Using Mozart for Visualizing Agent-Based Simulations

Hala Mostafa and Reem Bahgat

Faculty of Computers and Information
Cairo University, Cairo, Egypt
{h.mostafa, r.bahgat}@fci-cu.edu.eg

Abstract. Scientists from various domains resort to agent-based simulation for a more thorough understanding of complex real-world systems. We developed the Agent Visualization System; a generic system that can be added to a simulation environment to enrich it with a variety of browsers allowing the modeler to gain insight into his simulation scenario. In this paper we discuss how the various features of the Oz language and the Mozart platform aided us in the development of our system. Of particular importance were dataflow variables, high-orderness, the support for distribution and concurrency, the flexibility offered by QTk which was crucial in generating browsers whose structure is only known at runtime, in addition to a miscellany of features that were conductive to our work. We also highlight some of the implementation difficulties we faced and explain the techniques we utilized in overcoming them.

1 Introduction

Domains as varied as biology and mechanical physics have resorted to Agent-Based Simulation (ABS) to capture the behavior of, and interaction between, entities in their respective systems. An agent can be thought of as a software component which not only encapsulates code and data as in object-oriented programming, but can also be pro-active, autonomous, adaptive and collaborative [14].

In ABS, a scenario of entities that interact with each other and with their environment is modeled as a multi-agent system, hence the name Multi-Agent-Based Simulation (MABS). A MABS usually involves agents of different types. Each type represents a class of entities in the real-world system and captures its relevant attributes and behaviors. Compared to simulation techniques which assume that all instances of a certain entity are alike, ABS has the advantage of being able to explicitly model the heterogeneity of real-world entities by allowing entities of the same type to differ in their attribute values and behaviors [13].

The increasing demand for MABS by scientists foreign to the field of computer science created a need for simulation environments that facilitate rapid development of MABSs. One of the main facilities that should be provided by such platforms is the ability to visualize the proceedings of a simulation scenario from different perspectives without requiring the modeler to delve into technicalities.

P. Van Roy (Ed.): MOZ 2004, LNCS 3389, pp. 89–102, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Visualization

Information visualization (IV) is a research domain that aims at supporting discovery and analysis of data through visual exploration. Its principle is to map the attributes of an abstract data structure to visual attributes such as Cartesian position, color and size [2]. IV is one means of carrying out Exploratory Data Analysis which aims at the manipulation, summarization, and display of data to make them more comprehensible to human minds, thus uncovering interesting trends and relationships. Some IV tools rely on an *interface metaphor*; they allow the user to operate on data in the same way he operates on things in real life (e.g. the lens metaphor [11] where data is displayed differently when viewed through different lenses, and the rubber sheet metaphor [8] which allows the user to stretch parts of the display, thus revealing more details)

In this paper we illustrate how we used the Oz language, the Mozart development platform [7] and the QTk graphical package [5] to implement the Agent Visualization System (AVS); the first generic, distributed system specifically dedicated to the visualization of agent-based simulation scenarios [6]. The AVS is used as an add-on to a simulation environment to equip it with a rich set of visualization facilities offering a variety of textual and graphical browsers that allow the modeler to detect trends and relationships in the simulation scenario. Some techniques from IV were adapted and added to our system, while others were devised especially to be used in it. Regardless of their origin, all visualization techniques were thoroughly revised to make them generic enough to fit in our generic system.

The structure of the paper is as follows: Section 2 discusses some of the challenges faced in visualizing agents and the requirements fulfilled by our AVS. Section 3 describes the various browsers that make up the AVS while Section 4 outlines the high-level design of our system and its usage. Section 5 discusses some implementation issues and the various Mozart features that are involved in them. Some of the difficulties we faced are also mentioned, together with how they were overcome. We conclude and briefly discuss areas for future work in Section 6.

2 Agent Visualization Challenges and Requirements

The issue of visualization is of primary importance in ABS. The modeler needs to be presented with a view of the simulation that allows him to make and verify hypotheses regarding relationships between the various entities in it. However, the task of agent visualization poses major challenges, the most notable of which is that an agent's state is continuously changing, thus the data to be visualized is dynamic rather than static. At the same time, the responsiveness of the system is a key requirement, so those changes have to be visually reflected within an acceptable time limit. The volume of data to be visualized poses another challenge since a large number of agents is typically involved in a MABS, each of which has a set of attributes that the modeler may like to observe. Moreover, the nature of simulation data is usually not restricted to variable-value pairs with which traditional IV techniques are most effective. Rather, the simulation

produces data concerning actions, events and communication between agents in addition to variable-value pairs describing agents states. A visualization system should therefore include means to visually represent this multitude of data types.

Our AVS attempts to handle these challenges. In addition, the following are some requirements that are fulfilled by our system: In order to be usable in a wide variety of MABSs, the AVS needs to be completely generic; it should not require any a priori information about the particulars of the simulation scenario being visualized (agent types, together with their attributes and attribute meta-data, should be known only at run time), nor should it make assumptions about the domain of the simulation. Another requirement concerns the flexibility of the AVS. The user should have full control over the way his agents are represented, whether textually or graphically. Moreover, the user should be able to specify which subset(s) of agents he wants to observe in any given browser by choosing agents by ID, type, or location, or by expressing interest in agents satisfying certain criteria. In order to abide by the famous IV mantra *"Overview first, zoom and filter, then details on demand"* [10], the AVS is required, at all times, to provide an overall view of the proceedings of the simulation scenario, as well as allow the user to obtain details about any part of it. In order to both promote collaboration and distribute the computational load, it is advantageous to have a distributed AVS where observers can launch browsers from remote sites just as easily as they would from a central site. Another requirement concerns how often the display of an AVS browser is refreshed. Depending on the rate at which interesting events happen in a simulation scenario, the modeler may choose to be shown every single step of the simulation. Alternatively, he may only be interested in every nth step. For any browser, the user should be able to set the value of n, with the ability to freeze a view and create static browsers which are only refreshed on demand. Finally, when running in real-time mode, the AVS can skip the visualization of some older time units in favor of newer ones. The modeler may then miss some important or rare events in the course of a simulation. The AVS should therefore be able to run in playback mode where the user can step through the simulation at leisure.

3 AVS Browsers

The AVS offers the following browsers which provide graphical and textual representations of the simulation scenario (for more details, please refer to [6]).

- **Lens browser**: in Magic Lens filters [11] the lens metaphor is used for filtration, as well as presenting alternative views, of the underlying data. Our adaptation of the lens filter allows the user to set the acceptance criteria for the lens, move it over the environment where his agents live, and only view through it those agents satisfying the criteria.
- **Labeling browser**: this browser uses *dynamic labels* illustrated in Excentric Labels [1] for agents in densely-populated areas where static labels would occlude neighboring agents. The user moves a labeling region over his area of interest and only the agents within this region are labeled (Fig-

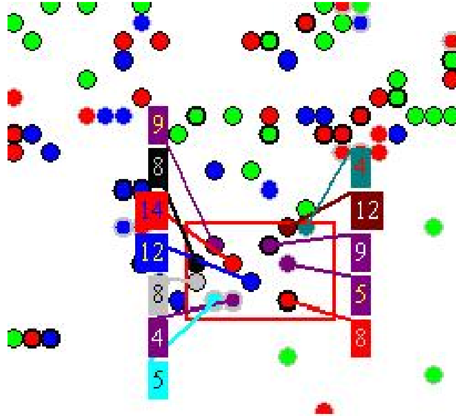

Fig. 1. Dynamic Labeling in a crowded area

ure 1). When the user moves the region, the set of labeled agents changes accordingly. Color is used to associate agents and their labels. We devised a placement algorithm to make sure that a label is as close as possible to its associated agent. In addition, the modeler can specify, for each agent type, the agent attribute whose value will be used to label agents of this type.

- **Aggregation browser**: when the region of interest is too large to be displayed in its entirety, the aggregate browser uses 1 cell to represent every N x N sub-region in the original simulation landscape, where N is specified by the user. All agents of a certain type in the sub-region appear as a single agent whose attributes are calculated from those of individual agents according to functions specified by the user (e.g. minimum, maximum, average). Choosing to expand a marquee-selected region creates a new browser showing the interesting region at normal size. The high degree of flexibility offered by Mozart made it possible to treat aggregate agents the same as normal ones; features like lens filters and dynamic labeling operate on a grid of agents, regardless of the nature of the agents inside the grid.
- **Dynamic Query (DQ) browser**: DQ is a way to dynamically and visually control the amount of data on display [4]. In a DQ display, the user executes a query on his dataset and watches the results of this query. This is done by associating with each data attribute an input control whose manipulation changes the chosen values for this attribute. The sub-queries formed by the controls are ANDed to form the overall query. In our DQ browser (Figure 2), the controls are automatically generated based on which attributes the user includes in the query (e.g. range slider controls for numeric attributes and radiobuttons for categorical ones). As the user manipulates these controls, the result pane instantly reflects the changes by showing agents that meet the current query and hiding all others.
- **Brushing-based browser**: the notion of brushing [9] is used in this browser to associate the sender of a message with its recipient(s). On clicking an agent's graphical representation, the same color is assigned to

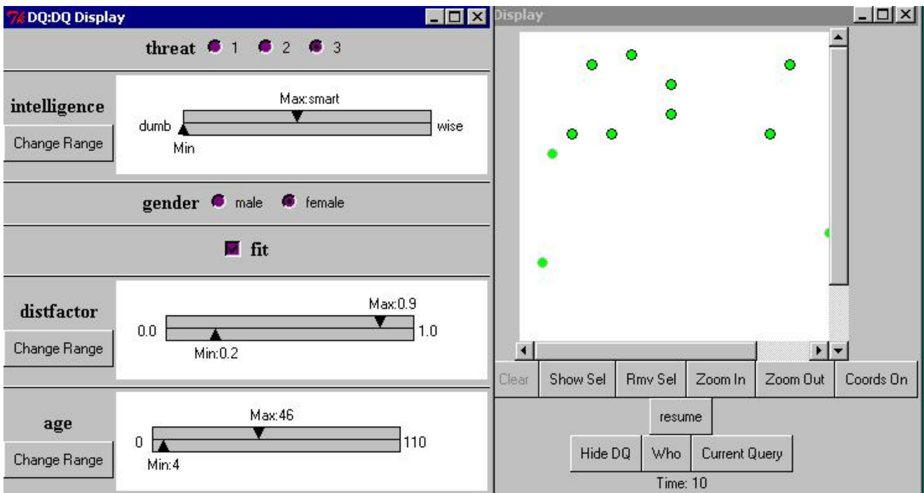

Fig. 2. A Dynamic Query browser with the controls on the left and result pane on the right

this agent and all the agents with whom he exchanged messages during the displayed time unit. A random color is assigned to the brushed entities. In order not to mislead the observer, the original colors of the agents are restored when brushing is no longer active, since color may encode an attribute value. The browser acts like a normal browser in all other respects and can be combined with any other feature.

- **Fading browser**: borrowing on ideas from Chat Circles [12], this browser makes it easier for the user to identify the most active participants in a conversation and observe the general pattern of communication. The color of an agent brightens every time he "speaks" and gradually fades during periods of silence. Optionally, a pop-up containing the message's label can appear next to an agent every time he sends a message.
- **Conversation sequence browser**: this browser focuses on conversations held among a group of agents. Inspired by Protocol Diagrams [3], we use a vertical line, called lifeline, for each agent, while messages are shown as labeled horizontal lines from sender to receiver. The user specifies the agents of interest and can later choose to add or delete a lifeline and change the message field used as a label. Clicking a message textually displays its details in a side browser.
- **Text browser**: the AVS allows the modeler to create text browsers that direct their output to either a file or a textual browser. This is convenient when the level of detail required by the user is high or when a persistent record is needed.

4 High-Level Design

In deciding whether the simulation system and the AVS should operate synchronously, we chose to allow them to go at their own paces, with the AVS operating on the most recent simulation data and discarding older data. The simulation should merely dump its output to files that the AVS can later process. These files also act as useful recordings of simulation runs, thus allowing the AVS to replay old runs.
The AVS therefore consists of the following four stages:

- **The readers** are notified by the mirror store to fetch data from files generated by the simulation system when the AVS is ready to process them.
- **The mirror store (MS)** uses input from the readers to construct a faithful replica of the simulation system at a certain point in time, thus acting as a local store for states/messages to avoid querying the simulation system every time a browser requires data. The MS also calculates values of *derived attributes* (as opposed to *raw attributes* which constitute the agent state as reported by the simulation system) using the calculation rules specified by the user (for example, the user can add the Boolean derived attribute isEligible that is true if the following conditions on raw attributes hold: (age == 18) and (gender == male)).
- **The dispatcher** is responsible for keeping track of the interests of the various browsers. In a large simulation, it would be unwise to forward data about every agent to every browser every time unit. Therefore, upon creation, a browser informs the dispatcher of its initial interests which are later used to decide which states/messages are sent to it and when. A browser can later modify its registered interests (e.g. when the user scrolls the display, the browser becomes interested in a different region of the simulation and thus updates its interests).
- **The browser** renders data received from the dispatcher depending on the graphical mapping specified for it (see next section). A browser can be *dynamic* or *static*. The MS pushes data to dynamic browsers on a regular basis whereas a static browser pulls data whenever the user explicitly asks for an updated view by pressing the refresh button.

AVS Usage

To use the AVS with a simulation system, the latter should periodically output the state of the simulation to a known location on the file system. It should also establish a socket connection with the AVS over which it sends a token at the end of every simulated time unit. When the AVS starts, it reads information about the simulation (environment dimensions, agent types, attributes, attribute types and legal values) from a simple text file. A GUI then asks the user to specify any derived attributes and the rules that will be used to calculate them from raw ones. To create a browser, the user provides a specification file containing the browser's type (e.g. text, aggregate), whether it is dynamic or static (and in the former case, the refresh rate), region of interest and criteria for displaying

agents, together with any additional features (e.g. lens filter). In the case of graphical browsers, a GUI asks the user how each agent type will be represented graphically. All these specifications can be stored to a file for later use with other browsers.

5 Implementation Issues

5.1 QTk-Related Issues

The Graphical Mapping. For all graphical browsers except the Conversation Sequence browser, the user specifies how an agent's graphical representation is calculated from its state. This is done by specifying a *graphical mapping* from the values of the attributes making up an agent's state to the values of various graphical properties of the shape representing this agent (e.g. color can depend on attribute a1 through an if-statement, width can be calculated by dividing attribute a2 by 10 and height can be determined by the average of attributes a3 and a4).

The user specifies the mapping through a GUI that uses the Tree control developed by Donatien Grolaux at the Université Catholique de Louvain. Nodes at the first level of the tree are associated with graphical properties of the shape chosen to represent agents of the type in question (Figure 3). Our customized node inherits from Grolaux's TreeNode to extend it with an awareness of what kind of node this is (property, expression, if, then, end or elseif). Each of these kinds responds differently when clicked. For example, clicking a 'property' node results in a dialog box asking whether the value should be calculated using a constant, a mathematical expression involving constants and agent attribute names or an if-statement. If the user chooses an if-statement, a sub-tree of if-then-else/end/elseif nodes is attached to the 'property' node. On clicking an 'if' node, the user is asked to enter a Boolean expression. Clicking 'then' and 'else' nodes allows the user to either enter an expression, or choose to have a further level of if-statements, in which case a new sub-tree of if-then-else/end/elseif nodes is inserted.

The final form of the entire tree is parsed into a nested Oz record which, when it is time to draw an agent of the type in question, is evaluated by replacing attribute names with actual values of the agent's attributes and carrying out the operations specified by the sub-records. The result is a set of values used to set the various properties of the shape representing the agent.

Implementing Dynamic Queries. QTk's flexibility played an important role in the dynamic generation of input controls for the DQ browser. Because QTk allows the programmer to specify the window structure at run-time, it was possible to have the DQ browser consult the meta-data of the attributes forming the query and construct the DQ window accordingly (i.e. decide on the types of input controls based on the query attributes). However, we faced a certain difficulty: the controls for which DQ was most famous are missing in QTk. Range-sliders and alpha-sliders, used to manipulate numerical and ordered string attributes,

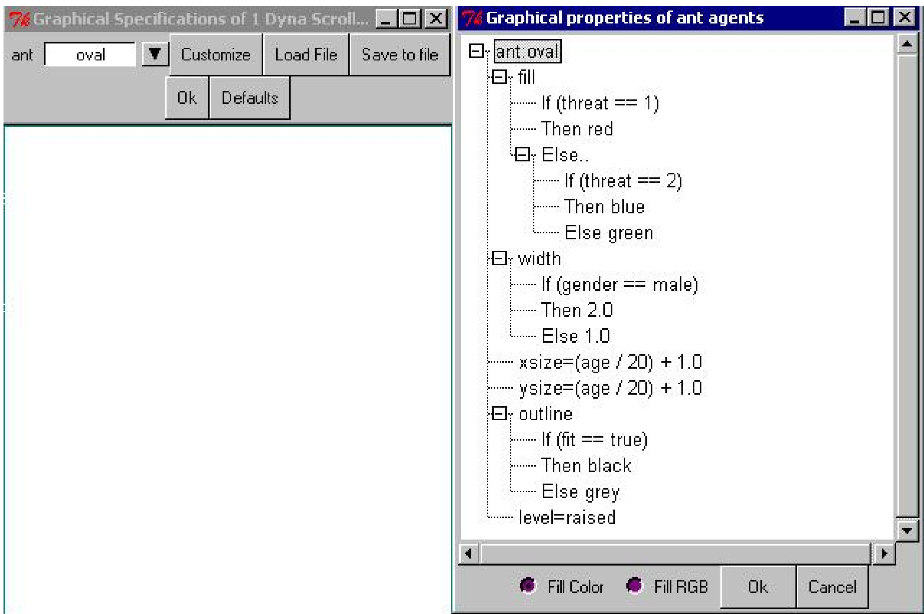

Fig. 3. The GUI for specifying the graphical mapping

respectively, are not part of QTk's repertoire. We therefore used QTk's canvas items to hand-craft a general slider control that can be used as a range-slider if initialized with a range of numbers, or an alpha-slider if initialized with an ordered list of strings.

We used the *callback* technique to allow the slider to call methods in the outside world when certain user actions take place. This was greatly facilitated by Mozart's high-orderness; the callback procedures can be passed as arguments to slider methods. These procedures are invoked when the user moves the maximum and minimum pointers, represented by two triangles that respond to the appropriate mouse events. The trough of the slider is a rectangle whose tag responds to mouse clicks by making the label and pointer jump to where the mouse is. The assembled whole was wrapped in a class to hide the numerous details while offering methods that are typically supported by slider controls (e.g. setting the range of the slider, the increment of the pointers and, in case of an alpha-slider, the list of ordered strings that acts as a value source).

Controls available in QTk were not used for DQ as-is. Each one was wrapped in a class that is aware of the attribute it manipulates, the values it can take, and the message that should be sent to the browser every time the user manipulates it. These messages, expressing sub-queries, are used by the browser to determine which agents will be added/removed to/from the result set. We therefore avoid re-processing the entire new query and only consider "delta"; the part of the query that changed as a result of the user's last action. We believe this greatly improved the performance of the DQ browser.

Wrapping QTk's Tags. QTk's canvas tags are very basic entities that allow simple manipulation (e.g. scaling, moving, deletion) but do not provide for more advanced manipulation (e.g. resizing, selection, dragging). To overcome this shortcoming, it was necessary to create bindings to associate certain events with certain actions to achieve the desired effects (e.g. bind the left-mouse-down event to an action that makes the width of the tag's outline thicker in order to give the effect of being selectable). We therefore implemented BaseTag which is a class that encapsulates QTk's tag and keeps track of things like tagID, scale and tag size in logical units. Also implemented are the mixin classes Resizeable, Dragable, Selectable and Marquee, each of which can be mixed with BaseTag to produce hybrid tags exhibiting a set of behaviors.

Callback functions were again extensively used. For example, in the lens and dynamic labeling browsers, the lens and labeling region are instantiated from a mixture of BaseTag, Resizeable and Dragable and initialized with callback functions that should be called in cases of resizing and dragging. The lens's callback function receives the new lens region as a parameter and uses it to apply the filter to agents in this region. The labeling region's callback function labels agents who have just gotten inside the region and removes the labels of those who have moved out of it. In the case of the aggregate browser, marquee selection uses a tag instantiated from BaseTag and Marquee whose callback function updates the value of an attribute called RegionToExpand.

5.2 Synchronization-Related Issues

Synchronizing Data Rendering. As mentioned earlier, the AVS fetches simulation data from files generated by the simulation system when it is ready to process them. To declare its readiness to receive a new batch of data, the mirror store needs to make sure that all browsers have finished rendering the previous batch. But the browsers render their data concurrently. Therefore, a browser needs a way of finding out whether it is the last one to finish rendering, in which case it should notify the mirror store so that the latter can fetch more data.

Dataflow variables markedly facilitated handling this issue. The following pseudocode shows what the dispatcher needs to do when it sends data to browsers:

```
proc {SendToAll BrowsersList Token}
   case BrowsersList of [Browser] then
      {SendTokens Browser Token finishToken}
   elseof Browser|OtherBrowsers then
      declare T to be a free variable
      {SendTokens Browser Token T}
      {SendToAll OtherBrowsers T}
   end
end
```

The initial call to SendToAll uses the value beginToken as Token. The following variant of the famous Test-and-Set technique forms part of any browser, with a Mozart lock to ensure its atomicity:

```
fun {TestAndSet T1 T2}
   start atomic section
      if T1 and T2 are both bound return true
      else
         T1 = T2
         return false
      end
   end atomic section
end
```

When a browser finishes rendering its data, it sends its 2 tokens to the TestAndSet function and notifies the mirror store if the function returns true. Therefore, as they proceed concurrently, browsers unify the two tokens passed to them. The last browser to finish will find that both tokens are bound, and can then notify the mirror store that all browsers are done.

The Readers-Writers Problem. Operations on the mirror store are divided into write operations (simulation data is fetched from files and written to the store) and read operations (extracting the interests of browsers). The store is in an inconsistent state while it is being written to because part of it reflects the state of the simulation system at a certain time, while another part reflects the state of an earlier time. The programmer is therefore faced with a readers-writers problem.

To synchronize the various readers and writers, a Semaphore class was implemented using dataflow variables and the built-in locking property of classes. Three semaphore instances are needed: a readers mutex, a writers mutex and a readers-writers semaphore. The scheme is as follows:

- The readers mutex is used to guarantee mutual exclusion during incrementing the counter in the following pseudocode (note that the first reader/writer waits on the readers-writers semaphore and the last one signals it):

  ```
  NumberOfReaders := NumberOfReaders + 1
  If NumberOfReaders == 1 then
     wait on the readers-writers semaphore
  ```

  The same goes for these steps:

  ```
  NumberOfReaders := NumberOfReaders - 1
  If NumberOfReaders == 0 then
     signal the readers-writers semaphore
  ```

- The writers mutex is used in a similar fashion. The difference between the classical readers-writers problem and ours is that in our case, writers can work in parallel since they write to non-overlapping regions of the store.
- The readers-writers semaphore is used to make sure that when one or more writers are active, readers are blocked, and vise versa. It is waited on by the first reader or writer, and signaled when the last reader or writer finishes.

The Canvas Mutual Exclusion Problem. A graphical browser draws on a QTk canvas that is accessed by more than one method for rendering, clearing and destroying it. Without proper precautions, these methods can undesirably interfere with each other's actions. For example, if clearing takes place during rendering, only the part that was rendered will be cleared, while other parts continue to have graphical entities and yet will be considered clear. If the canvas is destroyed while it is being drawn to, attempts will be made to access a non-existent canvas resulting in an error. For these reasons, it is necessary to have a canvas lock shared by all concerned parties. Two dataflow variables are also needed; shouldClose indicates when the user wants to close the browser, and canClose indicates when the canvas can actually be closed.

The pseudocode for the three relevant methods in a graphical browser is as follows (the methods for clearing and rendering are written as one):

```
meth clear/Render
   free canClose
   obtain canvas lock
      for all items to clear/render
         if shouldClose then break
         clear/render item
      end
   release canvas lock
   bind canClose
end
meth RequestClose
   shouldClose = true
end
```

An additional method is needed to allow an outside entity to know whether it is ok to close a browser, thus destroying its canvas. The method is blocking; it waits until it is ok to close the browser then returns.

```
meth Closable
   wait on canClose
end
```

The combined use of dataflow variables, locks and the loop breaking mechanism (through the 'break' loop feature), all of which are provided by Mozart, allowed the development of this simple, yet effective synchronization solution.

5.3 Miscellaneous Issues

The Use of Mixin Classes. The classes implementing the various AVS browsers do not fit into a single inheritance hierarchy. Instead, a basic behavior is enhanced upon using inheritance, with possible add-ins in the form of *mixin classes* implementing features like the lens filter and DQ. This arrangement has a twofold advantage: adding a new feature is greatly simplified, since we only need to inherit from the appropriate class(es) and add methods realizing the new feature. In addition, combinations of classes can be created at run-time based on browser

specifications. This precludes the need to have static classes for all possible combinations of browser types and features. Thus no matter how many new features we incorporate, it is left to the run-time combination of classes to add these features to the appropriate base browser types as desired.

Distribution Issues. As mentioned before, the dispatcher sends interesting data to the various browsers. To enhance uniformity, both local and remote browsers create ports for themselves and register these ports with the dispatcher. This has the advantage of making distribution transparent to the dispatcher, thus allowing it to send messages on the ports without worrying where the browsers actually reside.

One of the difficulties faced in allowing the dispatcher and browsers to reside on different sites is the problem of passing objects between sites. In the Mozart version we used (1.2.4), some of the commonly needed Mozart entities are sited, whereas the Abstract Data Types (ADTs) that we use to encapsulate states and messages depend on Mozart's Dictionary and Array which are both sited. To get over this problem, the ADTs were extended with a method that returns the object's contents in a simple unsited form (a record or a list). Those unsited entities are sent to the -possibly remote- browsers who can later reconstruct the original ADTs.

Improving Responsiveness. Throughout the AVS, various objects respond to various user actions. For example:

1. Scrolling results in data being pulled from the dispatcher (data is pulled lazily when a new region is revealed by scrolling or zooming out).
2. Moving a lens results in some agents being hidden (filtered out) and others being shown.
3. Manipulating DQ controls changes the number of agents on display.

Because the user can perform actions in rapid succession, we should consider whether it is necessary to respond to every single action. There are three alternatives matching the above three cases:

1. Queue the actions and process them in a First-Come-First-Served manner, but before processing an action, check whether it is still relevant and if not, disregard it. This is suitable in case 1 where the display should not scroll to a region that was already scrolled out of view by a subsequent scrolling action. Note though that in this case, new actions should not automatically overwrite older ones.
2. Process only the most recent action (new actions should overwrite older ones). This is suitable in case 2 where a user moving a lens filter very rapidly only cares about its effect on the final region it is placed on.
3. Process every single action in a First-Come-First-Served manner but do so asynchronously, i.e. the method call should not block until a particular action is responded to. This is suitable in case 3 where every change in the controls affects the agents in the result pane of the DQ browser.

Alternative 2 was realized using Mozart's attributes which are class members that allow destructive assignment. One such member stores the action to be processed, with new actions overwriting the value of the attribute, thereby discarding older actions. Alternatives 1 and 3 were realized using Mozart ports since they are, by definition, asynchronous channels on which messages appear in the order they were sent. The difference between 1 and 3 is that 1 blindly processes all the actions received by the port while 3 checks every action to see whether it is still relevant.

6 Conclusion and Future Work

In this paper, we illustrated the use of Mozart and Oz in developing the Agent Visualization System; the first generic, distributed system specifically dedicated to the visualization of agent-based simulation scenarios. The AVS provides several graphical and textual browsers that utilize a variety of Information Visualization techniques to offer a deeper insight into the simulation scenario being studied. Throughout the AVS, we made use of various features offered by Mozart including dataflow variables, threads, high-orderness, and many others. Mozart allowed us to use approaches from the object-oriented, functional, logic, concurrent and distributed paradigms in a smoothly-integrated way that would not have been possible with another development platform.

There are a number of useful extensions that can be made to the AVS. Currently, the fact that the Mirror Store is centralized makes it a bottleneck. The MS can be broken down by region, agents, or attributes. A central entity should keep track of which agents/regions/attributes are on which site. How the distributed fragments can be managed is a research topic. Another enhancement is to support landscapes that are not grid-like, possibly allowing the visualization of arbitrary-shaped landscapes, as well as nested landscapes where each cell is either a simple cell or a landscape.

Acknowledgements. The authors would like to thank Fredrik Holmgren of the Distributed Systems Lab at the Swedish Institute of Computer Science (SICS) for the numerous fruitful discussions with him during the design phase of the AVS. We would also like to thank Donatien Grolaux of the Université Catholique de Louvain for his patience with our technical questions regarding QTk.

References

1. Jean-Daniel Fekete, Catherine Plaisant: Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization, Conference on Human Factors in Computer Systems (CHI'99), ACM , New York, pp. 512-519, 1999.
2. Jean-Daniel Fekete, Catherine Plaisant: Interactive Information Visualization to the Million, Symposium on Information Visualization (InfoVis'02), Massachusetts, USA, October 2002.
3. Foundation For Intelligent Physical Agents: FIPA Interaction Protocol Library Specification. Document number DC00025F, 2000.

4. Jade Goldstein, Steven F. Roth: Using Aggregation and Dynamic Queries for Exploring Large Data Sets, Computer Human Interaction (CIH'94) Human Factors in Computing Systems, ACM, April 1994.
5. Donatien Grolaux, Peter Van Roy, Jean Vanderdonckt: QTk - A Mixed Declarative/Procedural Approach for Designing Executable User Interfaces, Engineering for Human-Computer Interaction (EHCI 2001), Canada, May 2001.
6. Hala Mostafa, Reem Bahgat: The Agent Visualization System: A Graphical and Textual Representation for Multi-Agent Systems. In Proceedings of the Second International Conference on Informatics and Systems (INFOS2004), Cairo, Egypt, 2004.
7. Mozart, http://www.mozart-oz.org.
8. Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, Steven P. Reiss: Stretching The Rubber Sheet: A Metaphor For Viewing Large Layouts on Small Screens. In Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology, 1993.
9. Chris North, Ben Shneiderman: Snap-together Visualization: Can users construct and operate coordinated views?, International Journal of Human Computer Studies, Elsevier Ltd., 2000.
10. Ben Shneiderman.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization. In Proceedings of the IEEE Symposium on Visual Languages, pp. 336-343, September 1996.
11. Maureen C. Stone, Ken Fishkin, Eric A. Bier: The Movable Filter as a User Interface Tool, Computer Human Interaction (CHI'94) Human Factors in Computing Systems, ACM, April 1994.
12. Fernanda B. Viegas, Judith S. Donath: Chat Circles, Special Interest Group Computer Human Interaction Conference on Human Factors in Computing Systems: the CHI is the limit, Pittsburgh, Pennsylvania, United States, pp. 9-16, 1999.
13. Gerd Wagner, Florin Tulba: Agent-Oriented Modeling and Agent-Based Simulation, The 5th International Workshop on Agent-Oriented Information Systems (AOIS-2003), 2003.
14. M. Wooldridge: Intelligent Agents. In: Gerhard Weiss (Ed.). Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, 1999.