A programming environment for behavioural animation

Introduction

The first development to put together research works done in the team has started in 1994. The first architecture of a general animation and simulation platform (GASP) can be found in [DC95]. We have then started to build both the kernel of this platform and a set of tools dedicated to the modelling of different aspects of an autonomous entity: geometry, dynamics, motion control, behaviour, artificial vision. The mechanical aspect is modelled with DREAM, our rigid and deformable bodies modelling system which generates numerical C++ simulation code for GASP [COZOT95b]. For human motion, a bio-mechanical approach has been preferred [MULTON99b]. HPTS concerns the modelling of the behavioural part of an actor [DONIKIAN98f] and is also used as an intermediate level for scenario authoring [DONIKIAN99c]. VUEMS is the acronym for Virtual Urban Environment Modelling System, and its main aim is to build a realistic virtual copy of urban environments in which we would perform behavioural simulations. Urban traffic has a high degree of complexity, as it requires interactions on the same thoroughfare between not only cars, trucks, cyclists and pedestrians, but also public transportation systems such as busses and trams. In the past years we have integrated all these transportation modes into GASP and applied this to different research projects [DONIKIAN96a,DONIKIAN98d,DONIKIAN99d,DONIKIAN99f]. In this page, we will first present the main characteristics of the simulation platform and then give an outline of the dedicated tools.

GASP: a modular programming environment

A behavioural animation is composed of a large set of dynamic entities evolving and interacting in a complex environment. To be able to model such applications, we need to implement different models: environment models, mechanical models, motion control models, behavioural models, sensor models, geometric models and scenarios. The main objective of GASP (General Animation and Simulation Platform) is to give the ability to simulate different entities composed themselves of different modules in different hardware configurations, without any change for the animation modules. When someone specifies a module, he does not have to make any hypothesis on the network location of other modules he must interact with. Nevertheless, he must be able to name them, and for that, modules are structured in a simulation tree. Each module of a simulation is a specialization of a class which can be viewed as the container of a computation function $Y = F(X, CP)$, where X is a set of inputs, Y a set of outputs and CP a set of Control Parameters. X and Y determine the data-flow from and to other objects. Each object has its own frequency and is activated periodically to calculate its new state. At each simulation step, the new input values are used to compute the outputs. This requires to connect each input of the object to an output of another object. This data dependency can be static or dynamic, as we cannot know at the beginning of a simulation, which objects might interact later. To take into account dynamic data dependencies, the number of inputs of an object can change during the simulation, unlike the one of outputs. A configuration file is used for each simulation to define which dynamic objects are used and on which hardware. As several processes can be used, this file describes first which processors are used, and then each process is named and located on a processor. As the modules of an entity can be located in separate processes, the location of each module is specified in the configuration file. During the simulation, inputs of an object must be supplied by values of outputs.

Rather than to define specifically how each reference object must send the new values of its outputs to interested reference object, it has been preferred an automatic mechanism which is based on a client/server mechanism. Each time the inputs of objects of a process require the value of the outputs of another one reference object, an object which contains only the outputs and control parameters of the reference object is created for this process: we call it a mirror object. The continuous communication between two agents can be managed by a two steps mechanism:

- firstly, the reference object communicates to its mirror the new value of its outputs and control parameters;
- secondly, the object interested by outputs or control parameters of another object can contact the embodiment of this object in its own process.

As each reference object runs at its own internal frequency, the data-flow communication channel must include all the mechanisms to adapt to the local frequency of the producer and of consumers (over-sampling, sub-sampling, interpolation and extrapolation). With the intention of minimizing communications between processes, the frequency of the communication between a reference object and each of its mirrors is computed especially for each mirror. GASP is currently available on both Unix and Linux workstations, including real-time, parallelism and distribution features. GASP offers a modular programming environment, and the kernel of the simulation platform manages the data-flow and event based communication between modules and their synchronization.

In the context of national research projects, GASP has already been used with success in other research labs in France. Our objective is now to make it available as an open-source version with an API in english, including documentation and tutorials. This new version of GASP (renamed OpenMASK for the occasion, as GASP is already used as a commercial product) is currently in beta version and will be available soon in its first release (summer 2002). Some application has been already developed with succes by using this new version.

HPTS: A Model for Behavioural Animation

Introduction

Information needed to describe the behaviour of an entity, depends on the nature of this entity. No theory exists for determining either the necessary or sufficient structures needed to support particular capabilities and certainly not to support general intelligence. As direction and inspiration towards the development of such a theory, Newell [NEWELL90] posits that one way to approach sufficiency is by modelling human cognition in computational layers or bands. He suggests that these computational layers emerge from the natural hierarchy of information processing. Lord [LORD94] introduces several paradigms about the way the brain works and controls the remainder of the human body. He explains that human behaviour is naturally hierarchical, that cognitive functions of the brain are run in parallel. Moreover cognitive functions are different in nature: some are purely reactive, while others require more time. Executions times and frequencies of the different activities are provided. Newell asserts that these levels are linked by transferring information across hierarchical levels, and that each of them operates without having any detailed knowledge of the inner workings of processes at other levels. All that is required is a transfer function to transform the information produced by one level into a form that can be used by another. Particularly important is the notion that symbolic activities occur, locally based on problem spaces constructed on a moment-to-moment basis.

Different approaches have been studied for the decision part of behavioural models in animation: sensor-effector or neural networks, behaviour rules, finite automaton approach. As human behaviour is very complex, none of the preceding models could be applied. More recently, a second generation of behavioural models has been developed to describe the human behaviour in specific tasks. The common characteristics of these new models are: reactivity, parallelism and different abstract levels of behaviours. In [BURMEISTER97], authors describe a multi-agent development environment named DASEDIS and use it to describe the behaviour of a car driver. In [NOSER97], a tennis game application is shown, including the behaviour of players and referees. A stack of automata is used to describe the behaviour of each actor. In the Motivate product proposed by Motion Factory for the Game Design Market, they have also introduced Hierarchical Finite State Machines, and actions associated with the states and transitions can be described by using an object-based programming language, named Piccolo [KOGA98]. As humans are deliberative agents, purely reactive systems are not sufficient to describe their behaviour. It is necessary to integrate both cognitive and reactive aspects of behaviour. Cognitive models are rather motivated by the representation of the agent's knowledge (beliefs and intentions). Intentions enable an agent to reason about its internal state and that of others. The centre of such a deliberative agent is its own representation of the world which includes a representation of the mental state of itself and of other agents with which he is currently interacting [TERZOPOULOS99]. To do this, Badler et al. [BADLER97a] propose to combine Sense-Control-Action (SCA) loops with planners and PaT-Nets. SCA loops define the reflexive behaviour and are continuous systems which interconnect sensors and effectors through a network of nodes, exactly

like in the sensor effector approach described above. PaT-Nets are essentially finite state automata that can be executed in parallel (for example the control of the four fingers and of the thumb for a grasping task). The planner queries the state of the database through a filtered perception to decide how to elaborate the plan and to select an action. More recently they have introduced Parameterized Action Representation (PAR) to give a description of an action, and these PARs are linked directly to PaT-Nets. It allows a user to control Autonomous Characters actions by instructions given in natural language [BADLER2000b]. In all these systems, the action is directly associated with each node, which doesn't allow the management of concurrency.

HPTS

According to Newell, our goal is to build a model which will allow some adaptative and flexible behaviour to any entity evolving in a complex environment and interacting with other entities. Interactive execution is also fundamental. This has lead us to state that paradigms required for programming a *realistic* behavioural model are: reactivity (which encompasses sporadic or asynchronous events and exceptions), modularity in the behaviour description (which allows parallelism and concurrency of sub-behaviours), data-flow (for the specification of the communication between different modules), hierarchical structuring of the behaviour (which means the possibility of preempting sub-behaviours) and time and frequency handling for execution of sub-behaviours (This provides the ability to model reaction times in perception activities). HPTS [DONIKIAN95b] which stands for Hierarchical Parallel Transition Systems, is a formalism proposed to specify the decisional part of an autonomous character. It offers a set of programming paradigms, which permit to address hierarchical concurrent behaviours. It consists of a reactive and cognitive model, which can be viewed as a multi-agent system in which agents are organized as a hierarchy of state machines. Each agent of the system can be viewed as a black-box with an In/Out data-flow, a set of control parameters and an internal state. The synchronization of the agent execution is operated by using state machines. To allow an agent to manage concurrent behaviours, sub-agents are organized inside sub-state machines. In the following, agents will be assimilated to their state machine angle, as there is not any constraint imposed on the programmer, concerning the body part of the agent. Each state machine of the system is either an atomic state machine, or a composite state machine.

Though the model may be coded directly with an imperative programming language like C++, we decided to build a language for the behaviour description. Figure 1 presents the syntax of the behavioural programming language which fully implements the HPTS formalism. The behavioural description language is not described in details. For a complete description of the model refers to [DONIKIAN01b], and for the management of resources and priority levels, refers to [DONIKIAN01e]. Keywords are written in bold, whereas italic typeface represents a non-terminal rule. A * stands for a *0..n* repetition while a $^{+}$ stands for a *1..n* repetition and a statement enclosed in **{ }** is optional.

SMACHINE *Id* ;
{
PARAMS *type Id {, type Id}** ; // Parameters
VARIABLES { *{type Id ;}** } // Variables
OUT *Id {, Id}** ; // Outputs
PRIORITY = *numeric expression* ;
INITIAL *Id* ; **FINAL** *Id* ;
STATES // States Declaration
{{
Id {[*Id {, Id}*]*} {***RANDOM***} {***USE** *resource list}*;
{{ /* state body */ }}
}+}
*{***TRANSITION** *Id {***PREFERENCE** *Value}*;
{
ORIGIN *Id* ; **EXTREMITY** *Id* ;
*{***DELAY** *float* ;*}* *{***WEIGHT** *float* ;*}*
*read-expr | write-expr {***TIMEGATE***}* ;
{{ /* transition body */ }}
}}*
}

Figure 1 : syntax of the language.

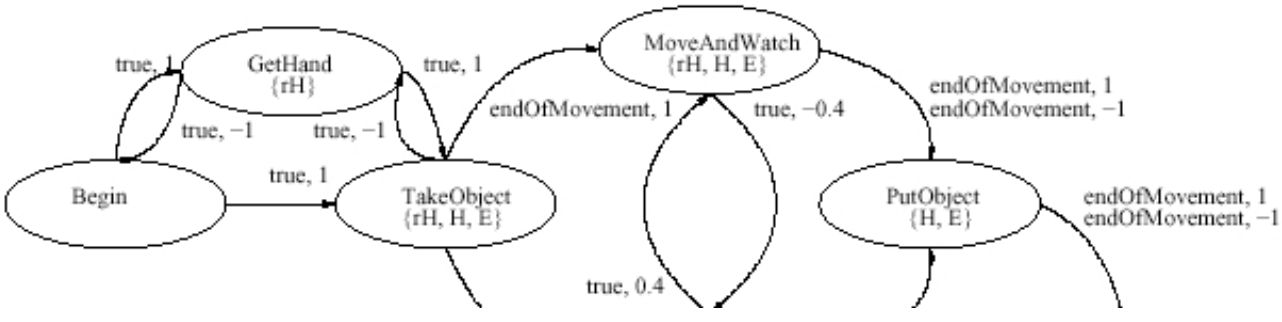
The description of a state machine is done in the following way: the body of the declaration contains a list of states and a list of transitions between these states. A state is defined by its name and its activity with regard to data-flows. A state accepts an optional duration parameter which stands for the minimum and

maximum amount of time spent in the state. Resources used by a state are defined by using the instruction *USE resource list*. A state machine can be parameterized; the set of parameters will be used to characterize a state machine at its creation. Variables are local to a state machine. Only variables that has been declared as outputs can be viewed by the meta state machine. A priority is attached to each state-machine and consists in a numeric expression which allow the priority to evolve during the simulation. A transition is defined by an origin, an extremity, a transition expression, two optional parameters and a transition body. The transition expression consists of two parts: a **read-expr** which includes the conditions to be fulfilled in order to fire the transition, and a **write-expr** which is a list of the generated events and basic activity primitives on the state machine. A preference value is defined for each transition.

Afterwards, C++ code for our simulation platform GASP [DONIKIAN98b] is generated. It is totally encapsulated: all transitions systems are included in their own class directly inheriting from an abstract state machine class which provides pure virtual methods for running the state machines and debugging methods. An interpreter has also been implemented, which is very useful for the behaviour specification phase as it allows to modify state-machines during the execution phase with an increase of only ten percent on the execution time.

Behaviour Coordination

Reproducing daily behaviours requires to be able to schedule behaviours depending on resources (body parts) and priorities (intentions or physiological parameters). A simple way is to say that behaviours which are using the same resources are mutually exclusive. This approach is not sufficient to obtain realism, as in the real life, humans are able to combine them in a much microscopic way. All day long, human being combines different behaviours, like for example reading a newspaper while drinking a coffee and smoking a cigarette. If all behaviours using common resources were mutually exclusive, an agent could not reproduce this example, except if a specific behaviour, integrating all possible combinations, is created for this purpose. This solution becomes rapidly too complex, and has motivated the recent integration of resources and priority levels into HPTS [DONIKIAN01e]. In the contrary of some previous approach, it is not necessary to specify exhaustively all behaviours that are mutually exclusive; this is done implicitely just by attaching resources to nodes, preference values to transitions and a priority function to each state machine, and by using a scheduling algorithm at run-time. Each state of a state machine can use a set of resources which can be considered as semaphores. Thus, resources are used for mutual exclusion between behaviours. Entering a node implies that resources are marked as taken and exiting it implies that thoses resources are released. In order to control the execution of parallel state machines and to offer an automatic adaptation between different behaviours, it is necessary to add notions of priorities and preferences. The degree of preference is a coefficient associated to a transition and corresponds to the proclivity of the state machine to use this transition when the associated transition is true. This coefficient allows to describe a behaviour with different ways of realization; possible adaptation depending on resources availability or need can be described. A priority function is associated to each state machine. This function returns a value representing the importance of a behaviour in a given context. This function can be used to control a behaviour during th erunning phase. As it is user defined, it can be correlated with the internal state of the character (psychological parameters, intentions) or with external stimuli. By combining those two notions, it is possible to create a scheduling method which globally favours the realization of most important behaviours while automatically adapting the execution of running ones. The scheduling system allows to describe independently all behaviours with their different possibilities of adaptation. During running phase, the adaptation of all other running behaviours is automatic. Moreover, consistency is ensured because the scheduler can only exploit consistent propositions of transition for each behaviour depending on the others.



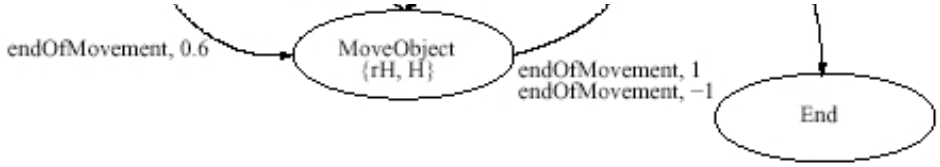


Figure: Moving object behaviour.

In the example of the above figure, the behaviour uses the following set of resources: Hl (left hand), Hr (right hand), rHl (reserve left hand), rHr (reserve right hand), M (mouth) and E (eyes). Resources rHl/rHr are used to handle releasing of resources Hl/Hr. The scheduler can only act on the next transition of a state machine. Hands are resources that often need more than one transition to be freed, for instance, putting down an object to free the hand resource. Then a state which only use resource Hr/Hl corresponds to a behaviour of freeing a hand resource. Note that once behaviours are described through state machines, they are controlled through their priority. This property allow to handle every type of executive behaviour without need of information about their internal structure in term of resources or possible adaptations.

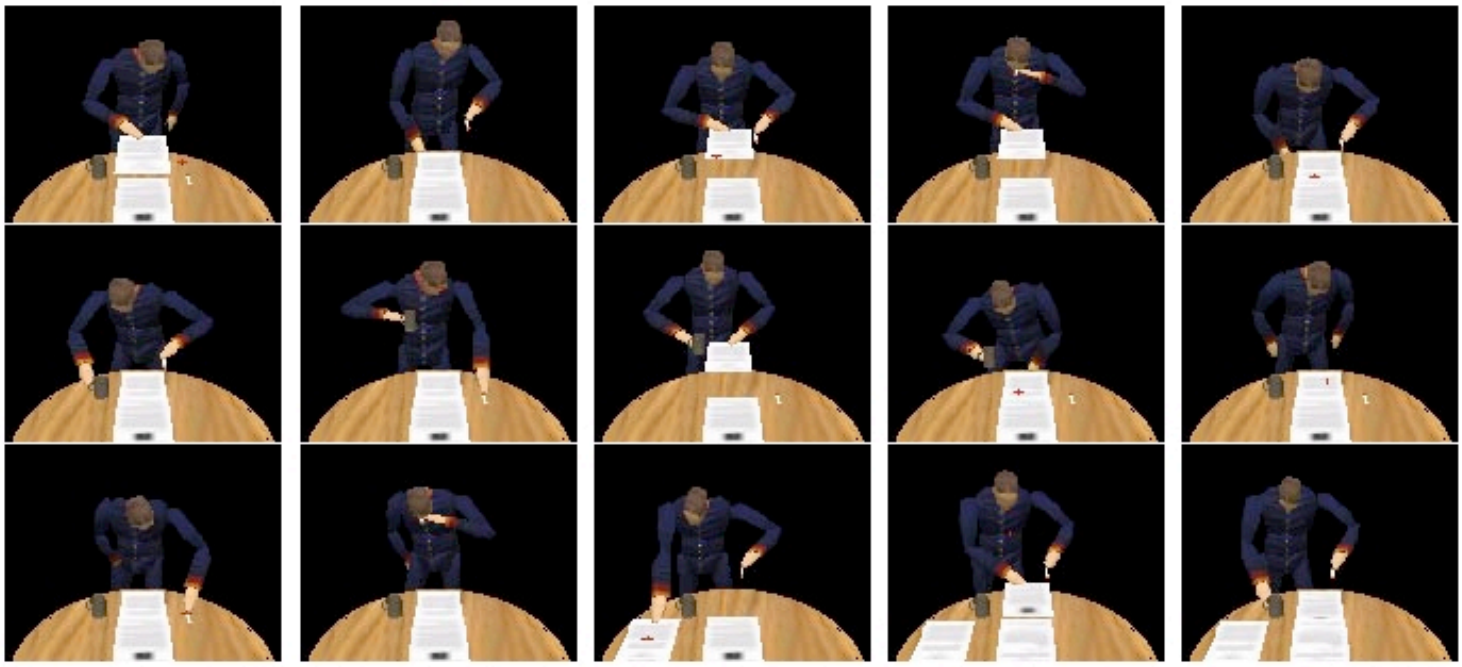


Figure: Behavioural Coordination Example.

Video Illustrations: Front view, Subjective view

Informed Environments

Using 3D modelling systems allow the generation of realistic geometrical models in which walk through is possible in real time. As this modelling operation is still a long, complex and costly task, a lot of work has been done to partially automate the rebuilding process. All these techniques are very useful for the visual realism of virtual urban environments but they are not sufficient due to the lack of life of these digital mock-ups. Walking through these virtual city models do not provide a real life feeling as they are uninhabited. In order to populate these virtual environments, it is necessary to specify the behaviour of autonomous characters with the ability to perceive their surrounding space and act on it. An autonomous actor whose main action is obstacle avoidance in an unstructured environment does not need other knowledge than the geometrical one. In order to simulate more sophisticated behaviours, other kind of information must be manipulated. The simplest behaviour, for a pedestrian walking in a street, consists in minimizing possible interactions, which mean avoiding static and dynamic obstacles. But, even in this simple walking activity, one needs to know the nature of objects he will interact with. For example, a public phone is considered as an obstacle to avoid for most people, but some of them will be interested by its functionality and will use it. For the crossing of a street, one activity consists in reading the signals, which mean that it is necessary to associate semantic information to geometric objects in the scene, and to update it during the simulation. N. Farenc [FARENC99] has suggested using an informed environment,

dedicated to urban life simulation, which is based on a hierarchical breakdown of an urban scene into environmental entities providing geometrical information as well as semantic notions. S. Raupp Musse [MUSSE2000] has used this informed environment to animate human crowds by using a hierarchical control: a virtual human agent belongs to a group that belongs to a crowd, and an agent applies the general behaviours defined at the group level. The knowledge on the virtual environment used by the crowd is composed of a set of obstacles (bounding box information of each obstacle to be avoided), a list of interest points (locations that the crowd should pass through and their associated regions) and a list of action points (regions where agents can perform actions).

In the realm of behavioural psychology, there have been a few studies on visual perception, mainly based on Gibson's theory of affordances [GIBSON86]. The theory of affordances is based on what an object of the environment {\em affords} to an animal. Gibson claims that the direct perception of these affordances is possible. Affordances are relations between space, time and action, which work for the organism. What the world is to the organism depends on what the organism is doing and might do next. For computer scientists, Gibson theory is really attractive because it assigns to each object some behavioural semantic, i.e. what the human being is likely to do with a given object. Associating symbolical information to objects, Widyanto experienced the Gibson's theory of "affordances" [WIDYANTO91], while M. Kallmann [KALLMANN98] introduces smart objects, in which all interesting features of objects are defined during the modelling phase.

In accordance with Gibson's ecological theory, components of the virtual urban environment should be informed. To produce more realistic behaviours, we [DONIKIAN2000] have specified a city model *affordant* to autonomous actors. In the field of psycho-sociology, M. Relieu [RELIEU98] has defined the notion of positive and negative affordances for a pedestrian. A positive affordance specifies the fact that the extrapolated trajectory of the corresponding pedestrian is not supposed to intersect the planned trajectory of the other one, while a negative affordance points out that they may collide. M. Relieu says also that a mobile entity uses the urban discrimination \footnote{A street is composed of connected lanes devoted to distinct mobile entities such as cars and pedestrians} to focus his attention, to select pertinent information for his actions inside the current region, while he maintains a secondary task to observe what is happening in regions close to its circulation area. Thanks to this knowledge, autonomous virtual actors can behave like pedestrians or car drivers in a complex city environment. A city modeller, named VUEMS (Virtual Urban Environment Modelling System), has been designed, using this model of urban environment, and enables complex urban environments for behavioural animation, and their 3D geometric representation, to be automatically produced. The scene produced by VUEMS is loaded and is then available for use by all autonomous entities. First, sensors can determine visible objects in their environment and then the behavioural module can have access to the information on these visible objects. The behavioural model of pedestrians, that has been developped, includes social and driving rules of interaction (minimize the interaction and choose in priority the left side to overtake), as explained in [THOMAS00].

Scenario Authoring

The scenario component of a behavioural simulation carries out the responsibility for orchestrating the activities of semi-autonomous agents that populate the virtual environment. In common practice, these agents are programmed as independent entities that perceive the surrounding environment and under normal circumstances behave as autonomous agents. However, in most experiments and training runs [DONIKIAN99d], we want to create a predictable experience. This is accomplished through direction of objects behaviours. To facilitate coordination of activities, objects have to be built with interfaces through which they can receive instructions to modify their behaviours. The scenario manager controls the evolution of the simulation by strategically placing objects in the environment and guiding the behaviours of objects to create desired situations. We have specified a scenario language [DONIKIAN99c,DEVILLERS01], which permits to describe scenarios in a hierarchical manner and to schedule them at the simulation time. This language has its own syntax and is composed of a set of specific instructions. The language contains classic instructions such as *if, switch, repeat until, random choice, wait* and which executed inside a timestep. It contains also more specific instructions (*waitfor, eachtime, aslongas, every*) that will spend more than one timestep to be finished and that can run in parallel during the execution of the scenario they belong to. All those instructions can be composed in a hierarchical manner. To manage actors, the language offers also specific instructions to specify the interface of an actor,

to reserve and free actors. By using priorities, a scenario cans steal an actor to another one, which will be informed of it by a message. Concerning messages, each scenario can subscribe to messages that are of interest for itself.

As the language is built upon C++, it is always possible to include C++ code everywhere inside scenarios by using four specific instructions:

- *include { ... }* , is used to insert C++ include commands;
- *declaration { ... }*, is used to declare C++ variables that will be used inside the scenario (Thus, we did not have to manage our own datatypes);
- *implementation { ... }*, is used to insert c++ code inside a scenario;
- *destructor { ... }*, is used to cleanly terminate the c++ part of the scenario.

A scenario can be decomposed into sub-scenarios and each scenario is corresponding to a set of tasks or actions, ordered on a global temporal referential. Tasks in a scenario can modify characteristics of actors, create them or ask an actor to perform a specific action. Internal representation is based on the use of Allen's logic and of rewriting rules to produce Hierarchical Parallel Transition Systems. A scenario can start at a predefined time given by the author but can also be started when a situation occurs (conditional scenario). Some of those scheduling informations are stored in a dynamic execution graph. A scheduler uses it to start or terminate scenarios. To detect situations, triggers and sensing functions are managed by the simulation observation.

References