VizScript: A High-Level Language for Rapidly Creating Custom Visualizations for Multi-Agent Systems *

Jing Jin, Rajiv T. Maheswaran, Romeo Sanchez and Pedro Szekely
Information Sciences Institute, University of Southern California
4676 Admiralty Way Suite 1001, Marina del Rey, CA 90292
+1-310-822-1511
{jing,maheswar,rsanchez,pszekely}@isi.edu

ABSTRACT

We address the problem of creating visualizations to debug and understand multi-agent systems. The key challenges are that (1) needs arise dynamically, i.e., it is difficult to know *a priori* what visualizations one wants, (2) extensive expertise on the system, the algorithms and visualization tools are often needed for implementation, and (3) agents can be running in a distributed environment. We have developed VizScript, a collection of tools that expedites the process of creation. VizScript does not require great knowledge of the intricacies of internal data structures or display creation and is functional with data streams from multiple sources. We show that by combining generic instrumentation, a knowledge base, and simple scene definition primitives with a reasoning system, we are able to recreate the visualizations for a complex multi-agent system with an order-of-magnitude less effort than was required in a Java implementation.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Algorithms, Measurement, Performance, Design, Languages

Keywords: Software Visualization, Scripting Languages, Multi-Agent Systems, Rule-Based Systems

INTRODUCTION

Understanding the behavior of complex software is challenging. Understanding the behavior of multi-agent systems is even more challenging given the additional timing and information sharing issues involved. We focus on building visualizations of multi-agent systems to help users understand the behavior and interactions among agents. While there are many tools to build visualizations of software [2, 12], they require significant effort to build. The users must also be developers who understand the software being visualized and specialized languages designed for software visualization.

Our goal is to enable users who understand the application requirements, but are not necessarily developers, to build custom visualizations in minutes rather than days. We want to support a paradigm similar to debugging: users run the software with a few visualizations of the system to observe its behavior. Upon observing interesting or suspicious behavior, users can quickly construct a new visualization to focus on a particular aspect of the behavior to gain further insight. This is similar to setting breakpoints in a debugger, re-running the application and observing the values of specific variables. The key point is that users don't know *a priori* what visualizations are needed.

In order to support such a paradigm, it is crucial to make visualizations easy to build. The alternative is to wade through long textual log files that report key events and variable values. Not only is it a time-consuming and frustrating endeavor, but it is much easier to gain insight into the behavior of a system from animated interactive graphical views than from linear text logs.

The desiderata for the visualization system are:

1. **Easy to build visualizations:** enable users to build new visualizations in minutes rather than days.
2. **Generic:** support a wide variety of visualizations and applications.
3. **Multi-agent:** support visualizations that integrate information from agents running on multiple machines.
4. **Online/Offline:** support visualization of the software while it is running, but also after the system has completed.
5. **Large applications:** support visualization of large applications with thousands of lines of code.
6. **Forward and reverse playback:** support the ability to play the visualizations back in time to enable users to observe the state of the system at any point in the past.

*The work presented here is funded by the DARPA COORDINATORS Program under contract FA8750-05-C-0032. The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
, .
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

The online/offline support is very important for thorough software testing. The online support allows users to observe the behavior of a running system. This is crucial during debugging sessions where users want to see whether recent changes to the software had the intended effect. Offline support is important for regression testing. It enables running the system many times and using the visualization tools to analyze specific runs. Offline support is also important in multi-agent systems where timing considerations make it difficult or impossible to fully reproduce the behavior of a specific run. Offline support is also challenging because we want to enable authoring of new visualizations after the system has run, and data has already been collected. Constructing tools to meet all these requirements is difficult in itself. The desire to make it easy to use is even more difficult, and is the focus of our work.

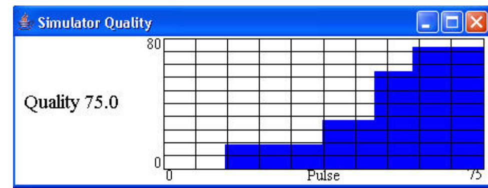
Building a software visualization involves two main tasks: *instrumentation* (the software to be visualized must be instrumented to trigger the visualization system) and *scene definition* (the specification of what to show when the triggers fire). These two tasks are often in conflict. If one makes the instrumentation easy by using the software data structures and procedure/method definitions as the triggers, the scene definitions become more complex in order to bridge the gap between the software abstractions and the visualization that the user wants to see. Alternatively, one can make scene definitions simpler by raising the level of abstraction on the triggers, but this requires adding more instrumentation code to detect higher level events.

Our approach, embodied in a tool called VizScript, is to simplify both instrumentation and scene definition by introducing a reasoning system between the two. The reasoning system enables the instrumentation component to produce very simple information and enables visualization authors to simply define higher-level abstractions matched to the requirements of their visualization. The reasoning system uses a knowledge base to record the instrumentation output and uses a production-system architecture to enable authors to define patterns that identify high-level events.

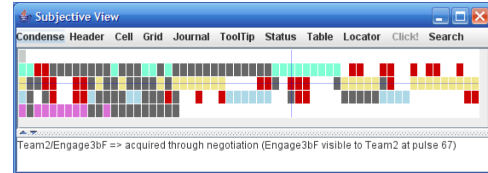
In the next section, we discuss some examples of visualizations. We then present the high-level architecture of VizScript, and a small detailed example to show how a visualization is specified. We describe the details of our reasoning system and show how it supports the example. Later, we present an evaluation of the system by comparing the effort to build visualizations with and without the system, and end with related work, conclusions and directions for future work.

EXAMPLES

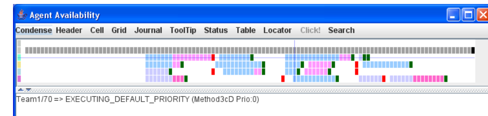
We consider a large multi-agent application where VizScript has been used extensively. In this application, agents help people manage activities they need to perform in order to meet an objective. Each agent has knowledge of the activities that its owner can perform, as well as enabling activities that others are tasked to perform. The agents continuously maintain a schedule, adapting it as necessary to cope with delays and failures. To do so, agents exchange information about activities that they can control: probabilities of success, importance of activities, etc.



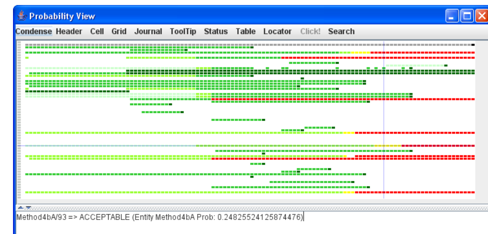
(a) Quality View



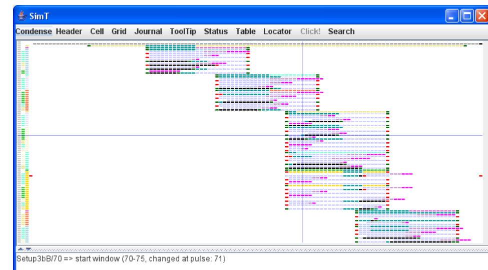
(b) Subjective View



(c) Agent Activities View



(d) Probability View

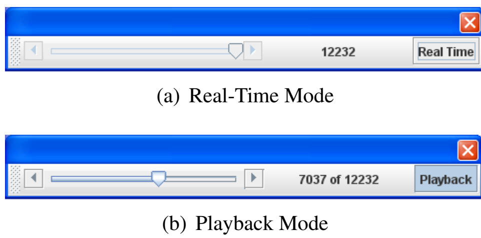


(e) Execution View

Figure 1: Example Visualizations

Evaluating the multi-agent system involves analyzing decisions over time, and thus, the needed visualizations are sophisticated animations. Figure 1 shows static shots of four such visualization examples. A full animation of the visualizations can be seen at: *http://www.isi.edu/~szekely/csc/aamas06/csc-demo-v01.html*.

The Quality View visualization shows the evolution of task achievement that the multi-agent team has accomplished thus far in the scenario. The horizontal axis represents time and the vertical axis represents quality. The visualization monitors reports from the agent responsible for tracking the quality of the overall team objective.



(a) Real-Time Mode

(b) Playback Mode

Figure 2: Time Sliders for Visualizations

The Subjective View visualization shows the activities that agents know about. The rows represent agents and the columns represent activities. Each agent is assigned a unique color. Cell (i, j) is painted with the color assigned to $agent_i$ if it owns $activity_j$. It is painted grey if $agent_i$ learns about (unowned) $activity_j$ during initialization. A cell is painted red if $agent_i$ came to know about $activity_j$ during schedule negotiation between the agents. The visualization integrates reports from every agent about the activities that they know about.

The Agent Activities View visualization shows the activities that each agent is performing at any given time. Rows represent agents and columns represent simulated time. Cell (i, j) is colored if $agent_i$ is executing an activity during $time_j$. The color represents the priority of the activity, or whether the activity was performed successfully or it failed. The tooltip shows the name of the activity. The visualization integrates reports from every agent when they start and end activities and when they change the priority.

The Probability View visualization shows the evolution of the probability of successfully executing activities. Rows represent activities and columns represent simulated time. Cell (i, j) is colored to show the probability of successfully executing $activity_i$ at simulated time $time_j$. The tooltip shows the exact probability together with information about the importance of the activity.

The Execution View visualization shows an integrated view of all activities, independent of the agent that owns it. Rows represent activities and columns represent simulated time. The cells show information about the release and deadline of activities, their possible durations, the planned start time, the execution status, the agent who owns it, etc. The display is densely packed with information, and shows all the relevant information to let developers and testers understand what is happening in the distributed execution of activities.

All views are coordinated. As the simulation advances, all the displays update simultaneously. More importantly, the visualizations can be played back in time. Using the time sliders, shown in Figure 2, the user can move time backwards (and forwards) to view the state at any given point in time. For example, the user can position the simulation at the time when a given activity started, to see the probability of that activity and all other activities, what the other agents were doing, and which activities each agent is aware of at that time.

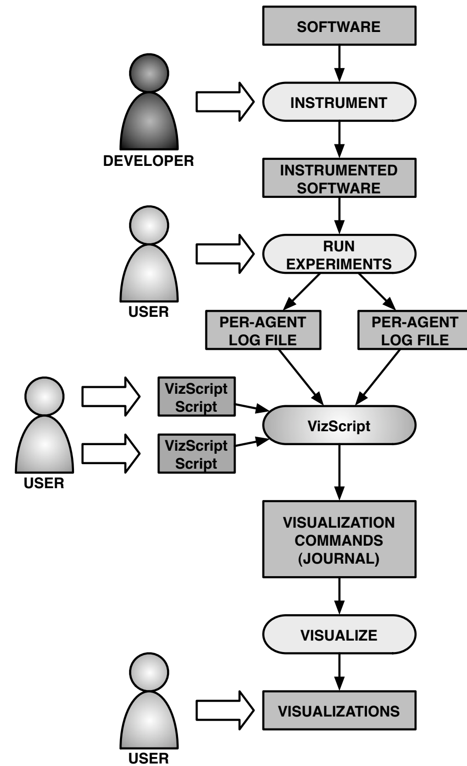


Figure 3: VizScript architecture.

ARCHITECTURE OVERVIEW

Figure 3 shows how VizScript is integrated with an application, and the different roles involved in defining new visualizations. The first task, performed by the software developers, is to instrument the application software to produce the data that drives the visualizations. This involves augmenting the source code with instrumentation commands, similar to print statements typically used for debugging. When users run experiments with the instrumented software, it produces a stream of records containing information about changes in the state of the system. In multi-agent systems, each agent produces a separate stream. These streams can be fed directly to the visualization system for online visualization, or can be saved to disk for use during offline visualization.

To produce visualizations, users write scripts that define what they want to see. These scripts are saved and reused. VizScript takes as input a set of scripts and the data streams produced by the application. The output is a sequence of graphics commands that are fed to a graphics library to drive the display. When using VizScript in online mode, users specify the scripts before running the application. In offline mode, users invoke VizScript with a set of scripts and files containing the data streams saved from a prior run. This enables users to run VizScript multiple times to produce alternative visualizations of the data produced in a prior run of the application.

Figure 4 shows the architecture of the VizScript interpreter. The first step is to consolidate the data streams produced by multiple agents. The consolidated data stream is used to

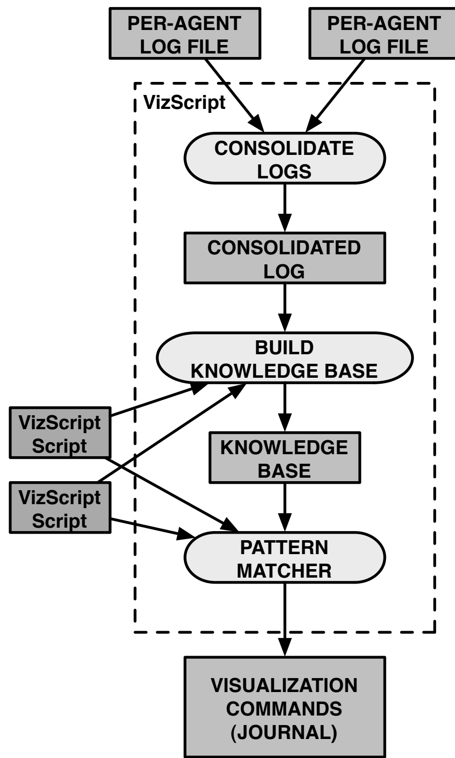


Figure 4: Components of the script interpreter.

build a knowledge-base that contains the information needed for visualization. The component that builds the knowledge-base takes the scripts as input and produces a knowledge-base containing only the information referenced in the given scripts. This is important to improve performance because the data streams often contain thousands of records, many of which are not relevant to the visualizations that the user wants to see.

Scripts are defined using rules. The condition part of a rule is defined using a pattern that is matched against the contents of the knowledge-base. The actions are commands to the graphics library. VizScript builds the knowledge-base one record at a time. After each record is added, VizScript fires all the matching rules in all scripts.

SOFTWARE INSTRUMENTATION

In VizScript, we take an object-oriented view of the software. We assume that the software to be visualized is defined in terms of objects, and the purpose of the visualizations is to show the evolution of the state of the objects in the system. We further assume that objects have unique identifiers, and their state is defined in terms of properties whose values can be primitive types (String, Integer, Double and Boolean) or references to other objects.

In order to show the evolution of the state of objects, the software is instrumented to announce changes to the state of objects by producing a stream of data records with the following fields:

- **Agent:** the agent who made the change to the object.

```
1  Team3,35582639330697,Team3,agentLabel,Team3
2  Team3,35582670779666,Problem1,releaseTime,15
3  Team3,35582670784415,Problem1,deadlineTime,112
4  Team3,35582670795310,CTask1,supertaskRelation,Window1
5  Team3,35582670800618,CTask1,releaseTime,30
6  Team3,35582670805647,CTask1,deadlineTime,56
7  Team3,35582670816821,Window1,supertaskRelation,Problem1
8  Team3,35582670821570,Window1,releaseTime,30
9  Team3,35582670825761,Window1,deadlineTime,56
10 Team3,35582670836656,CTask2b,supertaskRelation,Window2
```

Figure 5: Example data records

- **Time-Stamp:** the time of the announcement of the change.
- **Object:** the identifier of the object changed.
- **Property:** the name of the object property changed.
- **Value:** the new value of the object property.

In multi-agent software, we assume that multiple agents can reason about the same collection of objects. Each agent has its own in-memory representation of the same object, but they all use the same identifier. When one agent changes the state of an object, it must inform the other agents about the change. This means that there are times when agents have inconsistent versions of the same object. The Agent and Time-Stamp fields enable the visualizations to reason about the authors of changes and the latency of change propagation in the agents.

Our representation is similar to RDF triples [3], with the addition of the agent and time-stamp fields. These could have been represented using reification, but the additional triples required to do so introduce inefficiencies that we don't want to incur. The main benefits of our representation, are the same as the benefits of using RDF, namely that the representation is generic and easily distributable. The pitfalls are also similar to those of RDF, namely reasoning and efficiency. Later, we show how we reason with this information and performance results.

Figure 5 shows the first 10 records in a data stream collected by one agent from a multi-agent application containing thousands of such records. As discussed in the previous section, the data streams from multiple agents must be consolidated into a single stream before the visualization system can process it. The time-stamp clock has nano-second resolution, so every record in a stream has a unique, increasing time-stamp. We assume that the clocks for multiple agents may be skewed, but also assume that they run at the same speed. Consequently the issue in consolidation is to determine the skew. We do this by using an auxiliary agent that sends a Pulse message to every agent. All agents are instrumented to log receipt of this message. We assume that all agents receive the Pulse message at the same time and align their time stamps accordingly. Even though more accurate schemes are possible, we found this simple scheme effective in that it can correct arbitrarily large machine clock skews and the margin or error is on the order of a few milliseconds.

VizScript LANGUAGE

VizScript is an interpreted language with an integrated knowledge base and pattern matcher. The interpreted language is similar to other scripting languages such as JavaScript [6]. The pattern matcher is similar to the pattern matcher in OPS5

```
starfield {
  window["title"] = "Subjective View";

  colors["acquired initially"] = "#666666";
  colors["acquired through negotiation"] = "#CC0000";

  // Add agent rows
  when (?agent "agentLabel" ?label){
    generateStatusColor(?agent);
    addRow(?agent, ?agent, ?agent);
  }

  // Add entity columns and paint cells
  when ((?entity "entityType" "entityTypeMethod")
        || (?entity "entityType" "entityTypeTask"))
       && (?entity "entityOwner" ?owner) {

    addColumn(?entity);

    if (?owner."smInitializationCompleted") {
      color = "acquired through negotiation";
    } else {
      if (?reportingAgent == ?owner)
        color = ?owner;
      else
        color = "acquired initially";
    }
    toolTip = ?entity + " visible to " + ?reportingAgent
              + " at pulse " + "clock"."starfieldPulse";
    addCell(?reportingAgent, ?entity, color, toolTip);
  }
}
```

Figure 6: VizScript script for the Subjective View

[7]. We describe the language by first introducing an example and then present the different language constructs in more detail.

VizScript Example

Figure 6 shows the complete VizScript script for the Subjective View visualization. Line 2 specifies the title of the window, and lines 4 and 5 define the colors of cells: activities that an agent acquires during initialization are grey, and activities that an agent acquires later are red.

The rest of the script defines the rules to process the records in the data stream. The first rule in lines 8 to 10 specifies when to add rows to the visualization. It does so when it sees a record with property `"agentLabel"`, given that this property is used to announce the name of an agent. Line 9 generates a unique color for the matching agent, and the following statement adds a row for that agent. The first parameter `?agent` is the identifier of the row (the name of the agent), the second parameter is the label for the row (we use the name of the agent), and the third parameter is the symbolic name color for that row (in this example, we also use the name of the agent as the symbolic color name).

The second rule in lines 13 to 31 specifies when to add columns and how to color the cells. The pattern defined in lines 14 to 16 specifies that we are looking for entities corresponding to activities, i.e., tasks or methods. In addition, we bind the variable `?owner` to the agent who owns the activity. This pattern illustrates the use of Boolean expressions in the pattern language.

The body of the `when` statement specifies what to do when we detect a new activity. In line 18, we add a column corresponding to this activity. The identifier of the column is `?entity`, the identifier of the activity just detected. In lines 20 to 27, we define the color of the cell to be painted. The goal of this display is to show how much information each agent gathers from other agents. In lines 20 to 22, we specify that if the agent came to know about the activity after initializing its local model, then we use the symbolic color `"acquired through negotiation"`. Note that in line 5, we had mapped that symbolic color to red. The symbolic colors appear in tool-tips so it is good practice to give them appropriate names. In lines 23 and 24, we address the case of an activity where the reporting agent is the same as the owner. In that case, we color the activity with the color of the agent. In line 26, we have the case of an activity where the reporting agent and the owner are not the same. This is where an agent (the reporting agent) gained access to an activity owned by another agent during the initialization process. We use an appropriately named color (`"acquired initially"`) to indicate that.

In lines 28 and 29, we compose a detailed tool-tip to explain what happened. We access the `"clock"` object in the knowledge-base to show in the tool-tip the simulation time when the event happened. In line 30, we add the cell to the display in the row of the reporting agent.

Figure 1(b) shows the finished screen. Note the tool-tip at the bottom of the screen. The script could hardly be shorter. All the statements in it define a display or behavior property. None are redundant. The author is not required to write any statements to do book-keeping of any kind, translate data representation, or any other such programming tasks.

Knowledge Base

The knowledge base in VizScript is a frame system. The frames correspond to objects, and the slots in the frames correspond to properties. The knowledge base is built from the consolidated data stream. When a tuple of the form ($agent$, $object$, $timestamp$, $property$, $value$) is processed, a frame for the given object is created if one doesn't exist already. The slot named by the property is set to the given value. The agent and time-stamp are stored in facets of the slot. In the current implementation, we assume that all slots are single-valued. Consequently, when processing a tuple naming an object-property pair for which a slot is already defined, the existing value of the slot is replaced by the new value.

Variables

VizScript is a weakly-typed language. Variables can hold values of any type, and they do not need to be declared. There are three kinds of variables:

Simple variables can store one value. They can be read and written. For example:

```
a = 5;
b = a + 1;
```

Array variables map a list of values to another value. For example:

```
x[a, 1] = "yes";
x[2] = "no";
y[x[2]] = "maybe";
```

The `window` and `colors` arrays are special. The `window` array specifies the visual appearance of the window (title, location, size, etc.). The `colors` array specifies symbolic names for colors.

Pattern variables store bindings of patterns matched against the knowledge base. They can be read, but can only be assigned by the pattern matcher. Pattern variables start with "`?`" (e.g., `?entity`).

Expressions

VizScript supports variable reference expressions, a variety of arithmetic and Boolean expressions, function calls, and knowledge-base path-expressions to fetch values from the knowledge base. These path expressions are of the form $exp1.exp2$ where $exp1$ must evaluate to the name of an object in the knowledge base, and $exp2$ must evaluate to the name of a property. The value of the path expression is the value of the corresponding slot in the knowledge base, or the special value `NoValue` if the object is not in the knowledge base or the slot is undefined. For example, the following are path expressions:

```
"task1"."probability"
?x."probability"
```

When evaluated, the first expression yields the value of the `"probability"` of the object named `"task1"`. The second evaluates to the value of the `"probability"` for the object bound to variable `?x`.

Statements

VizScript defines four types of statement: *assignment, function call, if* and *let*. The first three have the obvious meaning. The *let* statement is used to explicitly invoke the pattern matcher. There are no iteration statements such as the traditional *for* and *while* loops. Iteration is implicit in that statements are executed multiple times when patterns are satisfied by multiple variable bindings. The pattern matcher and the *let* statement are explained in more detail below.

Patterns

A primitive pattern is of the form $(exp1, exp2, exp3)$, where each of the expressions can be either a pattern variable or an arbitrary expression that evaluates to a value. For example, the following are primitive patterns:

```
("task1" "probability" ?p)
(?x "probability" ?p)
(?x "probability" 0)
(?x myVar1 myVar2 / 2)
```

A binding is an assignment of values to the variables in a pattern that make the pattern true in the knowledge base. A pattern can have multiple bindings when multiple assignments of variables make the pattern true. VizScript computes all bindings when it evaluates patterns.

In the example above, the first pattern will have zero or one bindings depending on whether the `"probability"` of `"task1"` is defined in the knowledge base. When it is, `?p` is bound to the value of the `"probability"`. The second pattern will typically have multiple bindings consisting of pairs of objects and their probability. The third pattern is similar to the second except that it binds to all objects whose `"probability"` is zero. The fourth pattern illustrates the use of expressions. In general, patterns are Boolean combinations of primitive patterns. VizScript supports conjunction, disjunction and negation.

Structure of a VizScript Script

As illustrated in Figure 6, a VizScript script consists of global statements followed by a sequence of rules defined using `when` statements. The global statements typically define the appearance of the window using the `window` array, defines symbolic names for the colors using the `colors` array, and often invokes functions to set up a framework for a window that is independent of the data it shows (e.g., add rows or columns).

The `when` statements have the following structure:

```
when Pattern where BooleanExpression {
  Statement+
}
```

VizScript first executes the global statements and then consumes the consolidated data stream one record at a time. For each record it:

1. Adds the record to the knowledge base, if it refers to a property used in a script.
2. Selects `when` statements to evaluate.
3. Evaluates selected `when` statements.

The second step distinguishes VizScript from traditional rule-based systems. If we simply add a new record to the knowledge base and then compute the set of all possible bindings for a pattern, this would often result in bindings that have nothing to do with the record that was just added. If a pattern had bindings before the record was added, those same bindings would still be true if the pattern made no reference to the property of the added record. Intuitively, we want the `when` to fire when the added record affects the pattern.

In the selection process VizScript selects for evaluation only those `when` statements whose patterns include primitive patterns referring to the property that was just added to the knowledge base. Furthermore, it binds any variables in those primitive patterns to the object and value of the record just added. If such bindings are inconsistent, then the `when` statement is not selected.

For example, suppose that the following record is added to the knowledge base:

```
("agent1" 0 "task1" "probability" 0.3)
```

Consider the following `when` statements:

```
when (?x "probability" ?p) { ... }
when (?x "importance" ?i) { ... }
when ("task2" "probability" ?p { ... }
when (?x "probability" ?p) { ... }
when (?x "probability" 0) { ... }
```

The first `when` statement would be selected because it contains the property `"probability"`. The variable `?x` would be bound to `"task1"` and `?p` would be bound to `0.3`. The second statement would not be selected because it does not mention `"probability"`. The third statement would not be selected because even though it mentions `"probability"`, it requires the object to be `"task2"`, but the record is about `"task1"`. The fourth statement would not be selected either because it is looking for a `"probability"` equal to zero.

After selecting `when` statements and defining bindings to the object and value of the added record, VizScript proceeds to compute all the possible bindings for the remaining variables. This only occurs for composite patterns containing multiple primitive patterns. The bindings must satisfy the Boolean `Pattern` as well as the `BooleanExpression` defined in the `where` clause of the `when` statement. The `where` clause allows definition of additional constraints that cannot be expressed through unification of pattern variables (e.g., inequality of two variables). VizScript then executes the body of the `when` once for each possible set of bindings. This powerful feature enables updating multiple objects on the screen when some key property changes.

The `let` statement is similar to the `when` statement. The only difference is that it is not subject to the selection process described above. The following example illustrates the use of the `let` statement in the Agent View visualization:

```
when ("clock" "starfieldPulse" ?pulse) {
  addCell("pulse row", ?pulse, "current pulse");
  let (?method "entityType" "entityTypeMethod") &&
      (?method "executionStatus" "qualityChanging") &&
      (?method "executingPriority" ?p) {
    addCell(?method."entityOwner", ?pulse, prio[?p]);
  }
}
```

The `when` statement is triggered every time the simulation clock advances. The `let` statement then fetches *all* primitive activities (`"entityTypeMethod"`) whose quality is changing, and binds `?p` to the execution priority. The action records the priority that each executing activity had at every simulation time. The pattern can have multiple bindings, so it updates cells in multiple rows. This illustrates how the pattern enables statements to be invoked multiple times without the use of explicit iteration statements.

The body of `when` statements typically contain commands to drive the underlying graphics package. In VizScript we use our own graphics library called Starfields [13]. It supports the construction of line graphs and a variety of displays consisting of tables of dynamically changing color cells. Our Starfield package supports the ability to play the visualizations forwards and backwards, thus satisfying desiderata number 6.

Figure 7 shows part of the output of our example VizScript script shown in Figure 6. It is very generic and easily extensible to accommodate other graphics libraries. Each line contains a command to the graphics package. The first element in each line is an identifier for the window, the second is the time-stamp from the input data stream, and the other entries correspond to the various functions of the graphics package.

```
1  sf1,0,SfNewStarfield
2  sf1,0,SfSetWindowTitle,Subjective View
3  sf1,0,SfAddStatusColor,acquired initially,#666666
4  sf1,0,SfAddStatusColor,acquired through negotiation, \
5    #CC0000
6  sf1,0,SfAddStatusColor,Team4,#7fffd4
7  sf1,0,SfAddRow,Team4,Team4,Team4,
8  sf1,5298,SfAddColumn,Window2,Window2,
9  sf1,5299,SfAddCell,Team4,Window2,acquired initially, \
10   Window2 visible to Team4 at pulse 2
11 ...
```

Figure 7: Output of the VizScript script in Figure 6.

Even though VizScript currently works with our own Starfields graphics package, the VizScript script language is generic and could easily be used to drive a different graphics package. Only the functions called in the body of `when` statements would change, but the scripting language would remain the same.

EVALUATION

In this section, we first describe the context that led to VizScript by discussing the original development process for visualizations. We then evaluate the main claim of our paper by showing that VizScript significantly cuts down the effort for creating visualizations, when compared with the Java code required to produce the same animations. Finally, we show that the VizScript methodology is sound. We show that (a) the instrumentation costs are comparable to the instrumentation costs of the old system; (b) while the consolidated data streams are large, they are manageable; and (c) the times to process the data streams using Artificial Intelligence methods (knowledge-base and pattern matcher) are short enough to produce a usable system.

The original development process for visualization tools

To fully understand the utility of using VizScript, we describe the process by which some of the current visualizations came into being. The original system would write various state updates to a set of textual log files (one for the system and one for each agent). If the quality of the performance was unsatisfactory, one would have to search through the textual log files to discover the activity and reasoning failures that caused an undesired result. This was clearly inefficient so we designed an Execution View visualization to monitor all activities, deadlines, the schedule and various other parameters such as activity duration distributions. Implementing the Execution View visualization involved writing code in various components that announced when certain events of interest occurred. Listeners associated with the visualization would identify the relevance of the announcement, extract the appropriate information from the data structures in the proper components and update its display. When analyzing the system with the Execution View, we understood the evolution with respect to the scheduled and potential activities, but in

Visualization	Lines of Code		Savings
	Original	VizScript	Factor
Quality View	122	5	24.40
Subjective View	141	21	6.71
Agent Activities View	190	47	4.04
Probability View	258	23	11.22
Execution View	1214	93	13.05

Table 1: Effective Lines of Code for Visualizations

order to understand the choices made by the agents, we realized that we needed to understand how each agent was using its time. Thus, we designed the Agent Activities View which necessitated further instrumentation and implementation. The Agent Activities View revealed that there were inefficiencies in how agent's were using their time. To make better choices, agents needed to know the probabilities of success of the activities and tasks that made up the objective. To verify that agents were following the algorithms properly, we designed the Probability View visualization, which again necessitated additional instrumentation and implementation.

This process highlights several key challenges: (1) the set of visualizations needed to debug and understand the multi-agent system developed dynamically; we began with one and evolved more than ten as needs arose; (2) implementing a new visualization required significant time (sometimes over a week) from a programmer who had familiarity with all components of the system, the nature of the needs of the visualization and the visualization generation procedures; (3) because the tools needed access to the data structures of the agents, they were not capable of operating when the agents were distributed to different machines.

VizScript addresses all three issues: (1) The simplicity of VizScript allows a new visualization to be created very quickly (in a few hours for displays such as Subjective View and Agent Activity) as needs arise; (2) once the objects and properties that compose the tuples are decided, a designer needs to understand only a few constructs and display commands to generate a visualization; (3) because each agent generates and stores tuples, these data streams can be aggregated from various machines and processed together, allowing us to debug and understand a distributed multi-agent system.

Quantitative differences in coding

To quantitatively express the benefits of VizScript, we considered the lines of code that went in to creating various visualizations discussed in this paper. In Table 1, we show the number of lines it took to generate the Subjective View, Agent Activity View, Probability View and Execution View visualizations. The lines are effective lines of code (eLOC) [5], defined as the total lines of code (no blank lines or comments) excluding stand-alone braces and parenthesis. As we can see, VizScript enables approximately an order-of-magnitude reduction in the number of lines necessary to create these visualizations. Also, note that the total number of lines of code in VizScript are small, typically under a page.

Feasibility of Usage

While we have shown that VizScript offers an easy and fast way to generate visualizations for multi-agent systems, a natural concern is whether the logging costs (from recording all the event tuples) and processing costs (from evaluating the logged tuples against the patterns in the visualization scripts) will overwhelm computational resources making the system impractical.

We first address the cost of instrumentation, i.e., the computational requirements to gather the data to create the visualizations. In the original system, this involved creating announcements, identifying appropriate events and extracting the relevant data from proper components. In VizScript, instrumentation is the creation and storage of a stream of data records. We tested the costs of instrumentation on three problems: (1) a *unit test* problem with 3 agents, 18 potential activities, 14 tasks and a 55-epoch time-horizon, (2) a *small* problem with 4 agents, 43 potential activities, 16 tasks and a 115-epoch time-horizon and (3) a *large* problem with 5 agents, 245 potential activities, 93 tasks and a 184-epoch time-horizon. The execution time for the three problems under various instrumentation structures is shown in Table 2. Instrumentation does not have a significant effect on run-time: system run-time is proportional to the size of the problem, but the type of instrumentation has minimal effect on run-time. In fact, for the large problem the randomness inherent in each trial had a larger effect on run-time than type of instrumentation.

A potential pitfall of VizScript is the size of logging and processing costs to create the display. Table 3 shows the storage size for the log files when unzipped and zipped. The second column shows the number of lines in the consolidated log file. The third and fourth columns show the size of the unzipped and zipped consolidated log files created using the VizScript instrumentation for an entire run. While the raw files can grow large, the zipped versions are very manageable.

Table 4 shows that the time to read and unzip the zipped files is comparable to the time to read and process the raw uncompressed files. Consequently we configure the system to compress the log files allowing us to easily process hundreds of experiments. Also note that the time to read the log files is small compared to the times to run the simulations.

Table 5 shows the time taken to process the log files for various configurations of visualizations. The first four rows are the processing times when each visualization is the only one displayed. The fifth and seventh row show the processing times for two different collections of visualizations. The times to display are always on the order (and less than) the run time of the simulation. The sixth and eighth row are the sums of the times when processing each script by itself. When processing a collection, the processing time is sub-additive with respect to processing individually. This points to the advantages of sharing the knowledge base in multiple scripts.

The visualizations of small problems can all be produced in under 5 seconds. This performance level is perfectly ade-

Problem	Problem Run Time by Instrumentation Type (time in seconds)		
	None	Original	VizScript
Unit Test	5.334	5.438	5.334
Small	12.783	12.920	13.069
Large	128.690	127.500	116.215

Table 2: Execution Time For Various Instrumentations

Problem	Size of Log Files		
	Lines	Memory (MB)	
		Unzipped	Zipped
Unit Test	13,831	1.29	0.11
Small	76,641	7.09	0.60
Large	601,433	56.77	5.45

Table 3: Logging Costs for VizScript

quate for analyzing and debugging our multi-agent system. The visualization of the large problem takes 74 seconds to produce. This is a significant time to wait for interactive use. In this example, VizScript processes about 8000 records per second, and fires about 80 rules per second. We plan on identifying the bottlenecks and optimizing performance, in the next phase of work.

RELATED WORK

Demetrescu, Finocchi and Stasko [4] proposed two ways to bind visualizations to the actual software objects. The *event-driven* method requires users to annotate the source text of the software. These annotations represent events, which call the system's visualization routines. The main disadvantage of this approach is that users need to understand the system source code and visualization routines. The *data-driven* method allows users to define relations between software states and the visualization objects. Our approach is similar to this method, but we provide a reasoning system between the software product and the visualization scheme, which simplifies instrumentation also making the approach more generic.

Khaled, Noble and Biddle [8] presented a system called InspectJ, which is a program visualization system that uses AspectJ [9] to automatically collect program monitoring information for visualization. Users can use AspectJ to specify points in the program execution that will yield interesting information for visualization. However, one the main disadvantages of this approach is that it needs the program execution to feed the visualization system, precluding its use in offline analysis.

Liao and Cohen's work [10] focused on making instrumentation easier. They proposed a high level program monitoring and measuring system (PMMS) that accepts the original program and a set of questions posed in a formal specification language. Then, it installs instrumentation code directly into the program that determines the data that needs to be collected for visualization. However, PMMS's performance is limited by how well the system can understand the input program, while ours is only limited by the expressivity of our scripting language.

Log Type	Unit Test	Small	Large
Unzipped	0.42	1.81	13.61
Zipped	0.46	1.94	14.40

Table 4: Time to Read Log Files (secs)

#	Visualization	Unit Test	Small	Large
1	Subjective (S)	0.79	1.12	6.41
2	Agent Activity (A)	1.01	1.51	12.71
3	Execution (E)	1.06	1.89	6.79
4	Probability (P)	1.28	3.93	58.19
5	{(S),(A),(E)}	1.28	2.36	18.97
6	(S)+(A)+(E)	2.86	4.52	25.91
7	{(S),(A),(E),(P)}	1.77	5.47	74.05
8	(S)+(A)+(E)+(P)	4.14	8.45	84.10

Table 5: Time to Process Log Files (secs)

Tominski and Schumann [14] argued that today's visualization techniques often do not distinguish between the different properties of the data, thus visualizing all aspects of it. This leads to overcrowded and cluttered representations. They proposed a very similar approach to VizScript in which users can specify those aspects of the data that will be monitored, and use events once a particular pattern is detected to draw the visualizations. However, they use XML to define event templates, mapping them later to SQL queries. Our approach is more flexible, since VizScript scripting language and knowledge base can be seen as a deductive system making inferences on the data set with respect to time, producing composite and aggregated data visualizations.

Poutakidis, Padgham and Winikoff [11] argued that although existing debuggers provide essential infrastructure, they cannot effectively filter the information presented to the programmer. However, our approach turns out to be a good solution to this problem, since users can easily focus on any aspect of the system at any particular point in time.

Finally, Ahlberg and Wistrand [1] introduced *IVEE*, an interactive visualization and query system based on the concept of dynamic queries. The idea behind IVEE is to automatically import database relations, and use such relations to generate visualizations. In that sense, it is similar to VizScript given that its knowledge base could be seen as a dynamic database. However, VizScript has the capability of reasoning with time series events, in other words, its knowledge base representation and scripting language provide the functionality to analyze distributed information across time. Another key difference is the fact that the set of visualizations in VizScript is not predefined, since they are user customized through the language.

CONCLUSIONS AND FUTURE WORK

We address the problem of generating dynamic visualizations for large and complex distributed systems. We developed VizScript, an efficient and flexible collection of tools, which expedites the process of building visualizations for complex multi-agent systems. By combining a generic in-

strumentation, a knowledge base, and language primitives with a reasoning system, VizScript is able to reproduce visualization tools in a fraction of the time and human effort necessary when procedural programming languages are used.

We made significant progress towards the first desideratum (easy to build visualizations). VizScript lifts the requirement of extensive expertise on the system and algorithms. It enables building visualizations in hours rather than days, but we have not yet reduced the time to minutes. We believe an interactive environment where users can easily find the names of properties and predefined objects will help achieve this goal.

For the second desideratum (generic), we showed how the system can generate five visualizations of two types (line charts and our version of Starfields). We expect to incorporate new graphics libraries into the system, and thereby demonstrate additional generality of the approach.

We completely satisfy desiderata 3 through 6. VizScript's architecture allows it to work in single as well as distributed computing environments, making it very suitable for complex and large multi-agent systems and parallel computing. Furthermore, our approach can work online as well as in offline mode after the system has completed execution. This is a very powerful mechanism for software testing and debugging, since it allows the developer to run multiple instances of the software, and then select those that will be visualized. In addition, the framework allows the visualizations to play back in time to show the evolution of the system at any time point.

We also presented a performance evaluation of the framework, both in terms of the quantitative differences in coding a VizScript module, and the processing costs incurred in creating the displayed visualizations. We showed that the effective lines of code (eLOC) needed to create a visualization using VizScript represents approximately an order-of-magnitude reduction in effort. Furthermore, the processing costs both in terms of data space and time are manageable.

A shortcoming of VizScript is the lack of multi-value expressions. In our current implementation, objects inside expressions match to a single value given a property. There may be applications in which a range of values could be needed to represent a single property. We plan to address this issue by expanding our reasoning system and data streams. Another possibility for extension is the introduction of state history. Currently, our approach keeps the latest snapshot of the system to drive the visualizations. There may be cases in which a partial history of the evolution of the system may be needed. This would allow VizScript to work not only as a visualization creation framework, but also as an analytical and data mining tool.

Despite these current limitations, VizScript remains a promising and effective approach to creating dynamic visualizations for complex, large and distributed systems. VizScript represents a significant step towards satisfying the desiderata listed in the introduction.

REFERENCES

1. C. Ahlberg and E. Wistrand. Ivee: an information visualization and exploration environment. *infovis*, 00:66, 1995.

2. Sarita Bassil and Rudolf Keller. Software visualization tools: Survey and analysis. In *9th. International Workshop on Program Comprehension (IWPC'01)*, pages 7–17, 2001.

3. World Wide Web Consortium. Resource description language (rdf). http://www.w3.org/RDF.

4. Camil Demetrescu, Irene Finocchi, and John T. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? In *Revised Lectures on Software Visualization, International Seminar*, pages 16–30, London, UK, 2002. Springer-Verlag.

5. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 1998.

6. David Flanagan. *JavaScript: The Definitve Guide*. O'Reilly Media, 5 edition edition, August 2006.

7. Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

8. Rilla Khaled, James Noble, and Robert Biddle. Inspect j: Program monitoring for visualisation using aspectj.

9. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

10. Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.*, 18(11):969–978, 1992.

11. David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 960–967, New York, NY, USA, 2002. ACM Press.

12. John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

13. Pedro Szekely, Craig Milo Rogers, and Martin Frank. Interfaces for understanding multi-agent behavior. In *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces*, pages 161–166, New York, NY, USA, 2001. ACM Press.

14. Christian Tominski and Heidrun Schumann. An event-based approach to visualization. In *IV '04: Proceedings of the Information Visualisation, Eighth International Conference on (IV'04)*, pages 101–107, Washington, DC, USA, 2004. IEEE Computer Society.