SAM - An Animated 3D Programming Language

Christian Geiger, Wolfgang Mueller, Waldemar Rosenbach
C-LAB
Fuerstenallee11, 33102 Paderborn, Germany
{chris,wolfgang,bobka}@c-lab.de

Abstract

This article presents the animated visual 3D programming language SAM (Solid Agents in Motion) for parallel systems specification and animation. A SAM program is a set of interacting agents synchronously exchanging messages. The agent's behavior is specified by means of production rules with a condition and a sequence of actions each. Actions are linearly ordered and execute when matching a rule. In SAM, main syntactic objects like agents, rules, and messages are 3D. These objects can have an abstract and a concrete, solid 3D presentation. While the abstract representation is for programming and debugging, the concrete representation is for animated 3D end-user presentations. After outlining the concepts of SAM, this article gives two programming examples of 3D micro worlds and an overview of the programming environment.

1 Introduction

With the upcoming availability of 3D supporting low cost hardware and software static and dynamic 3D presentations become subject of interest for a wide range of applications. Today, 3D applications mainly focus on games, Virtual Reality, CAD, and various forms of technical illustrations. There are presently only very few applications which use 3D for other purpose like programming or specification. Examples are Cube, 3D BTTL, CAEL-3D, and Lingua Graphica [1,7,10,11]. 3D-Visulan and ToonTalk [6,12], define programs in 3D and additionally animate their execution.

In this article, we introduce our visual 3D programming language SAM (Solid Agents in Motion). SAM is a parallel, synchronous, state-oriented language for general purpose programming. SAM agents synchronously communicate by exchanging messages. Their behavior is specified by the means of production rules with a condition and a sequence of actions each. Main syntactic elements like agents, messages, and rules are 3D. Subelements are specified as text through forms. Similar to the concepts of tool tips, important textual information of an individual 3D object is shown when moving the mouse over the object. In their initial form agents, messages, and rules have an abstract semi-transparent 3D representation like spheres, cubes, cylinders, and cones. The execution of the 3D program is animated in 3D by smooth continuous motion and scaling of the individual objects. On the one hand, this allows a detailed behavioral analysis by animated execution. On the other hand, this has the advantage that the programmer has to manage only one environment and language for the specification, execution, and animation of a program. In addition, arbitrary solid 3D representations can be individually assigned to agents and messages so that it is possible to easily prototype rather complex animated 3D scenarios. The programmer can switch between abstract and concrete representations even during execution, i.e., animation.

The remainder of this article is organized as follows. The next section discusses related works in the field of 2D and 3D programming languages. Section 3 introduces the basic concepts of SAM. This covers the introduction of the basic objects, their representation, and the execution of SAM programs. Thereafter, we give two programming examples and an overview of the existing programming environment. The conclusion closes this article.

2 Related Works

For related works, we first discuss approaches in the field of 3D programming languages and visualizations. Thereafter, we consider the visual 2D programming languages Pictorial Janus and Agentsheets since SAM's concepts for combined specification, computation, and animation are partly inherited from these languages.

Lingua Graphica [11] and *CAEL-3D* [10] are both visualizations of procedural textual languages. Lingua Graphica defines a visual 3D syntax for C++ programs. It allows users to inspect and modify VR simulation code without having to leave the virtual environment. CAEL is a textual language which augments a subset of Pascal with procedures for specification of arbitrary 3D animations. CAEL-3D is the visual 3D syntax for CAEL.

3D-Visulan [12] is a 3D variant of Bitpict. In contrast to Bitpict, 3D-Visulan uses voxel elements and 3D-bitmaps. The behavior of the program is described by means of be-

fore-after rules which define how 3D-bitmaps change over time. Conditions in 3D-Visulan may have variables and can be combined by conjunction. Rules have a priority due to their location in 3D space, e.g., rules in the foreground have higher priority. A number of small applications are available like a 3D Turing-machine and a variant of the video game Space Invaders. 3D-Visulan was implemented on a Win95 PC with Visual C++ and OpenGL.

Cube [7] is a visual 3D data flow language based on a logic programming language with higher order horn clauses. Cube visualizes language elements (e.g., predicates, values) as cubes which are connected by pipes defining data flow between them. Cube has a static polymorphic type system and is type-safe. Types and values are visually distinguished by means of different colors for cubes. Values are identified by an icon above the value. Holder cubes are semi-transparent cubes which hold other values or cubes. If two holder cubes are connected by a channel their content is combined by unification. Predicates are composed of a definition cube with an icon above the predicate's name and a number of ports. Local types and predicate definitions in a cube are given on transparent 3D planes where each plane represents a horn clause. Recursive predicates like map(), n!() can be easily modeled with a plane for the recursive call and another one for termination.

Del Bimbo et al. have introduced a 3D visual language for specifying 3D *Branching Time Temporal Logic* (BTTL) [1]. The language was developed to increase the acceptance of BTTL in non-academic environments. The language basically gives abstract visual representations of BTTL formulae in 2D planes. The relationship between formulae when constructing them is given in 3D. BTTL formulae are constructed by the use of a dedicated 3D editor. The environment does not support the execution of the specified formulae.

ToonTalk [6] is a completely visual 3D programming environment and language for kids. It combines metaphors of video-games with character animation. The programmer, in form of a woman or man, freely walks or flies a helicopter through a city. The program as a set of agents is given by a set of houses. Spawning and deleting agents correspond to the building and exploding of houses. The behavior of a house is defined by a set of robots. The programmer teaches robots by demonstration after stepping into their thought bubble. The robot corresponds to a guarded rule with the guard in its thought bubble. Data come in form of boxes filled with integer or string values. Basic integer and string operations are performed when placing a value on another one. A little white mouse animates the operation by the use of a big red hammer. Communication between houses is implemented by birds. Each bird is associated with a nest. When having received a message, the birds carries that message to her nest. Due to fixed representation of the characters, ToonTalk has only limited application for other domains. The computational concepts of ToonTalk are comparable to those of Pictorial Janus which is outlined hereafter.

Pictorial Janus (PJ) is a completely visual 2D programming language [5]. A PJ program is given by a drawing where the execution is defined as the animation of the drawing. PJ is based on the concepts of the textual parallel logical programming language Janus. Basic tokens of a PJ program are graphical primitives, i.e., closed contours and connections between them. The meaning of a closed contour is independent from its geometrical representation, i.e., shape, size, color, etc. PJ distinguishes agents, functions, relations, and messages. A PJ program defines a set of agents which communicate by exchanging messages. Agents exchange messages through channels which are connected to their ports. The behavior of an agent is given by a set of rules which are located inside the agent's contour. In typical applications a rule specifies a recursive replacement of an agent.

In Repenning's *Agentsheets* [8], a designer creates a visual language by defining the look (depiction) and behavior of agents. The look is defined by the use of a depiction editor. An agent interacts on a grid with other spatially related agents. Different depictions give the different states of an agent. Unfortunately, there is not an explicit notion of a state in Agentsheets. Agents are defined in terms of graphical rewrite rules. The condition of a rule can be specified with respect to the depiction of their spatial neighbors on the grid. The next state is typically defined by discretely changing an agent's depiction which also defines a basic animation during runtime.

SAM combines several properties from the above languages. It integrates 3D program visualization with a programming environment. A programmer interacts in SAM with only one environment and language when implementing a program and its animation. A SAM program is mainly given as a set of 3D objects. Unlike some of the above 3D programming languages, SAM has no separate textual description but seamlessly integrates textual specification via forms. As soon as the program is specified its execution can be observed by animation. Derived from PJ and ToonTalk, SAM provides the animation of parallel pattern matching. End-user-oriented animation for collaborative design is due to Agentsheets and PJ.

SAM's behavioral semantics is a combination of the computation models of Agentsheets, PJ/ToonTalk, and basic concepts from synchronous languages like Esterel [4]. In contrast to Agentsheets, SAM has no spatial limitations to grid-based programs. In contrast to Agentsheets and PJ/ ToonTalk, SAM has an explicit notion of state and state transition, advanced communication facilities such as broadcast and reply, and a synchronous execution. SAM

overcomes a couple of PJ's main drawbacks [3] by introducing persistent state-oriented agents with explicit kill and spawn and, in order not to overload the graphical representation, SAM does not explicitly visualize communication channels. SAM additionally introduces actions which perform local transitions on agents graphical (and acoustic) properties.

3 SAM

SAM (Solid Agents in Motion) is a visual 3D programming language for parallel systems specification and animation. It is a synchronous parallel, state-oriented, general purpose programming language. The execution model is related to other synchronous languages like Esterel and State Charts which makes SAM also applicable for the modeling of reactive behavior [4]. A SAM program is a set of agents which communicate by exchanging messages. An agent is defined by a set of production rules which execute a sequence of actions.

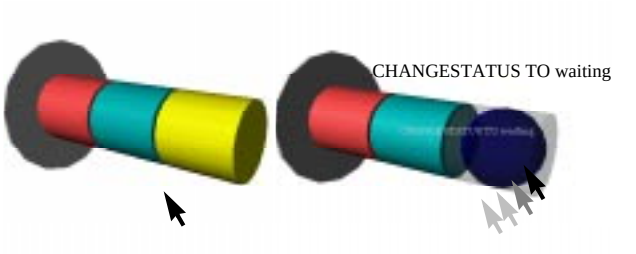


Figure 1. : Transparent Action

SAM agents and messages have an abstract and a concrete solid 3D representation. The latter one is of arbitrary form which can directly correspond to any meaning in its specific application. Specifications of messages, agents, rules, and actions are partly given as text and can be accessed via forms when double clicking on a 3D object. Comparable to the concept of tool tips, also known as bubble help, the textual part becomes visible when moving the mouse over the abstract presentation of an object. Figure 1 shows a sequence of actions as colored slices of a tube on the left. When moving the mouse over the last slice the action becomes transparent and indicates its content. In the above example the action defines a state transition (CHANGESTATUS) to 'waiting'.

In the remainder of this section, we first introduce the different SAM objects. Thereafter, we outline the execution and animation of programs.

3.1 Objects

Main SAM syntactical objects are 3D. These are messages, agents with ports, and rules with a precondition and a sequence of actions.

Agents and Messages.

Agents are 3D objects with an arbitrary number of input and output ports at their outside. An agent receives and sends messages via these ports. The different ports can be distinguished by their textual identifier and, roughly, by their different colors. In their abstract representation, agents are initially given as spheres, cones as ports at their outside, and a set of rules enclosed (see Figure 2).

An agent has an initial state. The state can be explicitly changed by a specific action when executing a rule. The abstract agent takes the color of its current state when color names are used as state identifiers.

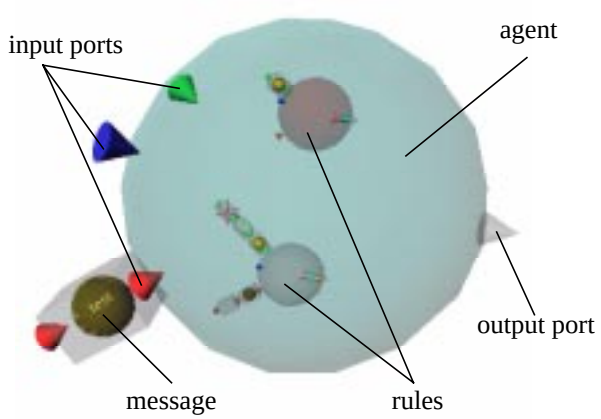


Figure 2. Abstract Representation of an Agent

Input and output ports are distinguished by the direction of the representing cone, i.e., whether it points to the attached agent or to another agent. In its abstract form, a message is given as a transparent box with two cones at its outside for getting connected to cones of other messages or agents (see Figure 3). Due to the type of its value the box encloses a sphere (string), diamond (diamond), or cylinder (arbitrary datatype). A message has its identifier, its value, its and the identifiers of its sender and receiver enclosed as text.

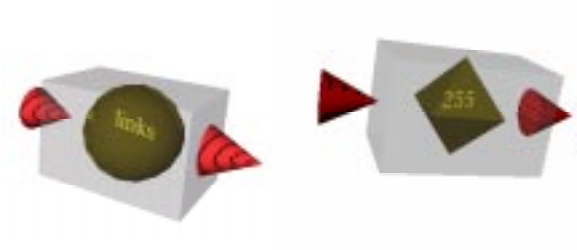


Figure 3. Messages of Type String and Integer

Rules.

Rules are located inside the agent. They have a condition and a sequence of actions. Each rule is equivalent to:

$$\underline{IF}(S, C)\ \underline{THEN}\ \{a_1; ....; a_n\},$$

where *S* defines the status condition in which state the rule applies and *C* defines a conjunction of subexpressions over message types and values at the input. A rule matches when *S* and *C* both evaluate to true. Subexpressions of condition *C* can be defined in a simple format:

$$id_1\ op_1\ id_2\ [op_2\ id_3]$$

where id_1, id_2, id_3 denote integer/string constants or message values at a specified port. Values of all scheduled messages at one port can be accessed via index. 'R[2].value', for instance, returns the value of the second message at input port R. op_1 stands for a boolean operator (>,<, >=, <=, ==, *!*=) and op_2 denotes an arithmetic operator such as +, *, -, /, for integer. For more advanced checks SAM has additional keywords specifying presence or absence of messages. With ANY a message of any type is accepted, NO means that no message is allowed at the specified port, and DONTCARE denotes that the computation does not care about the existence of a message at a given port.

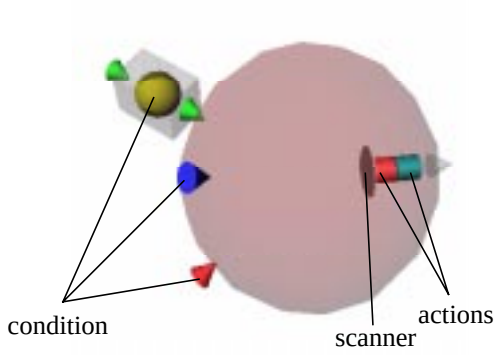


Figure 4. Rule with 2 Actions

Rules are basically visual copies of its agents with objects defining the condition at their outside and a sequence of actions inside. Actions are displayed as differently colored slices of a tube. Different colors refer to the different types of the operations.

Figure 4 gives an example of a rule with a condition and two actions. The condition is defined by one object at the upper port of the rule. The small ring inside the rule is the scanner which marks the first action in the sequence. Table 1 gives an overview of the currently supported actions with their representing color. These actions can be roughly divided into five categories:

(1) communication

(2) explicit state transition

(3) agent creation/deletion

(4) local actions

(5) import external source

Operations in the fourth and fifth category are for manipulation of the agent's graphical representation or emission of acoustic signals.

Action	Color
send message	yellow
change state	blue
duplicate agent kill agent	orange pink
move agent rotate agent scale agent color agent	red
play sound play MPEG	white

Table 1. SAM Actions

3.2 Execution

The execution of the program is based on synchronous communication. Synchronous communication means here that all agents always exchange their messages at the same time. The SAM execution cycle has two phases: (i) agent execution and (ii) communication (see Figure 5).

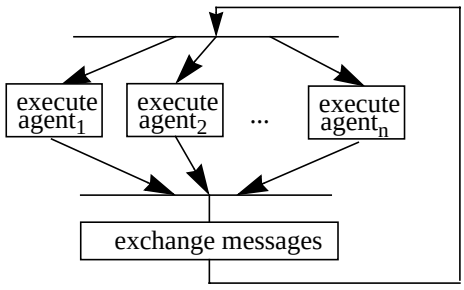


Figure 5. SAM Execution Cycle

In the first phase, all agents check the conditions of their rules in parallel. The order by which rules are elaborated is not defined in SAM. This is mainly since, due their location in 3D, any intended order of the enclosed rules is hard to determine. A rule which matches a specified state *S* and condition *C* is non-deterministically selected from the set of true rules by pattern matching. Thereafter, its actions are executed in sequential order. When executing a message generating operation the produced message is not immediately sent but kept at the output port of the agent until the beginning of the second phase. All other actions like moving, rotating, coloring, deleting, and creating agents are immediately executed.

The second phase starts when all agents have completed the execution of their actions. All messages move to their specified destination then. After their arrival the first phase triggers again the pattern matching for selecting rules, etc.

3.3 Animation

The program executes by animation with continuous smooth motions and scalings of objects. The user can observe how rules execute and determine the rough functional correctness of the program by a first visual inspection. The example in Figure 6 gives the animation of one execution cycle by 8 snapshots.

That figure shows one agent with two rules. After having received a message at the input port the animation executes in 5 steps:

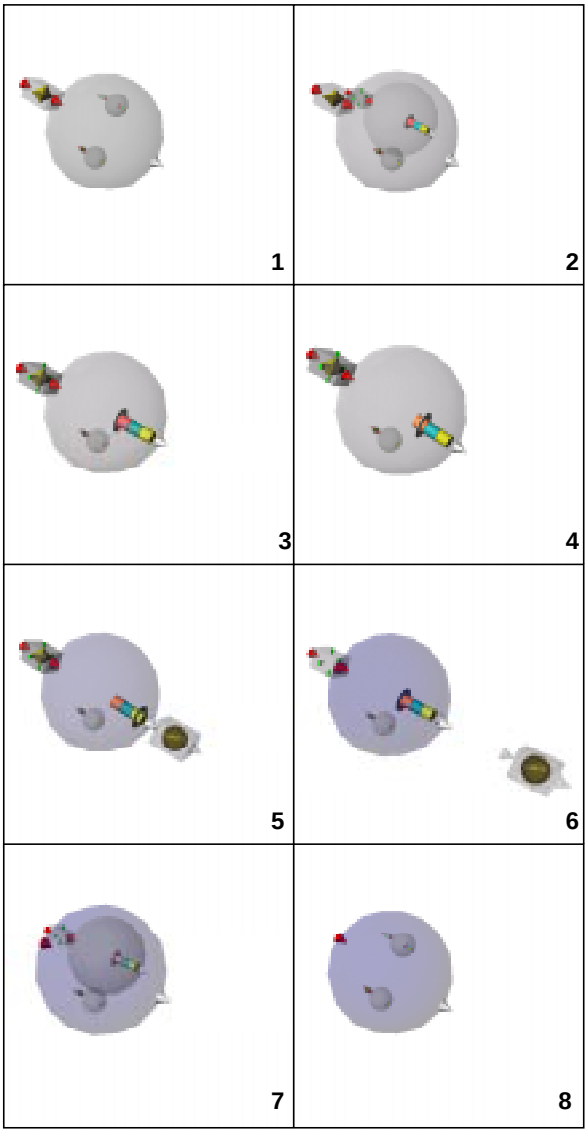


Figure 6. SAM Animation Sequence.

(1) the selected rule (rule with true condition) grows to the size of the agent until the condition overlaps the arrived messages (Frames 1-3).

(2) the scanner moves over the slices of the tube which indicates the execution of the individual actions (Frames 4-5).

(3) some actions generate messages at the output port. In our example of Figure 6 the last action generates a message in Frame 5.

(4) after having executed all actions the messages generated at the output move to their destination (Frame 6).

(5) in parallel to (4), the scanner returns to its initial position (Frame 6) and the selected rule returns to its initial size and location. Matched message(s) at the input(s) is/are consumed when rescaling the rule (Frame 7 -8)

Steps 1 - 3 are executed in the first phase of the previously defined execution cycle. Step 4 and 5 execute during the second phase when agents exchange messages.

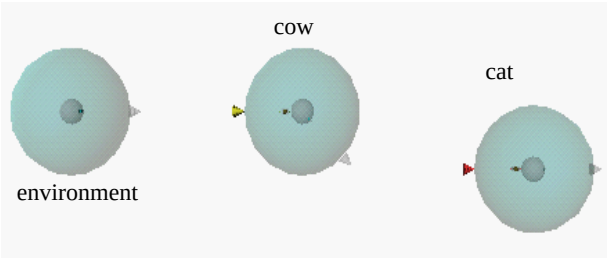
4 Examples

In this section we give two simple SAM programing examples. The first one illustrates the concepts of interacting agents producing and consuming messages. The second one gives an example for broadcast as well as the deletion and generation of agents.

4.1 Producer-Consumer

Our first SAM program models a producer-consumer scenario. Consider 3 agents in this scenario: *environment*, *cow*, and *cat*.

The *environment* agent continuously produces messages (*hay*) and sends them to the *cow*. Due to the synchronous execution model one message is produced in each execution cycle. In each cycle the *cow* also consumes one message and produces a second message (*milk)* which is sent to




```
AGENT environment
        STATUS X; PORT;
        IF (X, TRUE)
        THEN {SEND "hay" TO cow;}
END
```

```
AGENT cat
    STATUS Y; PORT Mouth;
    IF (Y,Mouth[1].type ==ANY)
    THEN {;}
END
```

```
AGENT cow
    STATUS Z; PORT Mouth;
    IF (Z,Mouth[1].type == STRING AND Mouth[1].value=="hay")
    THEN {SEND "milk" TO cat;}
END
```

Figure 7. Abstract 3D and Textual Form.

the *cat*. The *cat* simply consumes one portion of *milk* in each cycle.

The abstract visual representations of the program is shown in Figure 7. That figure also gives the program in textual format. In the textual description, we can easily identify the definition of the three agents. *Cow* and *cat* have one input port, i.e., *Mouth.* For each agent the example defines their initial states (*X, Y, Z*). The behavior of the *environment* is given by one rule which evaluates to true in each execution cycle since status always remains *X*. Consequently, in each cycle *environment* sends a message with STRING "*hay*" to *cow*. The rule of *cow* matches if the first message at the *Mouth* is of type *STRING* and of value "*hay*". Then, a message with STRING "*milk*" is produced and sent to agent *cat*. *Cat* finally consumes *ANY* message at its *Mouth* and executes an empty statement.

Figure 8 shows the corresponding concrete graphical representation of the program where the *environment* is given by the texture in the background.

Figure 8. Concrete 3D Representation of Producer-Consumer

4.2 Bees and Flowers

A little more complex example demonstrates some advanced features like broadcast, reply, and spawning and killing of agents. The example models a small micro world where a *hive* sends *bee*s to all *flower*s by broadcast. Figure 9 gives the abstract visual representation and the program of the two agents in textual format. When considering the behavior of the *flower* in its first visit the first rule applies and the *flower*

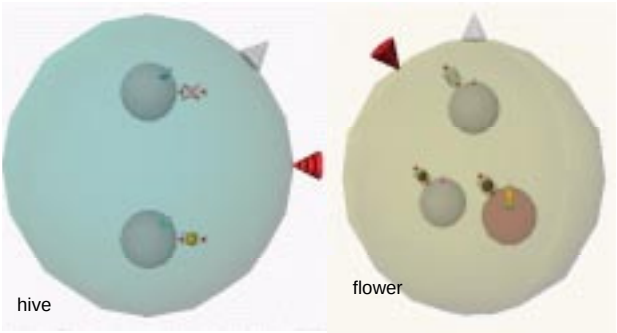
(1) replies to the sender with message "*honey*"

(2) creates an instance of type *flower* with initial state *new* at position *"Z - 80"*, where the position is determined with respect to the Z-position of the generating agent

(3) changes its state to "*visited"*

The second rule matches in the second visit when the *bee* carries STRING "*kill*". The third rule is for consuming any other messages.

Figure 10 gives the abstract (left) and concrete representation (right) of the initial program with one *hive* and five *flowers*. Note here, that the concrete model uses additional static 3D objects like a house and trees which are not part of the program. These objects are added to give the defined agents a more realistic environment.




```
AGENT hive
STATUS x; PORT in;
        IF (x, in[1].type == NOTHING)
        THEN {BROADCAST "empty" TO PORT in TYPE ANY;}
        IF (x, in[1].type == STRING AND in[1].value == "honey")
        THEN {REPLY "kill" TO in[1);}
END AGENT
```

```
AGENT flower
STATUS visited; PORT in;
        IF(new, in[1].type == STRING)
        THEN{   REPLY in[1].value = "honey";
                CPAGENT flower STATUS new POSITION Z -80;
                CHANGESTATUS TO visited;}
        IF(visited, in[1].type == STRING AND in[1].value == "kill")
        THEN {KILL (SELF);}
        IF (ANY, in[1].type == ANY) {;}
END AGENT
```

Figure 9. Abstract 3D and Textual Form.

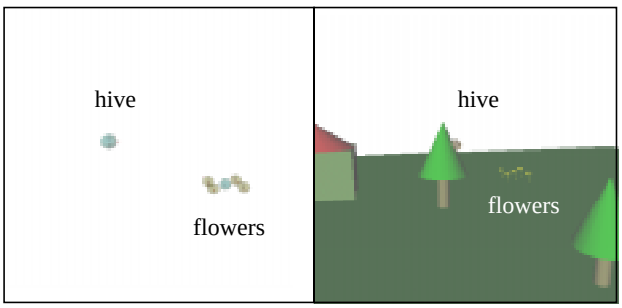


Figure 10. Abstract and Concrete 3D Representation.

Figure 11 shows additional snapshots during the execution when a *bee* visits a *flower* and two other *bee*s which return to their *hive*.

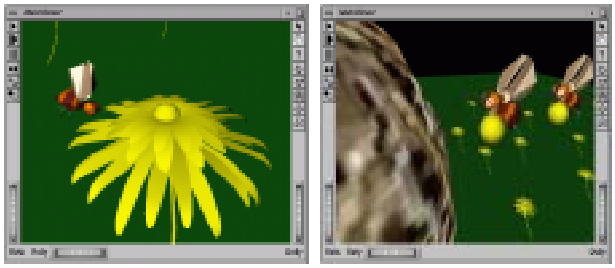


Figure 11. Snapshots of the Execution.

5 Implementation

The present system is implemented with OpenInventor and our animation library AAL (Animated Agent Layer) in C++ on SGI [2,9]. The SAM programming environment has an editor and an animator with an interpreter as depicted in Figure 12.

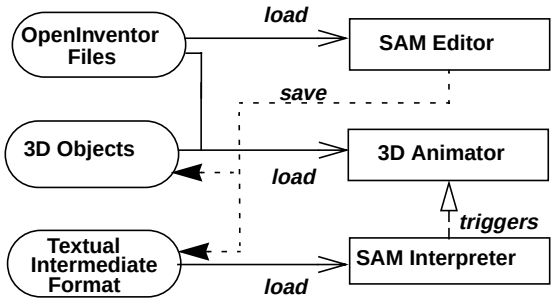


Figure 12. SAM System Architecture

The editor is a dedicated SAM graphical capture. When generating an agent, for instance, the agent has first to be specified with agent identifier, rough position, ports, and initial state in a form. Figure 13 shows the example of such a form. After entering the data the user can directly manipulate the object in its abstract representation by the means of a so-called object transformer (see Figure 14). The transformer gives the ability to move, scale, and rotate the object.

After selecting the agent the user can program a set rules by specifying position and precondition in a different form. Actions are defined in separate forms after the selection of a rule. Concrete representations can be associated to agents and messages by selecting an OpenInventor file in a list and assigning it to an object as demonstrated in Figure 15. The editor saves the created SAM program as two separate files. One file contains the program in SAM textual intermediate

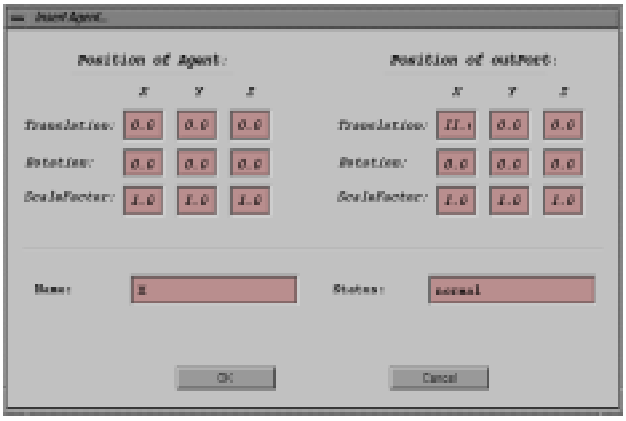


Figure 13. Agent Specification Form

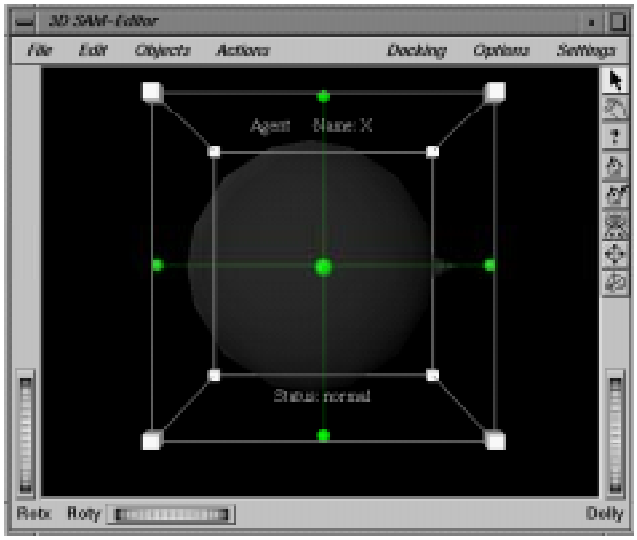


Figure 14. SAM Editor with Transformer

format which is close to the program text in Figure 7 and 9. The second file includes the 3D objects and their structure.

After the creation of the program the abstract as well as the concrete (OpenInventor) objects can be loaded into the SAM animator. The interpreter requires the program in SAM intermediate format to generate the runtime structure. 3D objects are associated to objects in the interpreter by common unique identifiers. Once being configured by the intermediate format the SAM interpreter executes the program and triggers its 3D representation by smooth continuous animation. It is possible for the animator to generate a visual 3D representation for the intermediate format when only the textual format is available.

6 Conclusions and Future Work

We have presented new concepts of the visual 3D programming language SAM. SAM is implemented as a syn-

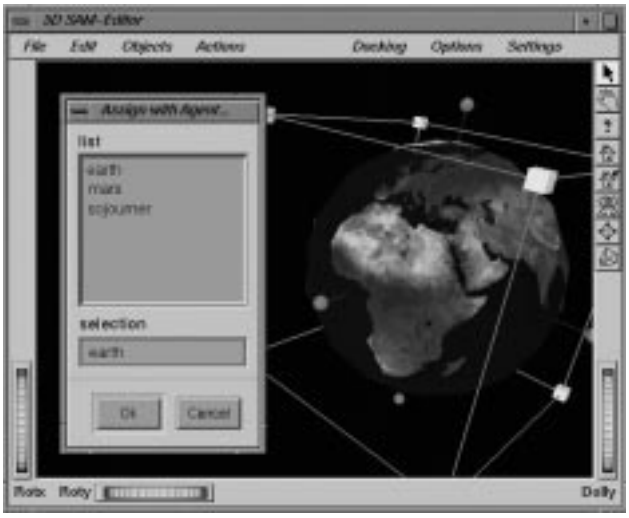


Figure 15. Assigment of Concrete Representation

chronous parallel language since synchronous languages are dedicated to the specification and analysis of parallel reactive behavior which is typical for virtual 3D micro worlds. For advanced communication, SAM implements broadcast, multicast, and reply. The selection and execution of rules is given by a smooth continuous animation. This greatly supports the first visual inspection of the program when checking the input/output behavior of objects. Advanced animations for rapid prototyping of 3D scenarios can be easily implemented by the means of few production rules. For the easy migration to an end-user representation the visual representation of each agent can be simply changed by assigning an external OpenInventor model to it.

Though our first results are most promising for its application as a general purpose 3D programming language, SAM still needs extensions in various directions. We are currently investigating several alternatives for SAM language extensions, e.g., for the definition of recursive behavior and loop constructs. Finally, in order to provide the basis for the specification of intelligent agents we think about concepts for dynamic rule sets with priorities.

Acknowledgments

We would like to thank Ralf Wegener for the implementation of the SAM editor and our colleagues at C-LAB and the Heinz Nixdorf Institut for various fruitful discussions.

References

[1] A. Del Bimbo, L Rella, E. Vicario. *Visual Specification of Branching Time Temporal Logic*. 11th Symposium on Visual Languages, IEEE Press, Los Alamitos, CA, 1995.

[2] M. Duecker, C. Geiger, R. Hunstock, G. Lehrenfeld, W. Mueller.*Visual--Textual Prototyping of 4D Scenes*. 13th Symposium on Visual Languages, IEEE Press, Los Alamitos, CA, 1997.

[3] M. Fromherz. *Visualization and Animation in Qualitative Modeling: A Case Study Using Pictorial Janus.* Technical Report SPL-92-112, Xerox PARC, Palo Alto, 1992.

[4] N. Halbwachs. *Synchronous Programming of Reactive Sytems*. Kluwer Academic Publishers, Dordrecht, 1993.

[5] K. Kahn, V.A. Saraswat. *Complete Visualizations of Concurrent Programs and their Executions*. 6th Symposium on Visual Languages, IEEE Press, Los Alamitos, CA, 1990.

[6] K. Kahn. *Drawing on Napkins, Video-Game Animation, and other Ways to Program Computers.* Communications of the ACM, Vol. 39, No. 8, 1996.

[7] M.A. Najork. *Programming in Three Dimensions. PhD Thesis.* Department of Computer Science, University of Illinois at Urbana-Champaign, 1994.

[8] A. Repenning, T. Summer. *Agentsheets: A Medium for Creating Domain-Oriented Visual Languages*. IEEE Computer, 28-3, 1995.

[9] W. Rosenbach. *3D Visualization and Animation of Communicating Agents*. Diploma theses . Universität Paderborn. 1997 (in german).

[10] F. van Reeth, E. Flerackers. *Three-Dimensional Graphical Programming in CAEL.* IEEE Symposium on Visual Languages, pp. 389 - 391, Bergen, Norway, 1993

[11] R. Stiles, M. Pontocorvo. *Lingua Graphica: A visual language for virtual environments.* IEEE Symposium on Visual Languages, pp 225 - 227, Seattle, WA, 1992

[12] K.Yamamoto. *3D-Visulan: A 3D Programming Language for 3D Applications.* Pacific Workshop on Distributed Multimedia Systems (DMS96) pp. 199-206, 1996.