Introduction to RAGS Release 1.0

Author : Elpida Tzafestas
Revised version
January 1996

Introduction to RAGS Release 1.0

Author : Elpida Tzafestas[1]

LAFORIA-IBP, Université Pierre et Marie Curie
4, Place Jussieu, Paris Cedex 05, FRANCE.
brensham@laforia.ibp.fr

Contents

---

[1] *The author is currently with the Intelligent Robotics and Automation Laboratory, Electrical and Computer Engineering Department, National Technical University of Athens, Zographou 15773, Greece. e-mail : brensham@softlab.ece.ntua.gr, http://www.softlab.ece.ntua.gr/~brensham*

Foreword

RAGS (Reactive AGents Simulator) has been initially designed and developed by the author as a customized simulation and experimentation testbed to be used for cellular robots modeling and simulation. Extensive use of the system, however, revealed many features shared across different simulation projects with different purposes and, most important, different agent models. A set of tools has been then developed that support this functionality at the system and application level and a method of simulation component specification has been designed and programmed. The present first general-purpose experimental version contains all the conceptual and programming tools necessary for specifying, running and evaluating experiments with multi-agent systems; those tools are agent-independent, i.e. the user of the system may define his/her own agent models and classes, only obeying to some minimal agent specification rules. By being general-purpose in this sense, RAGS aims at a large audience, ranging from experienced programmers, who might want a starting point for developing their own customized simulators, to novices that might wish to get started with multi-agent modeling and simulation. The system is an ideal testbed for specifying and running in reasonable time and with reasonable effort medium-term modeling and simulation projects, in domains like behavioral studies, ethology, economics, ecology etc. An average application programmer needs to have a good knowledge of the basic functionality of the Smalltalk-80 language version 4.1 as well as some knowledge of the Visualtalk interface development toolkit version 2.0. End users with limited or no knowledge of Smalltalk may use the system to experiment with, provided special-purpose interfaces are built on top of the basic RAGS graphic programming facilities.

1 Introduction

Simulations are useful when experimentation with the real-life counterpart is impossible, impractical or undesirable (imagine, for example, simulation of an eco-system or simulation of a nuclear reactor's operation under abnormal conditions). Simulations are also useful when analytical methods don't exist, or when the mathematical computations involved are too complicated. Finally, simulations are useful whenever the properties sought are not easily describable analytically (such as formation of spatial clusters in prey-predator systems). In all cases, simulation relies on the construction of a model of the system, i.e. an entity for which a 1 to 1 correspondence with the real system exists, such that all operations carried out with the real system have a counterpart in the model and the state of the real system at any time can be deduced without ambiguity from the state of its model[2]. A model is then an entity easier to manipulate and experiment with than the real system it stands for and results in the model state space may be transposed to the real system state space.

Multi-agent systems are systems (i.e. collections of components) having the additional feature that individual components, the agents, have a degree of atomicity, which is perceived by outside observers as intentionality in activity.[3] That is, agents are thought of as taking individual personalized "decisions" to act independently. In the example of the car trajectory simulation, the car has a goal, whether this be a particular final destination or a direction to follow or otherwise. Introducing such egocentric agents therefore allows for scaling the system on the number of agents : a 5-car system differs from a 25-car system only on the number of cars if there is no centralized processing of the cars' trajectories. The complexity of the two-party conflict resolution mechanism necessary for individual navigation, is constant on the number of cars.

Systems amenable to multi-agent modeling and simulation are all those that involve high numbers of possible heterogeneous egocentric agents for which modeling of the interactions is easier that modeling of the global resulting phenomena. Analysis of such multi-agent systems is difficult or even impossible due to high global complexity, heterogeneity and agent adaptivity (modification of internal parameters as a side-effect of activity).

RAGS has been developed to specify and run instances of reactive agents, although there is no restriction as to the complexity and sophistication of the agent models defined by the application programmer. In general, reactive agents are those agents whose activity follows a stimulus-response (S-R) scheme. Here, reactivity is used with the sense of responsiveness to external, and possibly unforeseen, events. This means that primitive "reactive" agents are allowed to have state (memory) across activations.

---

[2] Note that we can only talk about models of *systems* rather than isolated entities. A car's trajectory won't be simulated in itself, because it doesn't exist as such. A car's trajectory should be rather viewed as the outcome of interaction of the car with the physical world imposing constraints and laws.

[3] An agent is an agent because an observer may attribute some intentionality to it.

Furthermore, simple reactive agents may be combined to composite agents of higher levels (for example, an agent representing a simulated robot may be composed of a number of lower-level agents, or tasks, competing with each other for robot resources). The power of such reactive agents lies in their simplicity and their scalability. For example, it is straightforward to endow agents with adaptive capabilities, or to scale the number of tasks an agent has etc. RAGS is ideal to define and experiment with agent models in a bottom-up way by starting out with minimal agent requirements and proceeding incrementally to more sophisticated models.

To facilitate experimentation as well as to allow future extension of the system, RAGS has been developed as a small-size testbed rather than an isolated special-purpose simulator, that is, a number of general-purpose tools have been written to support user-defined models and needs. In this sense, RAGS is both a simulator and a simulator generator. It follows the hierarchical design and development modularization into system, application and user level.

System level : All lower-level primitives used by the system and shared across different applications. Those include tools for specifying, initializing and running simulations, observing simulations, gathering and displaying results, subclassing agents, visualizing and editing different simulation entities etc.

Application level : All code specific to the corresponding application. For applications that do not need to redefine crucial parts of the simulator, such as the time management facilities or the agent execution mode, writing the simulation specifications and the specific agent code suffices.

User level : All the facilities necessary for an end-user to navigate into the system and experiment with societies of agents. Often, in scientific applications, the user is the application programmer, so that only a few facilities need be supported at this level.[4] However, and for users that are not programmers, the graphic possibilities of Objectworks and Visualworks, together with the basic graphic programming tools of RAGS, have to be exploited to develop customized user interfaces.

Portability : RAGS works without modifications on UNIX workstations, PC's or MacIntosh computers. The user has simply to load the RAGS sources on his/her local Smalltalk-80 image.

In this general introduction to RAGS, we gave a flavor of the philosophy of multi-agent modeling and simulation –the interested reader may refer to the references and to their bibliographic part for tutorial and in-depth treatments of the subject. In chapters 2 to 7, we describe briefly the tools developed at the system level, while in chapter 8 we summarize the necessary steps for building a custom application. The RAGS classes' comments provide more details on the internal workings of the corresponding tools and may be obtained on-line by evaluating the expression *"RagsManager fileOutComments"*, which will output all the system comments on a file named *'rags.cm'* unless otherwise declared by the user in the popup dialog evoked by the command.

---

[4] The author has been running large numbers of long and tedious experiments without writing special-purpose graphic programming facilities, but only by evaluating expressions in a Smalltalk workspace.

2 Simulation

A multi-agent modeling and simulation project involves a three-cycle process : modeling (i.e. transcribing the studied system in terms of a multi-agent system), definition and run of experiments[5] (simulations), gathering and evaluation of the results. Results may be such that some backtracking be necessary, either to define and run additional experiments or even to refine the model. One major concern during RAGS development has been to provide maximum *automation* of the experimentation and refinement process, so as to speed up the above iterative three-phase study.

To account for automation, a mechanism of *specification* of the simulation has been developed on top of the modularization principle of the software development. If we assume that initialization and execution code is standard, a simulation (or experiment) is completely specified by the entities it consists of. The results obtained are then directly associated with the specifications to which they correspond, thus structuring the experimentation process.

A multi-agent simulation –experiment– is represented by an instance of the class `RagsProblem`, or one of its subclasses, and has the following components (simulation entities) :

- *Agents :* All those entities that are engaged in some sort of activity and may respond to changes in their immediate environment. Agents are different from (passive) objects in that they are dynamic or mobile or both (more details may be found in chapters 4 and 5).
- *World (or environment) :* It is the world the agents are situated in and that they may perceive and modify at will. In degenerate cases, where a user is interested in agent interactions outside of an explicit world, this may be defined as a unique place world (so that agents won't move) with no properties and will serve only to specify the simulation.
- *Observers :* It is a collection of objects that act as parasites. They are attached to specific simulation entities (the world or agents) to observe certain properties or values that represent direct or indirect results. The actual results are gathered into an object called *statistics* and may be visualized as a text or as a collection of curves/bar charts or output on a transcript or a file.
- *Termination :* It is an object representing a boolean expression to be evaluated and whose value *true* signals completion of the experiment. It resembles observers in that it depends on some of the simulation entities and/or the simulation instance itself. Terminations may be simple logical expressions (boolean tests) or AND/OR

---

[5] The use of the term experiment with reference to a model, rather than a real system, may seem misleading. Note, however, that often the models are interesting in themselves as alternative realities rather than representations of reality (*"A theory has only the alternative of being right or wrong. A model has a third possibility -it may be right but irrelevant"*, Manfred Eigen).

trees (see `Termination` and `CompoundTermination` classes in *Rags-Utilities* category).

Initialization of the simulation on a simulation specification decomposes into separate initialization of the world, the agents, the observers and the termination on the corresponding specifications. The *execution cycle* (or step) consists of looping through the list of agents giving each one a chance to execute. The list of agents may be taken either as it is or after shuffling the agents (in distributed systems theory, it is well known that random execution of agents yields more reliable results and enhances overall stability).

Time management is based on a time-slice technique with some optional memory : each agent responds to a message `#elapsed:` and may have an internal clock representing the duration of its current "action". Every time this duration expires the agent executes (if the agent does not have an internal clock, it executes always). This technique is called the *time-triggered execution technique*, since it assumes that each agent is triggered by the elapse of time. Actually, each agent is triggered by what are defined as its inputs (for instance, a simulated neuron is triggered by the presence of signals in its input lines), so that this mechanism is an artificial representation of the time lag between data input and action output at the single agent level. The advantage of using this technique rather than a usual synchronous or pseudo–synchronous round-robin is that it allows asynchronicity between the variable time-length actions of different agents. This time management technique is directly inspired from traditional discrete-event simulation, with the difference of defining distributed rather than centralized event management. The presence of a clock in simulations does not imply any synchronization; the clock should instead be viewed as a time–counter running in parallel with agents.

3 Specifications

One RAGS originality is the specification mechanism which simplifies experimentation by automating large parts of the simulation definition and execution. The specification definition and processing is independent of the nature, structure and semantics of the specified objects, and hence one of RAGS invariants.

A specification is a dictionary of associations of the type <key>-<value>, where keys are attributes of the specified object. For attributes that are itself composite objects rather than primitives (numbers, symbols, characters, strings, sets of primitives...) the equivalent specification value is defined recursively as a specification <class name>-<spec> (see also `Specification` class in the *Rags–Utilities* category and browse the *buildSpec* methods).

Example. For a car having the attributes (which correspond to distinct instance variables) *color*, *weight*, *maximum_velocity*, shape and *accessories*, one possible specification is

```
Car specification =
  (#color     #red,
  #weight     2,
  #maximum_velocity    200,
  #shape           spec1,
  #accessories     spec2)
```

where

```
spec1 =
     (#CarShape
     (#curved      true,
     #type         #lagunaSmall)),
```

CarShape is the name of a class

and

```
spec2 =
  (#StrophoMeter
    (#scale   5000,
    #color    #black),
  #AirConditioning
```

StrophoMeter and AirConditioning are names of classes

```
    (#autonomous       true,
    #compatibility     #renault)).
```

Not every attribute (instance variable) of an object has to be specified. Instance variables private to objects (like the internal timers of agents), auxiliary variables etc. need not be specified. Furthermore, all specifiable variables have to have a default value used for initialization whenever a value for that variable is not given by the specification. On initialization, the *specification fields* of the object –defined as an instance variable of its class– are scanned and the corresponding specified values –or the default ones, if no specification is given– are assigned to the fields. Agent classes have also *nested* and *recursive specification fields* corresponding to recursively specifiable objects :

- *Nested specification fields* : Fields corresponding to instance variables whose values are specifiable objects (like the *shape* field above).

- *Recursive specification fields* : Fields corresponding to instance variables whose values are variable-size collections of specifiable objects, for instance the *accessories* field above or the *components* field of composite agents (`CompositeAgent` class). Currently, those specifiable collections are supposed to consist of heterogeneous elements (no two of them are allowed to be of the same type, i.e. direct instances of the same class). Both nested and recursive field specifications' keys are class names.

Use of this elegant and uniform specification mechanism relies on a common initialization method *newFrom: spec* to which all specifiable objects should respond. Objects having nested/recursive specifiable variables propagate this message to their respective components. This way, the structure of specifications is analogous to the structures of objects they refer to. Furthermore, the specification mechanism meets the RAGS author's concern *to simulate structures' operation rather than observable functionality*, i.e. to reify and engineer the structures that give rise to observed phenomena rather than hand-code into the agents what a human observer thinks of them.

The instances of `RagsProblem`, i.e. the simulations themselves, are initialized on a specification with four keys : `#AgentsSpec`, `#EnvironmentSpec`, `#ObserversSpec` and `#TerminationSpec`.

All problem/agent/object classes' specifications may be parsed from an array description (see for example the *defaultAgentsSpec*, *defaultEnvironmentSpec* etc. in the `Demo` class). Special care is taken for points and dictionaries (other than specifications). The point 30@50 is parsed as #(#Point 30 50), while a dictionary containing the associations #potatoes-3 and #type-#black is parsed as #(#Dictionary #potatoes 3 #type #black). Furthermore, a possibility for macro substitution is provided : for example, when parsed, the expression #(#aClassName #aMessage) returns the result of the expression *(aClassName aMessage).*

4 Worlds

RAGS world is a *geometric repository* of heterogeneous agents, objects and signals, whose evolution in time may be observed. It is represented in the simulation environment as a closed, two-dimensional discrete grid. Each point in the grid is itself an object –instance of `WorldPlace` class– which serves as a local repository of agents, objects and signals. The use of such local repositories accelerates access to individual entities in a certain position or around it.

Objects, as opposed to agents, are static and passive and may be, in general, directly perceived, accessed and modified by agents. Obstacles and walls are typical examples of objects in the car navigation case. Objects are instances of `SituatedSimulationObject`[6].

Signals are "immaterial" objects. They may be perceived by agents just like objects but they are not situated, that is they don't occupy any physical space. Temperature and heat in an ecological simulation would be represented as signals. Signals may be global to the simulated world or local to individual world places (in which case they are called *properties*).

There is no special provision for inter-agent communication. In reality, communication is achieved by emission and perception of signals in the world. This constitutes *a unified framework of perception and communication*.

RAGS worlds are instances of the `RagsWorld` class and contain as instance variables the grid of places (a 2-D array of instances of `WorldPlace`) and a dictionary of signals (see *Rags-Environment* category).

As defined, RAGS worlds are representations of physical spaces in which agents are situated. However, many abstraction possibilities may be envisaged during advanced application development : non-euclidean metric worlds where agents are situated in many places simultaneously, multiple worlds focusing on different observed aspects etc.

[6] `SituatedSimulationObject` is a subclass of `SimulationObject`, whose instances are used as generic specifiable objects, for example as passive components of agents.

5 Agents

Agents are all those entities that are engaged in some sort of activity and may respond to changes in their immediate environment. Agents are thus different from objects in that they are dynamic or mobile or both. Simulated cars with a goal point would be represented as agents. Other agents present in a simulation may include uncontrollable agents whose behavior must be taken into account, but on which the designer/observer has no influence. This could be for instance the case of circulating buses among which the cars have to navigate. As said in the introduction, therefore, agents are all those entities that can be said to have an intentionality of their own.

Agents are classified as follows (see also *Rags-Agents* category) :

- *Simple or composite agents.* Simple agents are primitive agents no further decomposable into autonomous (intentional) components. Composite agents are agents consisting of lower level simple agents. Generally, composite agents have to perform some conflict resolution or arbitration of component agents. The most usual category of composite agent are those consisting of a number of tasks referring to separate activities and among which the agent has to arbitrate. For example, a navigating car-agent may have a number of tasks for different goal-points, in which case a possible task arbitration mechanism would be to assign different dynamic priorities to each one of them. Composite agents' components are generally instances of `IndivisibleTask`[7].
- *Agents situated or not.* Situated agents are all those agents actually existing within a given position/place in the world and taking part directly in the simulation execution step, while agents that are components of other composite agents are generally not situated. For example, the navigating car is a situated agent while its component tasks are not.

The actual code run by each one of the simple agents at the leaves of the agent tree structure is user-defined (method *elapsed:*) and may be perception functions ("reading" and possibly distorting data of the world), agent internal parameter modification (adaptation and/or learning) or actual "actions" in the world (such as moves, emission of signals etc.).

Agents of the same simulation may be very heterogeneous in structure, knowledge and ability. They may also have very different internal execution times (imagine for example slow and fast cars) as well as limited and diverse scopes of perception and action.

[7] This name was historically meant to differentiate between tasks *modeled* that way and cellular robots' tasks *implemented* as collections of concurrently executing cells.

6 Observations

Evaluation of a simulated system involves inspection of several aspects of the simulated world's state. To be able to evaluate once those aspects are identified, *an observation mechanism* was introduced. A number of special objects, called observers, are attached to all those entities (for instance agents) that are to be observed and operate as parasites during simulation, observing and recording values etc. The RAGS observation mechanism builds upon the Smalltalk *dependency* mechanism (see Smalltalk documentation and *Rags-Observers* category) : observers are dependent on observed objects registered as their *references*. Those references are responsible for sending the message `self changed` or one of its variants whenever something important happens –the message propagates to their dependents, among which are their observers, and decide locally whether to update their state or not. Once the simulation is over, the states (or else measures) of all observers are interpreted and collected to a statistics record of the simulation. Clearly, this constitutes an advantage of the simulation environment over real-world counterparts : we can evaluate and we can do it without modifying the real world, what would risk to bias the results obtained. Furthermore, introducing an explicit observation mechanism forces a user to formalize the conditions of interest. This would also constitute an adequacy criterion for an observation mechanism : whether it allows to formalize the conditions of interest and represent the results in a straightforward, easy to interpret way.

The main observer classes are :

- *Enumerators.* Observers of a variable that count the number of their activations.
- *Adders.* Enumerators that compute the sum of the values that have activated them.
- *Averagers.* Enumerators that compute the average of the values that have activated them.
- *Compound observers.* Observers that consist of a number of other observers of heterogeneous structure. Each of the component observers observes a different aspect of the same Smalltalk object.
- *Collection observers.* Observers that consist of a number of other, structurally identical, observers, used to observe collections of Smalltalk objects, for instance collection observers on all the agents of a simulation.
- *History recorders.* Observers registering the series (or else the history) of values that have activated them.

Simple observers' states are translated into statistics records, generally holding the minimum, the maximum and the average of all the values encountered. Collection and compound observers' states are translated to special dictionary structures, instances of the `Statistics` class –which is itself an indirect subclass of Dictionary– supporting additional summary and visualization facilities (see *Rags-Statistics* category). History recorders' statistics are the tables of values and may be visualized as curves.

7 Interfaces

RAGS interfaces are of two kinds :

- Graphic visualization and programming interfaces
- Textual interfaces and log/description files.

All graphic programming facilities (editors) are indirect instances of the `RagsEditor` class where all the basic facilities for opening/saving/updating and nested editors processing etc. facility are implemented. Editors usually open on a temporary copy of the reference object, so that all modifications may be aborted. At most one editor at a time may be open on one object. Editors may be opened recursively from within other editors and changes are propagated on closure through the use of a special *closing specification* implementing actions on return points in the program (for details, refer to the comment of the class `RagsEditor`) The closing specification mechanism also allows for editors to close recursively whenever the top-level editor closes.

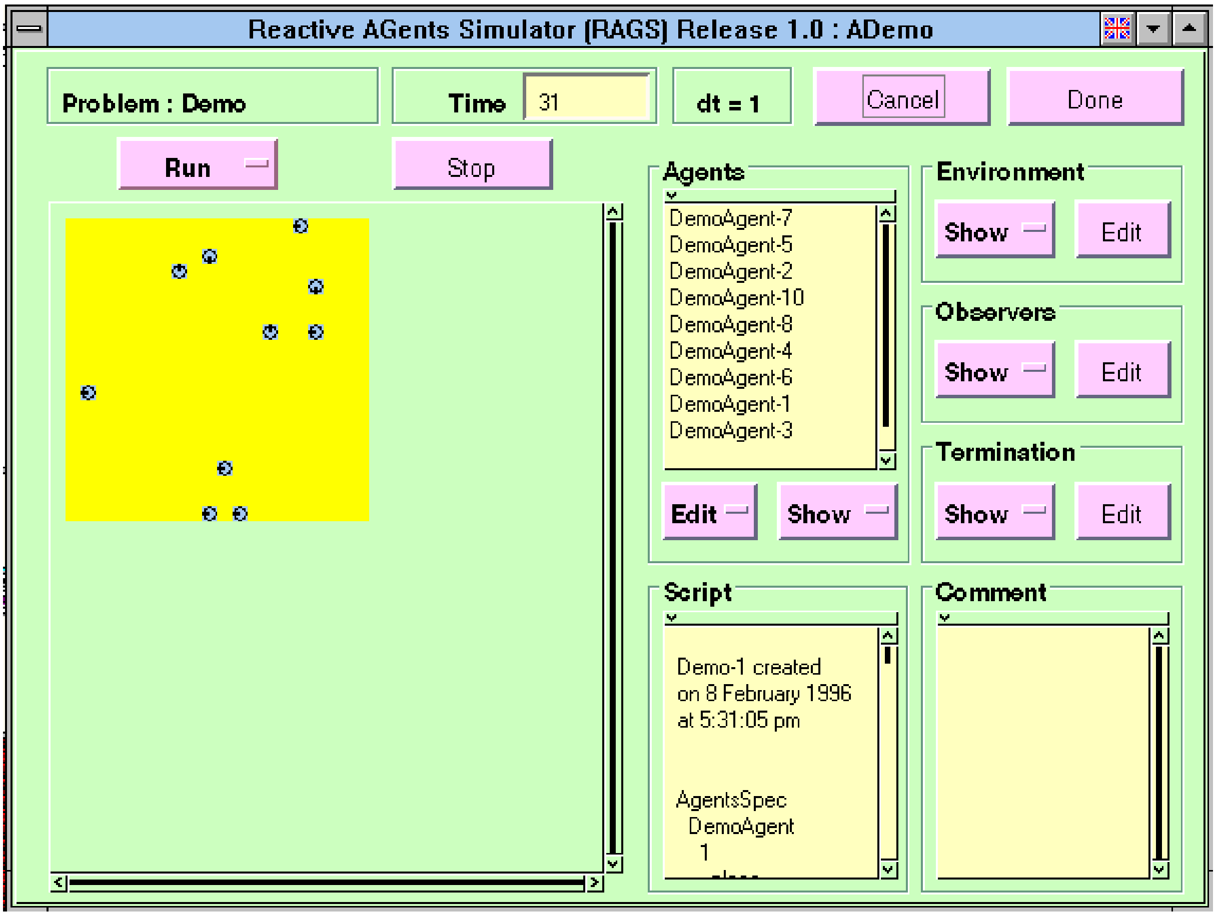

Figure 1 A snapshot of the basic RAGS simulator (instance of `ExampleEditor`)

The actual simulator is an instance of `ExampleEditor`, which is opened on a simulation (or a temporary copy of it) –that is an indirect instance of `RagsProblem`– and includes facilities for running the simulation with or without visualization on the screen, with or without tracing on a Smalltalk transcript, with periodic screen refresh or tracing etc. (fig. 1). It also includes facilities for editing and inspecting the simulation on-line, visualizing temporary statistics as textual descriptions or as charts/curves (RAGS category *Stat-Tools*) etc. The *script* instance variable is used to hold a textual description of all operations that have been performed on the simulation since its definition, including a textual description of its opening specification. Simulations not defining agent mobility, need not necessarily be visualized, unless it is for edition purposes.

The user may write and evaluate his/her own expressions to run blindly a number of simulations and output their statistics together with their specifications on a Transcript window or on a log file (see `RagsProblem` class protocol *'tests'*).

RAGS already includes editors for all types of simulation entities : agents (simple or composite, situated or not) and populations of agents, simulation objects (situated or not) and populations of them[8], simple and compound terminations, the world and its places, the different kinds of observers, as well as the specifications of all simulation entities (see *Rags-Tools* category). There is also an editor (called generator, as well) for the `RagsProblem` class (fig. 2) as well as for the `Agent` and `SimulationObject` classes and their subclasses (fig. 3) that allows the user to edit the complete simulation specifications (for instance the world and the population specification editors are visualized in figures 4 and 5). Except for the case of the RAGS worlds and their places, all other entities are inspected by opening DialogViews on textual descriptions of them.

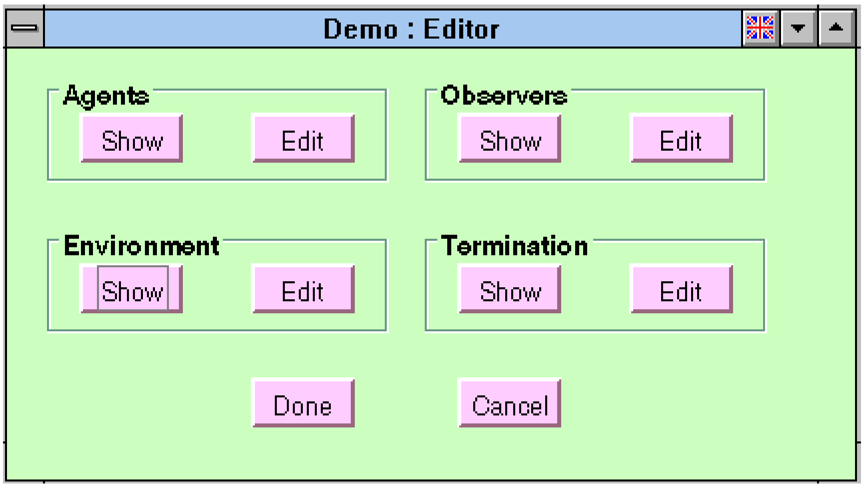

Figure 2 A snapshot of the RAGS simulation generator. It opens on a simulation or problem class (here it is the `Demo` class) and allows for redefinition/edition of the default agents/environment/observers and termination specifications.

---

[8] Agents and simulation objects are never edited directly. Instead, what is edited is their specifications and the corresponding agents/objects are subsequently re-initialized on these modified specifications. This scheme allows for crucial private variables of the agents/objects to remain intact. For agents, it also means that the user modifies the parameters of their behavior but does not completely change the behavior itself.

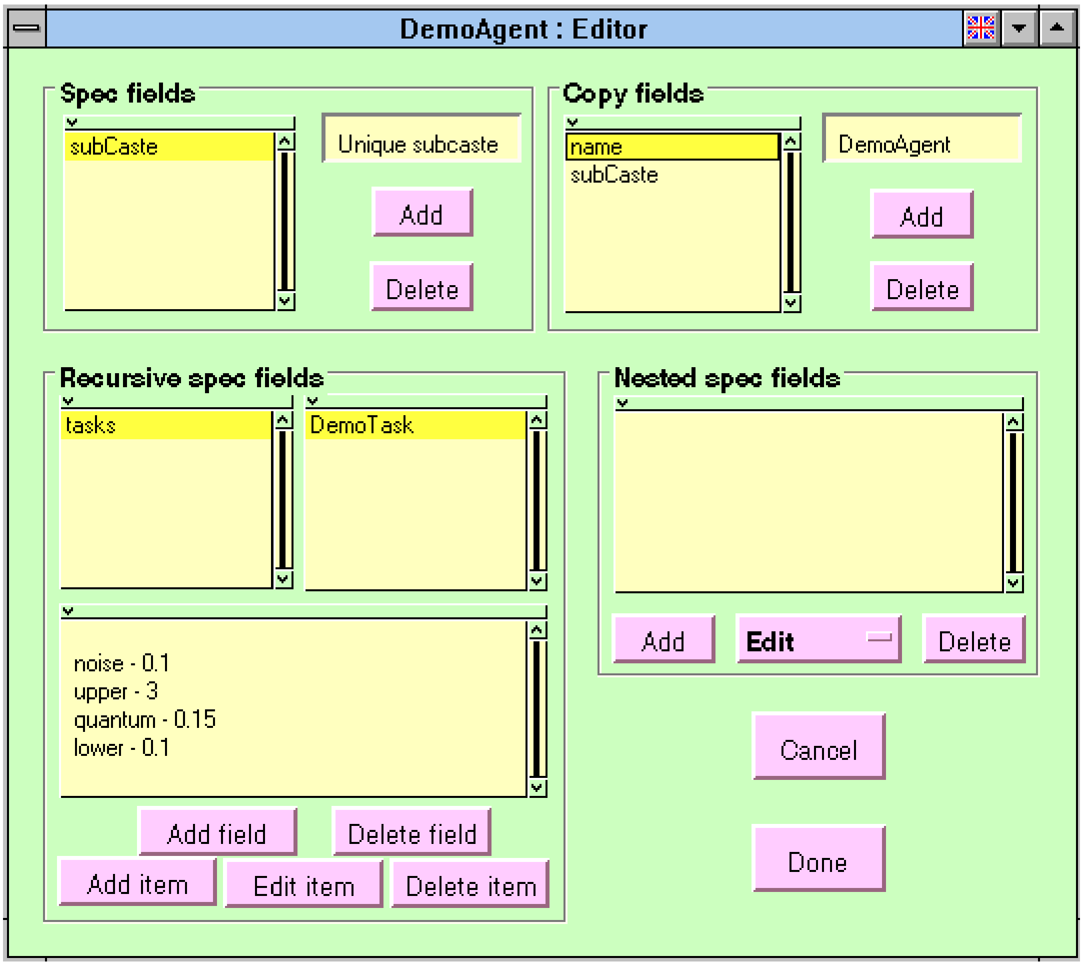

Figure 3 A snapshot of the RAGS agent generator. It supports facilities for addition/deletion of fields and for redefinition of default values.

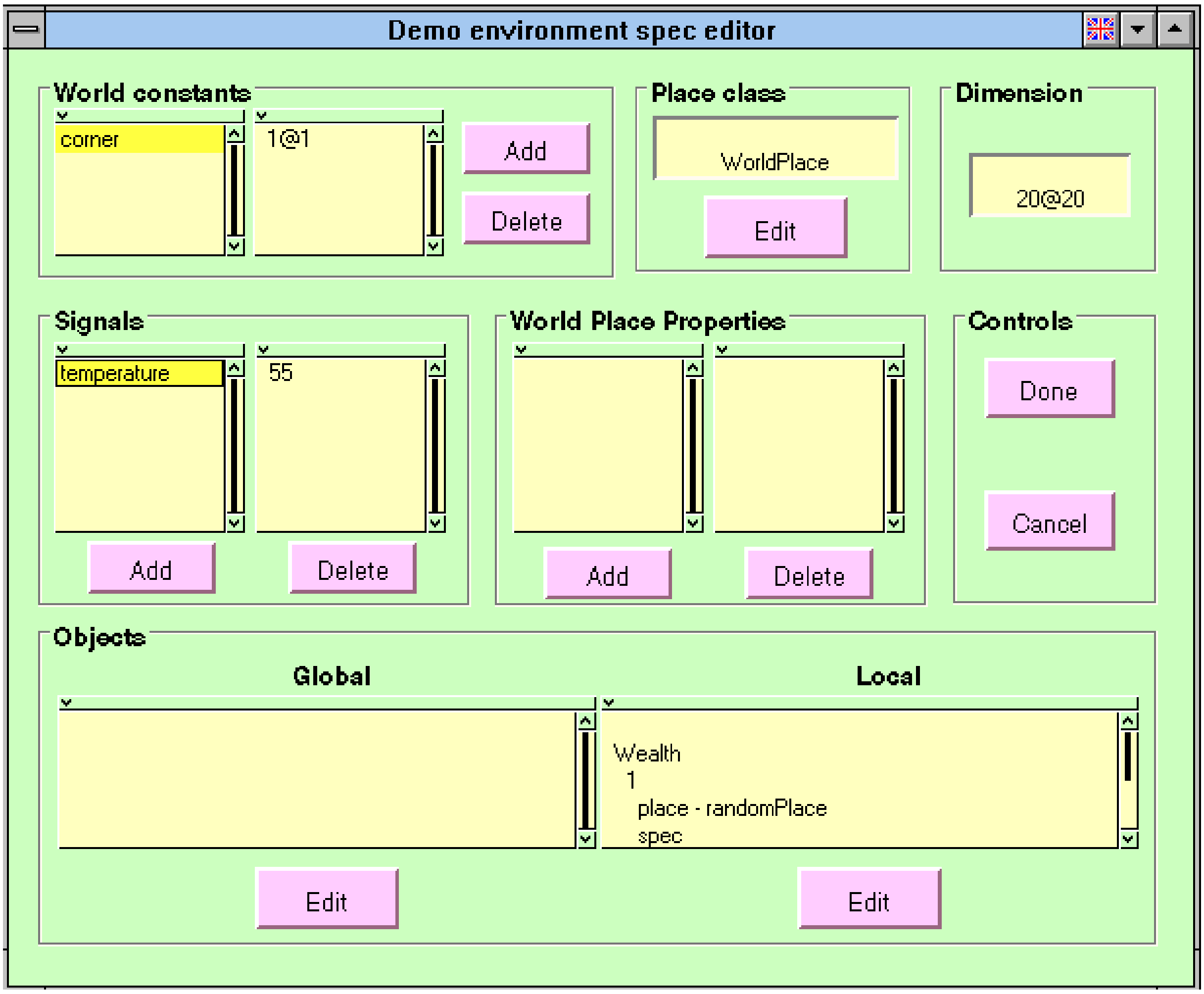

Figure 4 A snapshot of the RAGS world specification editor

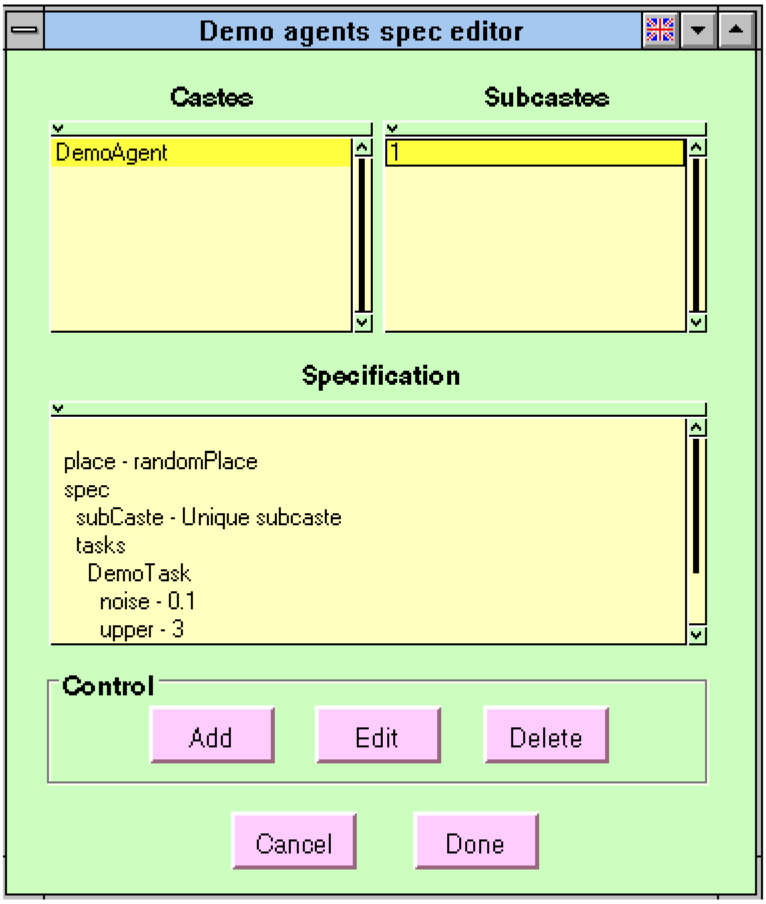

Figure 5 A snapshot of the RAGS population specification editor

8 Application development

8.1 Modeling and application specification

We are going to model and simulate a system of producers/consumers that have the peculiarity of fleeing to another place whenever the perceived wealth around them falls out of certain limits. We give below the necessary steps to follow in order to define and write the application code as well as how to run and edit experiments. The complete code for this application may be found in the *Demo* category delivered with RAGS sources. The reader is encouraged to study and test this code while constantly referring for details to the comments of the RAGS classes involved.

Let us call the class of simulations we are going to define `Demo`. The development steps of this application are summarized below.

* Firstly, we create a subclass of `RagsProblem`, `Demo`. We proceed then to define (edit) its specifications.

* *World specifications.* We assume that the simulated world is a grid of size 20x20 without any signals. We also need a way to represent the places' wealth. We define therefore a subclass of `SituatedSimulationObject`: `Wealth` with a single specifiable instance variable, `level`, representing the corresponding wealth (equivalently, we could define a property of the world places, called *wealth*). The definition and the default values of the `Wealth` class are edited with the aid of an `ObjectGenerator`, that is invoked when the expression `Wealth edit` is evaluated. The default specifications of the `Demo` class are themselves generated with the aid of a `ProblemEditor`, that is invoked when the expression `Demo edit` is evaluated (fig. 2). This latter editor invokes recursively special editors for each one of the four specifications (agents, environment, observers and termination). For instance, the environment specification below is edited with the aid of a `WorldSpecEditor` (fig. 4).

```
defaultEnvironmentSpec
"Method automatically generated."
^#(#RagsWorld
     #(#dimension #(#Point 20 20)
     #signals
          #(#RagsDictionary)
     #worldConstants
          #(#RagsDictionary)
     #objectsSpec
          #()
     #worldPlaceSpec
     #(#WorldPlace
               #(#objectsSpec
                    #(#Wealth
                         #(1
                              #(#spec
                                   #(#type #simulationObject
                                   #level 0.1)
                              #population 1
                              #place #randomPlace)))
               #properties
                    #(#RagsDictionary)))))
```

* *Agents definition and specifications.* Let `DemoAgent` be the class of producer/consumer agents. Each one of them has one single task, instance of `DemoTask`. Instances of this class of tasks should have four specifiable instance

variables, `lower`, `upper` (corresponding to the lower and upper limits of acceptable outside wealth), `noise` (the maximum distortion of wealth perception and production) and `quantum` (the maximum quantity of consumption/production). Tasks should also have three private variables, `state` (either `#produce` or `#consume`, showing what the agent will do next), `wealth` (the local accumulated wealth of the agent) and `rnd` (a random number generator, used to simulate distortion of input and output). The definition and the default values of the `DemoAgent (DemoTask)` class are edited with the aid of an `AgentGenerator`, that is invoked when the expression `DemoAgent (DemoTask) edit` is evaluated (fig. 3).

We define the default number of agents to be 10 and we assume that they are initially distributed randomly over the grid. The agents specification is generated with the aid of an `AgentsPopulationSpecEditor` (fig. 5).

```
defaultAgentsSpec
"Method automatically generated."
^#(#DemoAgent
     #(1
          #(#spec
               #(#tasks
                    #(#DemoTask
                         #(#upper 3
                         #lower 0.1
                         #quantum 0.15
                         #noise 0.1))
               #subCaste #'Unique subcaste')
          #population 10
          #place #randomPlace)))
```

If all agents perform synchronously (although at random order), agents (and tasks) do not need an internal time counter. Instead, the *elapsed:* method is actually the sense-act method :

```
elapsed: dt
     (state = #produce)
          ifTrue: [self produce; state: #consume]
          ifFalse: [    "Then, state=#consume"
                    self consume; state: #produce].
     self agent changed: #wealth with: self wealth.
     self testFlight ifTrue: [self flee]
```

The *consume*, *produce*, *flee* and *testFlight* method may be found in RAGS sources, together with additional methods used for specification edition and initialization.

* *Observers specifications.* What are the interesting aspects of the simulation to observe? We adopt two : the number of moves of each agent (let's see how social it is!) and the history of its wealth (this will be a result of its interactions with other agents). As explained in the observers' classes comments, we need to include expressions `self changed: #observerKey {with: aValue}` in the code wherever these values change or, in any case, wherever we think it might be interesting to observe these values. Therefore, we include the expression `self agent changed: #move` in the *flee* method and the expression `self agent changed: #wealth with: self wealth` in the *elapsed:* method.

```
defaultObserversSpec
"Method automatically generated."
^#(#CompoundObserver
     #(#Agents
          #(#CollectionObserver
```

```
            #(#aspect #agentsAtCaste:
            #referentialAspect nil
            #displayIndividuals false
            #singleSpec
                #(#CompoundObserver
                    #(#referentialAspect nil
                    #Moves
                        #(#Enumerator
                            #(#aspect #move
                            #referentialAspect nil
                            #initialValue 0))
                    #History
                        #(#HistoryRecorder
                            #(#aspect #wealth
                            #referentialAspect nil))))
            #parameter #DemoAgent))
    #referentialAspect nil))
```

* *Termination specifications.* We define the simplest termination criterion : *simulation time = 2000.*

```
defaultTerminationSpec
"Method automatically generated."
^#(#Termination
    #(#aspect #currentTime
    #value 2000))
```

Some extra code for specification edition, defaults etc. is necessary (see the RAGS sources).

To start and run a new simulation once the application is programmed, we need to evaluate in a workspace the expression (see also the method *startDemo*) :

```
|x|
x := Demo new initializeDefault; name: #Whatever.
x edit
```

On opening of the RAGS simulator, the user may experiment with the various buttons and menus to get a feel of the system's facilities. For instance, run the simulation for 100 cycles blindly (so that it goes faster –the display will be refreshed in the end of those 100 cycles, however) by pressing the button *Run*, choosing the menu options *run for period* and *silently* and writing "100" to the dialog view asking for the number of cycles. Then choose one of the agents from the list of agents and inspect its wealth history : press the button *Show*, choose the menu options *selected agent*, *statistics* and *as Charts*, then choose *History* in the statistics chooser. You will see the curve showing the evolution of the selected agent's wealth.

To run a number of experiments and redirect the resulting statistics on the Smalltalk transcript or on a file, use the methods defined in the `RagsProblem` class protocol *'tests'*; for example, evaluate the following expressions :

```
    Demo run: 2 withDisplayEvery: 500.
    Demo runOut: 5 on: 'test' withDisplayEvery: 500.
```

8.2 Advanced issues

* *Defining agents of variable speeds.* Each agent/task should have an additional specifiable instance variable, `threshold`, and a private one, `internalTime` (initialized to `threshold`). The *elapsed:* method should be renamed as, say, *execute* and the new *elapsed:* method should be :

```
elapsed: dt
     internalTime:=internalTime-dt.
      (internalTime<=0)        ifTrue:        [self        execute.
      internalTime:=threshold]
```

* *Modifying the simulation step.* For example, modifying the step so that one particular agent executes always first, or in a way that all agents execute in decreasing order of wealth. The default step method in the `RagsProblem` *'simulation'* instance protocol has to be redefined :

```
step
     "Among all agents, there is one (the leader) always playing
first.
     By default, the leader is the first agent in the list."
     self currentTimeIncrement.
     self leader elapsed: dt.
     (RagsEntity shuffleArray: self subordinates)
          do: [:agt| agt elapsed: dt]

step
     "Agents execute in order of decreasing wealth
     (rich agents are more equal than the others!!)."
     self currentTimeIncrement.
     (SortedCollection withAll: self agents sortBlock:
               [:agt1 :agt2| (agt1 tasks asArray first wealth) >
                         (agt2 tasks asArray first wealth)])
          do: [:agt| agt elapsed: dt]
```

* *Using common random numbers* to reduce statistical variance of results between two alternative configurations. An example experimentation protocol follows.

```
testCommonRandoms: n
      "Introduce variances in agents' parameters, but keep common
      random numbers between two runs (on each run, the agents
      will be distributed differently on the grid).
      Test with n agents"

      |rnd minL minU minN minQ maxL maxU maxN maxQ commons j
      pb1 pb2 dict|
      rnd := Random new.
      minU := DemoTask minUpper.
      maxU := DemoTask maxUpper.
      minL := DemoTask minLower.
      maxL := DemoTask maxLower.
      minQ := DemoTask minQuantum.
      maxQ := DemoTask maxQuantum.
      minN := DemoTask minNoise.
      maxN := DemoTask maxNoise.

      "Initialize the array of the common random numbers.
      Need 4 randoms per agent."
      commons := Array new: (4*n).    j := 1.
      1 to: n do: [:i|
          commons
               at: j put: minL + ((rnd next) * (maxL - minL));
               at: (j+1) put: minU + ((rnd next) * (maxU - minU));
               at: (j+2) put: minN + ((rnd next) * (maxN - minN));
               at: (j+3) put: minQ + ((rnd next) * (maxQ - minQ)).
          j := j+4].

      "Initialize the two runs."
      pb1 := self new initializeFromSpec: (self agentsSpec: n);
               name: #Run1.
      pb2 := self new initializeFromSpec: (self agentsSpec: n);
```

```
            name: #Run2.
    (Array with: pb1 with: pb2) do: [:pb|     |i|
       i := 1.
       pb agents do: [:agt|
            (agt tasks asArray first)
                 lower: (commons at: i);
                 upper: (commons at: (i+1));
                 noise: (commons at: (i+2));
                 quantum: (commons at: (i+3)).
            i := i + 4]].
    dict := Dictionary new.
    dict at: #Run1 put: pb1 run; at: #Run2 put: pb2 run.
    ^dict
```

* *Modifying the editor's menus.* Now, assume that we want to add to the simulator an option to display wealth of multiple agents on a single view (to compare them, for instance). It is logical to add the new option to the agents "Show" menu, so we redefine the corresponding menu (from the class protocol *'resources'*) either by using VisualWorks' menu editor or (more tedious) by hand. Finally, we should add to the demo's *'actions'* protocol the corresponding *showAgentsWealthIn:* method.

```
agentShowMenu
     "UIMenuEditor new openOnClass: Demo andSelector: #agentShowMenu"

     <resource: #menu>
     ^#(#Menu #(
               #(#MenuItem
                    #label: 'selected agent'
                    #submenu: #(#Menu #(
                              #(#MenuItem
                                   #label: 'structure')
                              #(#MenuItem
                                   #label: 'state')
                              #(#MenuItem
                                   #label: 'contents')
                              #(#MenuItem
                                   #label: 'statistics'
                                   #submenu: #(#Menu#(
                         #(#MenuItem
                              #label: 'asText')
                         #(#MenuItem
                              #label: 'asCharts'))
                         #(2)
                    #(#showAgentStatistics
#visualizeAgentStatistics)))
                              #(#MenuItem
                                   #label: 'spec'))
                              #(5)
                              #(#showAgentStructure #showAgentState
                              #showAgentContents nil #showAgentSpec)))
               #(#MenuItem
                    #label: 'all agents')
               #(#MenuItem
                    #label: 'agents wealth'))
               #(2 1)
               #(nil #showAllAgents #showAgentsWealth))
decodeAsLiteralArray


showAgentsWealthIn: anEditor
     |ar1 ar2 i|
     ar1 := Array new: self agents size.
     ar2 := Array new: (ar1 size).
     i := 1.
     self agents do: [:agt|
```

```
            ar1 at: i put: agt name.
            ar2 at: i put: (agt findCurveFor: #wealth).
            i := i + 1].
    anEditor
        selectAndVisualizeCurves: ar2
        labels: ar1
        label: ('A wealth story!!')
        selectMessage: 'Choose agents'
        messageIfAbsent: 'No "wealth" observers.'
```

To introduce more advanced simulator features (such as new instance variables, more elaborate canvases etc.), the first step would be to create a subclass of the basic simulator class (`ExampleEditor`), say `DemoEditor`. Furthermore, we should define the default editor class for `Demo` to be `DemoEditor`.

```
editor
    "In Demo 'defaults' protocol (eventually!)."
    ^DemoEditor
```

Test the new demo and the modified agents show menu to see what it gives!

More sophisticated extensions may be envisaged at the system, rather than the application, level : defining new observers and statistics classes, defining multiple open worlds (as abstractions of the physical world, for example), customizing graphic programming interfaces (for example, defining standard experimentation protocols and creating a graphic environment for experimenting with just these values). In all those cases, the programmer will have to study in greater depth the implementation details of the RAGS system, starting from the RAGS system comments. RAGS remains an open, experimental development system.

Annotated bibliography

G. Booch (1991). *Object-oriented design with applications*, Benjamin/Cummings, Redwood City, CA, 1991. (The most unusual of all object-oriented programming and design books. Starts with an unexpected introduction of the field as another child of the science of complexity, proceeds with a theoretical presentation of the approach, discussing such features as classification and incrementality, and ends with a number of applications on several of the most well-known object-oriented programming languages. Contains an extensive, classified bibliography)

J. R. Bourne (1992). *Object-oriented engineering - Building engineering systems using Smalltalk-80*, IRWIN & Aksen Assoc. Inc. Publishers, 1992. (Includes several chapters on engineering applications, with emphasis on topics such as constraints, reusability, rapid prototyping and simulation)

T. Bouron, J. Ferber, F. Samuel (1990). MAGES : A multi-agent testbed for heterogeneous agents, pp. 221-239, *Decentralized A.I. 2, Proceedings of the Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '90)*, Y. Demazeau & J.-P. Müller (Eds.), North-Holland, 1991. (Description of a multi-agent simulation testbed written in Smalltalk-80 and emphasizing on communication structures between heterogeneous agents)

T. Budd (1991). *An introduction to object-oriented programming*, Addison-Wesley, Reading, MA, 1991. (An excellent introduction to object-oriented programming, based on a comparative presentation of Object Pascal, Objective-C, C++ and Smalltalk. Explains elegantly advanced issues, such as object polymorphism and multiple inheritance and includes a long bibliography and source code for several examples. Overall highly recommended textbook)

E. Charniak, C. K. Riesbeck, D. V. McDermott, J. R. Meehan (1987). *Artificial intelligence programming*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987. (The classic treatise on programming techniques useful to artificial intelligence researchers)

B. Cox (1986). *Object-oriented programming - An evolutionary approach*, Addison-Wesley, Reading, MA, 1986. (The classic tutorial on object-oriented programming. Based on Objective-C. Presents the evolutionary path that brought programming to OOP. Probably the best textbook to start with)

E. H. Durfee, T. A. Montgomery (1989). MICE : A flexible testbed for intelligent coordination experiments, pp. 25-40, *Proceedings of the Ninth Workshop on Distributed Artificial Intelligence*, 1989. (Description of MICE. Various utilities to support comparative experimentation with different applications. RAGS is grateful to this discussion of requirements)

J. Ferber, A. Drogoul (1992). Using reactive multi-agent systems in simulation and problem-solving, pp. 53-80, *Distributed Artificial Intelligence : Theory and Praxis*, by N. M. Avouris & L. Gasser (Eds.), 1992. (Contains a discussion of multi-agent simulation as opposed to traditional simulation techniques together with a few examples)

L. Gasser, C. Braganza, N. Herman (1987). MACE : A flexible testbed for distributed DAI research, pp. 119-152, *Distributed Artificial Intelligence*, M. N. Huhns (Ed.), Pitman Publishing/Morgan Kaufmann, London/San Mateo, California, 1987. (A description of MACE, one of the very first multi-agent computing environments, close to the object-oriented philosophy and with a lot of user facilities)

L. Gasser, C. Braganza, N. Herman (1988). Implementing distributed AI systems using MACE, pp. 445-450, *Readings in Distributed Artificial Intelligence*, A. H. Bond & L. Gasser (Eds.), Morgan Kaufmann, San Mateo, California, 1988. (MACE at work : What agents know, sense and do)

A. Goldberg, D. Robson (1983). *Smalltalk-80 : The language and its implementation*, Addison-Wesley, Reading, MA, 1983. (The bible of all Smalltalk-80 programmers)

A. Goldberg (1983). *Smalltalk-80 : The interactive programming environment*, Addison-Wesley, Reading, MA, 1983. (Bible, part 2)

I. Graham (1991). *Object-oriented methods*, Addison-Wesley, 1991. (Includes two chapters on databases and object-oriented databases, a discussion of CASE tools and an extensive bibliography)

A. M. Law, W. D. Kelton (1991). *Simulation modeling and analysis*, McGraw-Hill, 1991. (A lot on simulation techniques (discrete-event or distributed), modeling of complex systems, performance evaluation and data analysis with examples from industrial engineering)

T. Maruichi, M. Ichikawa, M. Tokoro (1989). Modeling autonomous agents and their groups, pp. 215-234, *Decentralized A.I., Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '89),* Y. Demazeau & J.-P. Müller (Eds.), Elsevier/North-Holland, 1990. (Description of a multi-agent environment with emphasis on language constructs and communication)

Objectworks\Smalltalk Release 4.1 User's Guide, ParcPlace Systems, 1992.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object-oriented modeling and design*, Prentice Hall, Englewood Cliffs, NJ, 1991. (A comprehensive introduction to object-oriented modeling and design with emphasis on issues such as abstraction, dynamic and functional modeling distribution and structured development)

O. Tanir, S. Sevinc (1994). Defining requirements for a standard simulation environment, *IEEE Computer*, February 1994, pp. 28-34. (Although not specifically about multi-agent simulation, the authors address some important issues on models, abstraction, verification, specification, extensibility etc.)

E. Tello (1986). *Object-oriented programming for artificial intelligence -A guide to tools and system design*, Addison-Wesley, Reading, MA, 1989. (Emphasizes on existing tools and their applicability to various artificial intelligence applications)

VisualTalk Release 2.0 User's Guide, ParcPlace Systems, 1994.

A Bugs report*

"Don't blame me for the bug in your program.
I only proved it correct, I didn't try it."
- *Donald Knuth*

1 Statistics

The curves and bar charts as visualized through the simulator (that is, from within an instance of `ExampleEditor`) are static (they are not updated as the observers they visualize evolve). This is due to historical reasons: initially those tools have been developed to visualize *final* results, i.e. results after a simulation was over, so no special care had been taken about dynamicity. A future version should include dynamic statistics visualization utilities as well as advanced processing routines for results gathered (for instance, correlation analysis of time series).

2 Generalizing the applicability of the specification mechanism

It would be useful to specify results (to be displayed either on a Transcript or on a file). Such a specification would save rewriting tedious experimentation class methods, as well as saving unnecessary information from being displayed. This specification would be initiated by selecting statistics, display intervals, sub-sets of agents etc. As with results, it would also be useful to have specifications of the structure/state/contents fields of agents/simulation objects, so as to avoid rewriting methods such as the *localStateString* in every class.

3 Generalizing the recursive structure of all entities

A future version will make no distinction between objects, agents and simulation. Everything will be an entity capable of including recursively other entities, themselves being complex in structure as well. Differences will show in the actual behaviors of different entities, while structures will only serve as descriptions.

---

* *Those problems will be tackled during the development of the next version of the RAGS system.s*

B The RAGS classes

/*   Rags-Entities category   */

#RagsEntity subclass: #DisplayableEntity
#RagsEntity subclass: #GenericEntity
#ObservedEntity subclass: #NamedEntity
#DisplayableEntity subclass: #ObservedEntity
#Model subclass: #RagsEntity

/*   Rags-Utilities category   */

#DisplayableEntity subclass: #CompoundTermination
#Object subclass: #GenericUtility
#Specification subclass: #PopulationSpecification
#Dictionary subclass: #RagsDictionary
#Random subclass: #RagsRandom
#RagsDictionary subclass: #RecursiveDictionary
#RecursiveDictionary subclass: #Specification
#DisplayableEntity subclass: #Termination

/*   Rags-Statistics category   */

#StatRecord subclass: #AvgStatRecord
#AvgStatRecord subclass: #CounterStatRecord
#CounterStatRecord subclass: #DurationStatRecord
#Object subclass: #InflatableArray
#RecursiveDictionary subclass: #Statistics
#Object subclass: #StatRecord
#AvgStatRecord subclass: #UnevenAvgStatRecord

/*   Rags-Observers category   */

#Enumerator subclass: #ActivityDurationRecorder
#Observer subclass: #CollectionObserver
#Observer subclass: #CompoundObserver
#Enumerator subclass: #DirectAdder
#DirectAdder subclass: #DirectAverager
#VariableObserver subclass: #Enumerator

#VariableObserver subclass: #HistoryRecorder
#Enumerator subclass: #IndirectAdder
#IndirectAdder subclass: #IndirectAverager
#DisplayableEntity subclass: #Observer
#Enumerator subclass: #PlaceVisitorsObserver
#Observer subclass: #VariableObserver
#VariableObserver subclass: #VariableObserverOnce

/* Rags-Environment category */

#SituatedSimulationObject subclass: #Obstacle
#ObservedEntity subclass: #RagsWorld
#Source subclass: #Sample
#RagsEntity subclass: #SimulationObject
#SimulationObject subclass: #SituatedSimulationObject
#SituatedSimulationObject subclass: #Source
#WorldPlace subclass: #TracedPlace
#ObservedEntity subclass: #WorldPlace

/* Rags-Agents category */

#ObservedEntity subclass: #Agent
#Agent subclass: #CompositeAgent
#SimpleAgent subclass: #IndivisibleTask
#Agent subclass: #SimpleAgent
#CompositeAgent subclass: #SituatedCompositeAgent
#SimpleAgent subclass: #SituatedSimpleAgent

/* Rags-Problems category */

#NamedEntity subclass: #RagsProblem

/* Rags-Tools category */

#ObjectGenerator subclass: #AgentGenerator
#PopulationSpecEditor subclass: #AgentPopulationSpecEditor
#RagsEditor subclass: #AgentsEditor
#RagsDialog subclass: #AgentsPopulationViewer
#RagsEditor subclass: #CollectionObserverSpecEditor

#SimpleSpecEditor subclass: #CompositeSpecEditor

#RagsEditor subclass: #CompoundObserverSpecEditor

#RagsEditor subclass: #CompoundTerminationEditor

#CompoundTerminationEditor subclass: #CompoundTerminationSpecEditor

#RagsEditor subclass: #ExampleEditor

#RagsDialog subclass: #ListViewer

#RagsEditor subclass: #ObjectGenerator

#PopulationSpecEditor subclass: #ObjectPopulationSpecEditor

#RagsDialog subclass: #PlaceViewer

#RagsEditor subclass: #PopulationSpecEditor

#RagsEditor subclass: #ProblemEditor

#RagsDialog subclass: #RagsChooser

#SimpleDialog subclass: #RagsDialog

#ApplicationModel subclass: #RagsEditor

#RagsDialog subclass: #RagsFieldEditor

#UniqueInstanceEditor subclass: #RagsManager

#RagsEditor subclass: #RagsPlaceEditor

#RagsEditor subclass: #RagsWorldEditor

#SpecificationEditor subclass: #SimpleSpecEditor

#RagsEditor subclass: #SimpleTerminationEditor

#SimpleTerminationEditor subclass: #SimpleTerminationSpecEditor

#CompositeSpecEditor subclass: #SituatedCompositeSpecEditor

#SimpleSpecEditor subclass: #SituatedSimpleSpecEditor

#RagsEditor subclass: #SpecificationEditor

#RagsEditor subclass: #UniqueInstanceEditor

#RagsEditor subclass: #WorldSpecEditor

#RagsDialog subclass: #WorldViewer

/* Rags-Support category */

#BoundedWrapper subclass: #RagsBoundedWrapper

#DialogView subclass: #RagsDialogView

#ApplicationStandardSystemController subclass: #RagsEntityEditorsController

#DefaultLookPolicy subclass: #RagsLookPolicy

#MenuButtonController subclass: #RagsMenuButtonController

#MenuButtonView subclass: #RagsMenuButtonView

#Process subclass: #RagsProcess

#SelectionInListView subclass: #RagsSelectionInListView

#SelectionSetInListView subclass: #RagsSelectionSetInListView

#SelectController subclass: #RagsSimulationController

#AutoScrollingView subclass: #RagsSimulationView

#TextEditorController subclass: #RagsTextEditorController

#TextEditorView subclass: #RagsTextEditorView

#ApplicationWindow subclass: #RagsWindow

/* Stat-Tools category */

#Model subclass: #Curve

#Model subclass: #CurveCollection

#Model subclass: #HorizontalLine

#RagsBarChartView subclass: #MultiBarChartView

#View subclass: #RagsBarChartView

#View subclass: #RagsCurveView

#GenericEntity subclass: #StatModel

#RagsEditor subclass: #StatViewer

/* Demo category */

#RagsProblem subclass: #Demo

#SituatedCompositeAgent subclass: #DemoAgent

#IndivisibleTask subclass: #DemoTask

#SituatedSimulationObject subclass: #Wealth