, and expected performance of the model execution.

A. Sequential Structure

The sequential structure consists of ordered execution of the operations. An active object advances along its trajectory through all explicit positions (excepting the case of *jump* and *call* actions) and executes the related rules and actions. The sequentiality is provided by two steps: the sequence of the ETP positions on the trajectory, and the sequence of rules at an ETP position. Therefore, a program could be described as a sequence of actions distributed over a set of ETP trajectory positions. In the following program sample the behavior Bhv is a sequence of trajectory positions ETP1-ETP4. Each position includes a sequence of rules, and each rule is a sequence of actions.

```
behavior Bhv{
  position (139,98);
  type CYCLE;
  direction FORWARD;
  steps 30;
  trajectory T1{
    position (79,45);
    trjposition ETP1{
        position (9,289);
        type UNCOND;
        sequence of rules;
    },ETP2{
        position (155,51);
        type COND;
        expression EX1;
        sequence of rules;
    },ETP3{
        position (227,277);
        type ITERATED;
        expression EX2;
        sequence of rules;
    },ETP4{
        position (348,83);
        type UNCOND;
        sequence of rules;
    };
  };
};
```

The sequentiality could be described as well as one son tree of the multi level complex active objects. Each object has just one child in the tree structure (Fig. 2). The model executes sequentially the behavior of the complex object starting from the leaf child component, follows its parent's behavior, and recursively goes up to the root of the tree, until reaches the main object's behavior.

In AOML the correspondence for *goto* is the j*ump* action, which transfers the execution control to another position on the same trajectory. To get back the control of execution the *call* action is available. The *call* action invokes the functionality describes by another local or global trajectory position and at the end returns the control to the caller position. The *call* action provides in AOM the flow control for *procedure* and *function*. To achieve that, the *call* action invokes the functionality described by an ETP position of the current trajectory, or by another behavior. After the completion of the *call* action the object continues the current trajectory.

B. Alternative Structure

The structural languages know two types of alternative instructions: *if* and *case*. In AOM the *if* functionality can be implemented both by the rule level, and by the ETP position level as well. In fact the if schema is just *if-then* rather than *if-then-else*, and is equivalent of case with two

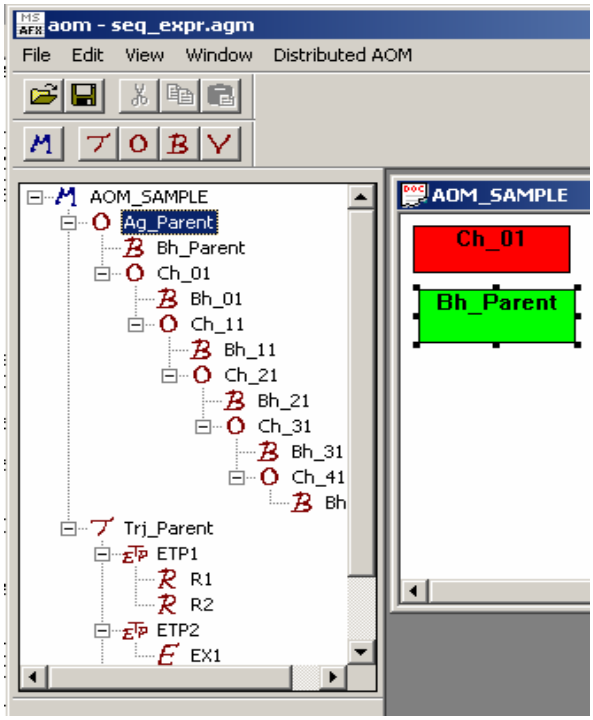

Fig. 2. Sequential flow control of the behaviors

alternatives. The first alternative (i.e. *then*) is performed for the true condition, and the second one (i.e. *else*) is performed for the negated form of the same condition. The complexity of task imposes one or another solution level.
Let us see an example through which the parameter *r* is set true in the case of parameter *n* grater than 10, and false otherwise:

```
trjposition ETP1{
  rule{
    condition{
      behavior(b1).parameter(n)>10};
    action{
      set
       behavior(b1).parameter(r),TRUE;
    };
  },
  rule{
    condition{
      behavior(b1).parameter(n)<=10};
    action{
      set
       behavior(b1).parameter(r),FALSE;
    };
  };
};
```

The *case* instruction may be implemented by a number of ETP positions equal with the number of the instruction branches. The expression of each ETP checks the equality between instruction variable and the particular value of the branch.

Example:

```
trjposition ETP1{
  type COND;
  expression {E == VALUE1};
  sequence of rules;
};
trjposition ETP2{
  type COND;
  expression {E == VALUE2};
  sequence of rules;
};
```

C. Repetitive Structure

The repetitive structure has three main forms: *while*, *repeat* and *for*. Their correspondent schemas are supported by the different basic types of the ETP structure that can be:

UNCOND – the sequence of rules embodied into the ETP is executed unconditioned. It corresponds to the sequential execution of instructions;

PRECOND – the rules are executed until the associated expression is evaluated as true. The evaluation is performed before the execution of the first rule of the sequence. It corresponds to the repetitive structure *while-do* of the structured programming;

POSTCOND – it is similar with the PRECOND type, unless the evaluation of the expression is performed after the execution of the rule sequence. It corresponds to the repetitive structure *do-until* of the structured programming;

COND – the sequence of rules is executed one times whether the associated expression is evaluated as true. It supports the implementation of the *if* and *case* schemas.

ITERATED – the evaluation of the associated expression returns an integer that specifies the number of repetitive executions of the rule sequence. This type supports the *for* schema of the structured programming.

A sample of the PRECOD type in the AOML language is as the following:

```
trjposition ETP2{
  type PRECOND;
  expression{
    trjposition(ETP2).positionx<50};
  rule{
    action{
      set agent(Button).presentation
          (PRES).graphics(G1).drawtext,
          "Ok pushed";
    };
  };
};
```

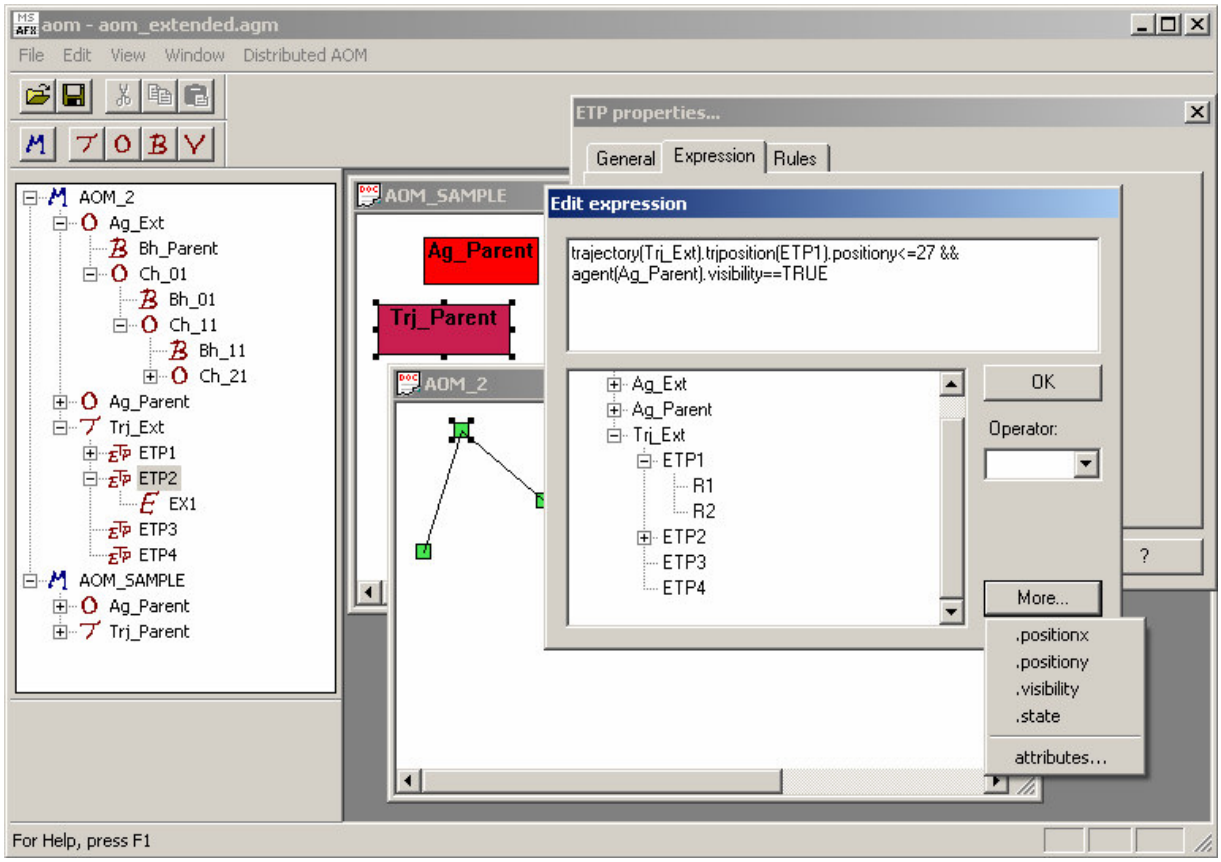

Fig. 3. AOM model development environment

V. EXPERIMENTS

The AOM model and AOML language have been developed at the Technical University of Cluj-Napoca (Fig. 3). The AOM related projects have been implemented in C++, Java and VRML and experienced the AOML language, fuzzy logics (FUZZYLOG), multithreading (DYMO), multimidia presentation (AOMIDIA), programming paradigms (PARADIGM), and visual techniques (VISTECH v1 and v2).

VI. CONCLUSIONS

Actually, the *jump* action could not be completely eliminated at the high level structures such as behavior, trajectory and ETP positions. It supports the transfer of the flow control between different functional levels and scopes (i.e. local vs. global). The decision on elimination or keeping of the *jump* action is argued by the manner in which such a schema could be directly manipulated in visual programming at the level of the basic entities.

Conceptually the *jump* could be substituted only at the rule level by a schema like *if-then-else* rather than *if-then*. Moreover, the *if-then-else* schema allows the imbricate structures. Hereby the ETP data structure is a binary tree rather than a list, where each node is a list of rules. Actually, the schema to advance along a trajectory is a simple list stored in the behavior structure and the flow control over the ETP positions is according with the *if-then* structure. The *while*, *repeat* and *for* structures are implemented only at the ETP level.

The consistency and appropriateness of the proposed set of the flow control structures should be extensively checked and argued through more practical experiments on various programming cases.

REFERENCES

[1] "Active Object Model" AOM Documentation and Projects, http://users.utcluj.ro/~gorgan/res /aom /aom.html

[2] D. Gorgan, "Visual Programming Techniques". Computer Science Education: Challenges for the New Millenium. Eds. G. van der Veer, I.A. Letia, Ed. Casa Cartii de Stiinta. pp. 129-142, 1999. http://users.utcluj.ro /~gorgan/res/webdocs/ repository/papers/ROC-C99P.zip.

[3] D. Gorgan, D.A. Duce, "The Notion of Trajectory in Graphical User Interfaces". Design, Specification and Verification of Interactive Systems '97, M.D.Harrison, J.C.Torres (eds.), SpringerWienNewYork (ISBN 3-211-83055-3), pp.257-272, 1997. http://users.utcluj.ro/~gorgan/ res/webdocs/repository/papers/dsv97.zip