

VizScript: On the Creation of Efficient Visualizations for Understanding Complex Multi-Agent Systems ^{*†‡}

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

ABSTRACT

One of the most difficult tasks in software development is understanding the behavior of the final product. Making sure that a system behaves as users expect is a challenging endeavor. Understanding the behavior of a multi-agent system is even more challenging given the additional complexities of multi-agent problems. In this paper, we address the problem of users creating visualizations to debug and understand complex multi-agent systems. We introduce *VizScript*, a generic framework that expedites the process of creating such visualizations. *VizScript* combines a generic application instrumentation, a knowledge-base, and simple scene definitions primitives with a reasoning system, to produce an easy to use visualization platform. Using *VizScript*, we were 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 Keywords

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

General Terms

Algorithms, Measurement, Performance, Design, Languages

Author Keywords

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

*We thank Kevin Smyth, Chris van Buskirk, and Gergely Gati for their work on the *LiveTree* graphics library, and their many helpful comments.

†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. Approved for Public Release, Distribution Unlimited.

‡Permission to make digital or hard copies of part or all 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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. IUI'08, January 13-16, 2008, Maspalomas, Gran Canaria, Spain. Copyright 2008 ACM 978-1-59593-987-6/08/0001 \$5.00

INTRODUCTION

Understanding the behavior of complex software is challenging. Understanding the behavior of multi-agent systems is even more challenging given some common characteristics of multi-agent problems: additional timing constraints, information sharing issues, dynamic and uncertain domains, distributed state information, large data space, etc. One alternative to debug and understand such systems is to wade through long textual log files that report key events and variable values. This is not only time consuming and frustrating, but it is also more error-prone. It is much easier to gain insight into the behavior of a system from animated interactive graphical visualizations than from linear text logs.

In this paper, we focus on building visualizations of multi-agent systems to help users understand the behavior and interactions among agents. The key challenges for creating such visualizations are (1) needs arise dynamically, i.e., it is difficult to know *a priori* the visualizations that one needs or 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. While there are many tools to build visualizations of software [2, 16, 17], 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 and graphics.

Our approach, embodied in a framework called *VizScript*, expedites the process of creating effective visualizations for understanding and debugging complex software systems. *VizScript* visualization platform has been successfully applied to the *Criticality-Sensitive Coordination* project [12, 18], a very complex and large multi-agent system that has successfully outperformed extensions of prominent decision theoretic and scheduling schemes. The *VizScript* visualizations enabled *CSC* researchers to identify important shortfalls in different versions of the system, and provided insights to develop more powerful heuristics that greatly improved the final quality of the application.¹ *VizScript* 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

¹The *CSC* multi-agent system received the DARPA COORDINATORS Phase I and II Champions distinction for unmatched performance.

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. 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 in a very generic way. The general idea is to visualize system components and their evolution during execution. Therefore, the initial 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 it 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.

The multi-agent feature of our previous list is a capability, not a restriction. Ideally, a visualization system should have the property of integrating information coming from different sources, and what is more important, it should be able to make sense of it. In a more general scenario, an agent could be considered any software module that produces an outcome that is consumed by other software components.

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.

The playback support is crucial when multiple visualizations of different system components are built. It allows users the ability to quickly and accurately isolate individual application components at particular instants in time. Furthermore, the ability to correlate different attributes from multiple visualizations by playing them back in time is also an invaluable aid for software debugging and system understanding.

Constructing tools that meet all these requirements is difficult in itself. The desire to make it general and easy to use is even more complex, and is the focus of our work. Usually, building a software visualization tool 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.

VizScript simplifies both instrumentation and scene definition by introducing a reasoning system between the two. The reasoning system enables the instrumentation component to produce simple information and enables visualization authors to 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 other words, authors do not need to understand the underlying software in order to write the patterns that detect the interesting events to be visualized, lowering the cost of creating the visualizations. Notice that we are not only proposing a set of visualizations tools to understand software activity, we are in fact also introducing a framework that enables rapid creation of visualizations, facilitating programmatic analysis of software (e.g., summary statistics, identification of patterns, etc), system development and application debugging.

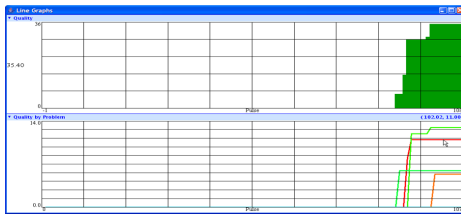
Before fully introducing *VizScript*, we present on the next section some examples of the visualizations that *VizScript* can easily generate: graphs to show the evolution of numeric variables, and charts to show the evolution of nominal variables; introducing also some of the properties of such visualizations. We then present the high-level architecture of *VizScript* along with the software instrumentation. The next section introduces the *VizScript* language with a detailed example to show how visualizations are specified, describing also the details of our reasoning system. 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

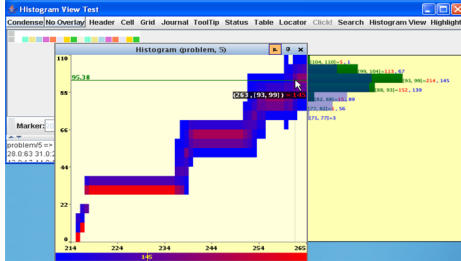
This section presents examples of the types of visualizations that can be generated through *VizScript*. First, we consider visualizations from the *CSC* multi-agent system, where *VizScript* has been used extensively. Then, we present an example of *VizScript* using a different distributed application. Finally, we show again visualizations of the *CSC* system but connecting *VizScript* to different graphics libraries to show the generality of our approach.

CSC Multi-agent System Visualizations

In this complex Multi-agent application, agents help people manage activities they need to perform in order to meet an objective. Each agent has knowledge only of the activi-



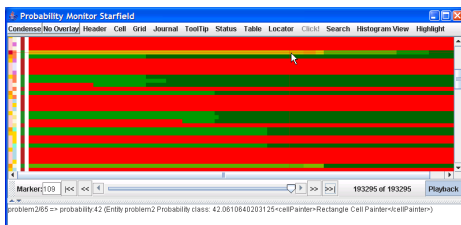
(a) Quality Views



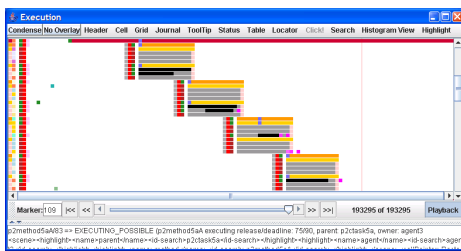
(b) Histogram View



(c) Agent Activities View



(d) Probability View



(e) Execution View

Figure 1: CSC Example Visualizations

ties 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, utilities, etc. Each scenario is conformed of subproblems that contribute to the overall final quality. The domains solved by this multi-agent system are characterized by uncertainty, dynamism, complex reward functions and distributed information. Therefore, determining appropriate actions becomes quite difficult, especially

as the number of agents and the coupling between them increases. *VizScript* greatly helps in understanding the behavior and interactions among agents in this system. Evaluating this system involves analyzing decisions over time, and thus, the needed visualizations are sophisticated animations. Figure 1 shows shots of 5 out of 40 of the visualizations generated with *VizScript* for this particular application. The visualizations are generated using *VizScript*'s embedded graphics library, which is an augmented version of the *Starfields* graphics package from previous work [19]. A full animation of the visualizations can be seen at [18].

The Quality View visualizations on Figure 1(a) show the evolution of task achievement that the multi-agent team has accomplished thus far in the scenario in total and by problem. The horizontal axis represents time and the vertical axis represents the total quality accumulated.

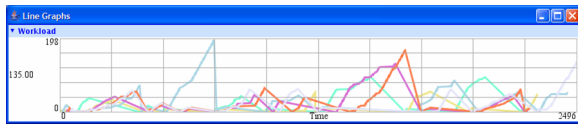
The Histogram View visualization on Figure 1(b) shows the overall distribution of quality for executing an activity in the schedule. The visualization helps to identify the benefits of multi-method changes to the schedule, and the impact of those changes on the overall quality of a problem.

The next view on Figure 1(c) 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 succeeded or failed. The visualization integrates reports from every agent when they start and end activities, or change their priorities.

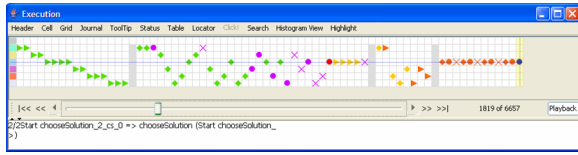
The Probability View visualization from Figure 1(d) shows the evolution of the probability of successfully executing activities. Rows represent activities and columns represent simulated time. Cell (i, j) is colored to show the probability of successfully executing *activity_i* at simulated time *time_j*.

Finally, the Execution View visualization on Figure 1(e) shows an integrated view of all activities from all agents. 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.

As the simulation advances, all the displays update simultaneously. More importantly, the visualizations can be played back in time. Using the time sliders, the user can move time backwards (and forwards) to view the state of the system 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 the effect it has on the current schedule. Furthermore, the graphics library provides the ability to search and cross-reference activities and plan components, offering the capability of isolating activities for



(a) Agent Load View



(b) Agent Execution View

Figure 2: DCOP Visualizations

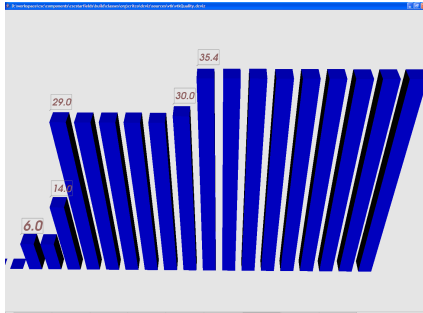


Figure 3: VTK Cumulative Quality View

further analysis. The visualizations created by *VizScript* can be generated online while the system is running, or by processing input data in offline mode. *VizScript* functionality and characteristics facilitates rapid creation and modification of visualizations that greatly helps in understanding and debugging complex software.

Visualizations From a DCOP Application

One of the main characteristics of *VizScript* is the generality of the approach. In this subsection, we present some visualizations generated by *VizScript* for a distributed constraint optimization application (*DCOP*). On this application, agents are trying to coordinate to solve graph coloring problems. Two examples of the visualizations generated by *VizScript* are shown on Figure 2. The Agent Load View on Figure 2(a) shows the amount of work load each agent has across time. The second visualization shows the type of work each agent is performing at each pulse. Different shapes in the visualization represent different events (e.g., send/receive messages, etc). Colors represent different kinds of messages or procedures. On this application, *VizScript* was ran in full online mode.

Visualizations Using Different Graphics Libraries

So far, we have seen examples of visualizations generated by *VizScript* in different applications but using only the *Starfields* graphics package. However, the *VizScript* interpreter is independent of the graphics library. To demonstrate this capability, we show in this subsection two examples of external graphics libraries connecting to *VizScript*. The first example can be seen in Figure 3. It is a 3D visualization generated with the *VTK* graphics library [16]. *VTK* is a very popular open source library for visualization, computer

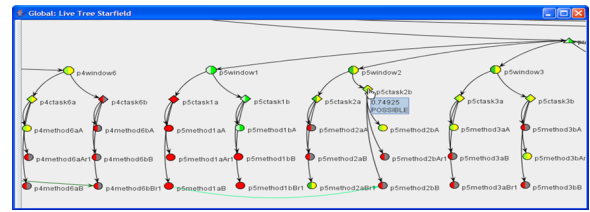


Figure 4: *LiveTree* Hierarchy View

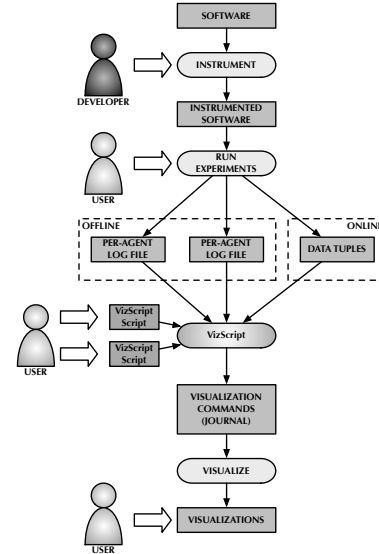


Figure 5: *VizScript* Framework.

graphics and imaging. The example displays the cumulative quality obtained by the system over time. Notice that the data displayed by *VTK*, exactly matches the one visualized previously through *Starfields* on Figure 1(a).

The second example displayed in Figure 4 uses a graph framework (*LiveTree*) to represent hierarchical relations of tasks. This *LiveTree* application is derived from the *JUNG* framework [8], an open source library for modeling data as graphs or networks. It shows hierarchical relations among activities for a scheduling problem using a tree view. On this application, *VizScript* is used to perform analysis on the raw problem data. The results of the *VizScript* analysis are then used to generate values for the nodes in the tree. An example of the *LiveTree* visualization using *VizScript* can be seen on Figure 4. On this example, nodes are composed of two basic colors representing the probability that the current activity gets scheduled and the probability it gets quality.

VizScript ARCHITECTURE

In this section, we provide a high level overview of the main components of the *VizScript* architecture. First, Figure 5 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

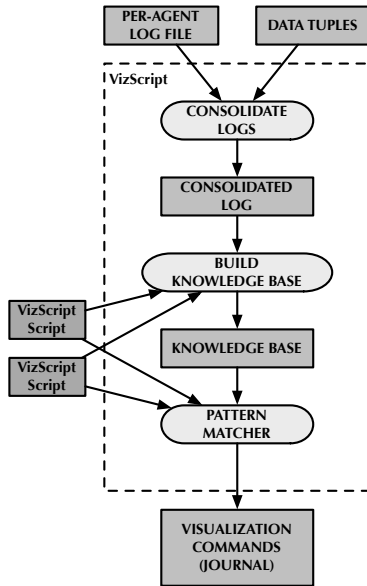


Figure 6: Components of the *VizScript* interpreter.

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 later in offline mode.

To produce visualizations, users write scripts that define what they want to see. *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. In online mode, users specify the scripts before running the application, and the continuous stream of records produced are fed directly into the *VizScript* engine for visualization. In offline mode, users invoke *VizScript* with a set of scripts and files containing the data streams saved. This enables users to run *VizScript* multiple times to produce alternative visualizations on the data. One of the capabilities of the *VizScript* architecture is that integrates information from distributed sources seamlessly. Notice that if the application is fully distributed, we could also use *VizScript* independently at each distributed source. If this were to be the case, each distributed component would visualize its own stream of records in isolation.

The architecture of the *VizScript* interpreter is shown in Figure 6. The interpreter takes as input a collection of data streams, one per agent, and a collection of scripts that define the visualizations that the user wants to see. The first step is to consolidate the data streams produced by the agents. This process is presented in more detail in later sections. The consolidated data streams are used to build a knowledge-base that contains the information needed for visualization. The knowledge-base gets built containing only the information referenced in the input scripts. This is important to improve the performance of the interpreter because the data streams often contain over 100K records, many of which are not relevant to the visualizations required by the user.

```

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
  
```

Figure 7: Example data records

Scripts are defined using rules. The condition part of a rule is a pattern that is matched against the contents of the knowledge-base. *VizScript* builds the knowledge-base one record at a time. After each record is added, *VizScript* fires all the matching rules in all scripts. The output of *VizScript* is a sequence of visualizations commands that are fed to a graphics library to drive the display. The *VizScript* interpreter is independent of the graphics library, and could be used with different graphics packages. The scripts do depend on the graphics library as they explicitly invoke procedures from it.

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: (a) an **Agent** who made the change to the object. Notice that for non multi-agent systems this value could correspond to the component firing the event. (b) a **Time-Stamp** representing the time when the event happened; (c) the **Object** that identifies the object changed; (d) a **Property** representing the name of the object property changed; and (e) a **Value** that is the new value of the object property.

Our representation is similar to RDF triples, 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.

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, so there will be 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. Figure 7 shows the first 5 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 a pattern matcher and an integrated knowledge-base. *VizScript* statements manipulate scalars and associative arrays using standard assignment, arithmetic, conditional and function call expressions. Functions are provided to invoke commands in the graphics library and to update the knowledge-base. The ability to update the knowledge-base is very important, since it allows *VizScript* to do further pattern analysis and more complex inference. The interpreted language is similar to other scripting languages such as JavaScript. The pattern matcher is similar to the pattern matcher in OPS5 [7]. We describe the language by first introducing an example and then present the different language constructs in more detail.

Script Example

Figure 8 shows the complete *VizScript* script for the Probability View visualization from Figure 1(d). Line 3 specifies the title of the window. Lines 5 to 7 include some external libraries to initialize starfields. The basic setup for the Probability Starfield is done from lines 9 to 15. Line 9 tells the engine to sort components in terms of entities (i.e., the methods and tasks in the schedule). Line 10 adds a column to the display to represent the agents, next line adds a column to keep track of the status of an entity. Lines 13 to 15 generate a color map distributed uniformly for 100 bins.

The rest of the script defines the rules needed to process the records in the data stream. The first rule in lines 17 to 21 specifies when to add rows to the visualization. The script adds rows when a new entity of type method or task gets recorded in the knowledge-base, if the rest of the information related to such entity is already present (e.g., the owner, release time, etc). The body of the rule specifies the way the new row is inserted in the display. First, you consider the entity for the sorting mechanism. Then, the script generates a unique color for the matching agent, and the following statement adds the row for that agent. The last rule specifies when to add cells to the starfields columns. We are interested in displaying the probability of success of any entity as time clock advances, and whose execution status is not marked as possible. If the pattern is satisfied, the color for the prob-

```

1 starfield{
2   // Set window title
3   window["title"] = "Probability Monitor Starfield";
4   // Basic starfield settings
5   #include "../include/sfSetting.dcviz"
6   #include "../include/pulseRow.dcviz"
7   #include "../include/entityStatusUpdater.dcviz"
8   // Basic starfield setup
9   addTreeSorter("entitySorter");
10  addColumn("agent");
11  addColumn("entityStatus");
12  // Probability color map from RED to GREEN
13  bins = 100;
14  createColorMap("probability", "#FF0000", "#009900", bins);
15  createStatusColorsFromMap("probability");
16  // Add methods
17  when ((?entity "entityType" "entityTypeMethod") ||
18        (?entity "entityType" "entityTypeTask"))
19        && (?entity "supertaskRelation" ?parent)
20        && (?entity "releaseTime" ?release)
21        && (?entity "entityOwner" ?owner)
22  {
23    addTreeSorterRow("entitySorter", ?parent, ?release,
24                    ?entity, ?entity, ?entity."entityType");
25    generateStatusColor(?owner);
26    addCell(?entity, "agent", ?owner);
27  }
28  // Add probability
29  when("clock" "starfieldPulse" ?cpulse)
30    &&! (?entity "smExecutionStatus" "QUALITY_POSSIBLE")
31    && (?entity "pSchedule" ?prob)
32  {
33    probColor=getColorFromMap("probability", ?prob, bins);
34    addCell(?entity, ?cpulse, probColor, "Entity " +
35            ?entity+" Probability class: "+?prob*bins);
36  }
37 }

```

Figure 8: *VizScript* script for the Probability View

ability of success of an entity is generated in line 33 from the distribution of colors previously created. These patterns illustrate the use of Boolean expressions and variable bindings in the language. Lines 34 and 35 add the new cell. It adds the cell in the row corresponding to the entity, and at the column given by the current time pulse when the event happened, with the color obtained before. The `addCell` statement also composes a tool-tip that contains the entity name and its probability of success.

Knowledge-Base

The core programming system in *VizScript* can be considered a full knowledge-base system, since it includes a declarative scripting language to access, query, and apply deductive reasoning consistently to the information recorded efficiently in its pattern recognition database.² The knowledge-base in *VizScript* is in general a frame system. Where the frames correspond to the objects in the data streams, and the slots in the frames correspond to the object properties filled with different values [13, 15].

The knowledge-base is built from the consolidated data. The streams are conformed of tuples of the form (*agent, timestamp, object, property, value*) as exemplified in the Figure 7. Every time a tuple 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. The current implementation of *VizScript* accepts multi-valued slots.

²See [21] for an explanatory definition of knowledge-base systems

Consequently, when processing a tuple naming an object-property pair for which a slot is already defined, the existing slot gets partitioned to accommodate for 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:

```
bins = 100;
a = bins + 1;
```

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

```
x[a, 1] = "yes";
x[2] = "title";
window[x[2]] = "Probability Monitor Starfield";
```

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"
?entity."entityType"
```

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

Statements

VizScript defines four types of statements: *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

Patterns specify events of interest. A pattern is a boolean combination of primitive patterns. *VizScript* supports conjunction, disjunction and negation. A primitive pattern is of

the form (*object,property,value*), Where the *object* and *value* elements can be variables of the form ?*x*, or expressions that evaluate to a scalar or an object in the knowledge-base. The *property* is an expression, which when evaluated yields the name of a *property* in the knowledge-base. Properties define relations between objects and values, providing the basis for the deductive mechanisms of *VizScript*. For example, the following are primitive patterns:

```
("task1" "probability" ?p)
(?x "probability" ?p)
(?x "probability" 0)
```

A binding is an assignment of values to the variables in a pattern that make the pattern true. A pattern can have multiple bindings when multiple assignments of variables satisfy it, and *VizScript* computes all of them. 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.

Structure of a *VizScript* Script

As illustrated in Figure 8, a *VizScript* script consists of a set of optional global statements followed by a sequence of `when` statement rules. The global statements typically define the appearance of the window, symbolic names for colors, and often invokes functions to initialize the visualization (e.g., sorting mechanism, predefined rows or columns, etc). On the other hand, the `when` statements constitute the core reasoning mechanism of the *VizScript* interpreter. They have the following structure:

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

The optional `where` clause of the `when` statement allows definition of additional constraints that cannot be expressed through the unification of pattern variables (e.g., inequality constraints), and that need to be satisfied for the variable bindings. If a particular `when` clause is satisfied, then its body gets evaluated updating the visualizations.

VizScript first executes the global statements, and then consumes the consolidated input stream one record at a time. For each record it first adds the record to the knowledge-base, if it refers to a property used in a script. Then, it selects the `when` statements to evaluate, considering only those related to the record just added to avoid computing useless bindings. Finally, evaluates the selected `when` statements, processing the body of the rule if the variable bindings of the patterns are satisfied against the knowledge-base.

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" 0) { ... }
```

VizScript would evaluate the first `when` statement because it contains the property "probability". It would bind `?x` to "task1" and `?p` to 0.3. It would not evaluate the second statement 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 binding the pattern variables based on the record just added to the knowledge-base, *VizScript* computes bindings for any variables that are still unbound in the `when` statement by matching against the full contents of the knowledge-base. As mentioned before, the bindings must also satisfy the `BooleanExpression` in the `where` clause if it has been defined to finally execute the body of the rule. *VizScript* executes the body of the rule once for each possible set of bindings. This powerful feature enables updating multiple objects on the screen when some key property changes.

As mentioned before, *VizScript* introduces a `let` statement to query the knowledge-base. Unlike the `when` statement, the bindings it produces are not required to contain the record just added. The following example illustrates the use of the `let` statement in the Agent View visualization from Figure 1(c):

```
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 when the simulation clock advances. The `let` statement fetches all "Primitive" activities with status "qualityChanging", binding "`?p`" to the execution priority of the activity. The action paints the corresponding cells with a color denoting the priority.

The body of `when` statements typically contains commands to drive the underlying graphics package, compute statistics, summarize information or update the knowledge-base. Therefore, the *VizScript* language not only makes sense of the knowledge contained in the data set, but also has the ability to enhance it to support more complex pattern analysis. These properties make *VizScript* be a very powerful tool for redirecting users' attention to those important patterns in the visualizations.

EVALUATION

We developed *VizScript* because we need powerful, easy to use visualizations tools to help us understand, debug and de-

Visualization	Lines of Code		Savings Factor
	Original	VizScript	
Quality View	122	5	24.40
Agent View	190	47	4.04
Probability View	258	18	13.57
Execution View	1214	93	13.05

Table 1: Effective Lines of Code for Visualizations

velop a large multi-agent application. The visualizations are used by many team members every day. Our evaluation is based on our experience re-implementing our visualizations using *VizScript*. Because we rely on our visualizations to make progress on our multi-agent system, the *VizScript* visualizations must be at least as good as our previous, custom Java implementations:

Capability: the new visualizations are either equivalent or more detailed than the old ones. Furthermore, *VizScript* provides the functionality of a deductive system, easily identifying patterns to draw users' attention in the visualizations.

Instrumentation cost: negligible in both implementations.

Offline mode: supported in *VizScript*, but not in the old implementation.

Rendering: the *VizScript* visualizations often take 2 times longer to render than the custom Java implementations (e.g., about 20 seconds instead of 10 seconds for a medium size 10 agent scenario).

Even though the rendering costs are higher, the offline mode supports a concept of operation that compensates for the slower performance. Without offline support we had to run simulations interactively in order to visualize them. We had to wait for both the simulation and the visualization to run. With offline support we run many simulations in batch mode and select the most interesting ones to visualize. We don't have to wait for the simulations to run, which often takes much longer than the time to visualize the results.

We have not ran full user studies yet, so we cannot yet prove completely that *VizScript* is easy to use. However, the preliminary experience with *VizScript* is very encouraging. The DCOP visualization examples presented in the example section were generated by an independent group of students with the collaboration of only one of the authors. Furthermore, the port to the *VTK* graphics package and the visualization example were carried out in a few hours of coding. To quantitatively express the benefits of *VizScript*, we considered the lines of code that went in to creating various of the *CSC* visualizations discussed previously. We measure the effective lines of code (eLOC) [6]³ required to implement four different visualizations using the old custom Java approach and the new *VizScript* approach. Table 1 shows that *VizScript* enables approximately an order-of-magnitude reduction in the number of lines necessary to create these visualizations.

³eLOC is defined as the total lines of code (no blank lines or comments) excluding stand-alone braces and parenthesis.

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 2: Logging Costs for VizScript

Three additional points are worth noting. (1) a useful visualization can be specified in about one page of *VizScript* code; (2) two of the four visualizations were not re-implemented by the developers of the original custom-Java visualizations; and (3) *VizScript* proved to be practical for data streams containing over one million of data records. For example, Table 2 shows the logging costs of *VizScript* for different problems size. The second column shows the number of lines in the consolidated data stream. The third and fourth columns show the size of the unzipped and zipped files. While the raw files can grow large, the zipped versions are very manageable. Furthermore, the time to read and process the largest zipped file in the table took only 14.40 seconds. We plan on identifying the bottlenecks and optimizing performance *VizScript* in order to be able to process data streams with many millions of records in the next phase of work.

RELATED WORK

The problem of automating the design of graphical presentations has been widely recognized. Mackinlay’s work on applying a graphical theory to support the automatic design of presentations is well known [11]. While his work considers only static information, one of the main contributions of our work is the support for dynamic and continuous data to integrate information from multiple sources. A more similar approach to *VizScript* is the information visualization spreadsheet from Chi *et al* [3]. Their work considers spreadsheets to enable the display and exploration of information. Their approach, built on the top of *VTK* and *Tcl* scripting language, is more tailored to visualizing a wide variety of data. Our approach is more concerned with the identification and visualization of critical patterns in data, making it more suitable for applications with a high degree of interrelations.

Demetrescu *et al* [5] 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.

InspectJ [9] is a program visualization system that uses AspectJ 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 [10]’s work 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.

Tominski and Schumann [20] 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 with respect to time.

IVEE [1] is 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 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 information across time. Another difference is the fact that the set of visualizations in *VizScript* is not predefined, but user customized through the language.

ZEUS [14] is an agent building toolkit for rapid constructing collaborative agent applications. It supplies a visual environment for capturing user specifications of agents. It also has some built-in visualizations to depict the running status and statistic information of the system. Unlike *VizScript*, *ZEUS* is more like a building and modeling tool instead of a visual tool for understanding the complex behavior of multi-agent systems. Its visualizations are hard-coded, and can only be used within *ZEUS*.

Brahms [4] is a modeling and simulation tool. One of its components, *AgentViewer*, can be used to visualize objects from the simulation, including agents, their locations, and communications. It is driven by a database that stores all the simulation events. This approach is generic, and the visualization display is rich. However, its power is limited by the initial design of *AgentViewer*. Whenever users want to see additional information in the visualizations, *AgentViewer* has to be re-implemented by system experts, which requires lots of time and effort. Our approach uses a simple script lan-

guage to build visualizations dynamically and quickly, without requiring extensive expertise on the actual system.

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 such visualizations. By combining a generic instrumentation, 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 toward 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 different types of visualizations depending of the graphics package used. Line charts, starfields, and 3D graphics are possible. We expect to incorporate more graphics libraries to demonstrate additional generality of our 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 logging costs incurred in storing the data streams and processing the visualizations for small, medium and large problems. We showed that the 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 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. Ongoing work also focuses on further scale-up to data streams with several million records by identifying the bottlenecks and optimization factors of *VizScript*. Despite *VizScript* current limitations, *VizScript* is a promising and effective approach to creating dynamic

visualizations for complex, large, and distributed systems. *VizScript* represents a significant step toward satisfying the desiderata listed in the introduction of this paper.

REFERENCES

1. C. Ahlberg and E. Wistrand. Ivey: 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. E. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *IEEE Information Visualization '97*, 1997.
4. W. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof. Brahms: Simulating practice for work systems design. *International Journal of Human-Computer Studies*, 49:831–865, 1998.
5. 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.
6. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 1998.
7. Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
8. J.O'Madadhain, D. Fisher, S. White, and Y. Boey. The jung (java universal network/graph) framework. Technical Report 03-17, University of California, Irvine-School of Information and Computer Science, October 2003.
9. Rilla Khaled, James Noble, and Robert Biddle. Inspect j: Program monitoring for visualisation using aspectj.
10. Yingsha Liao and Donald Cohen. A specification approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.*, 18(11):969–978, 1992.
11. J. Mackinlay. Applying a theory of graphical presentation to the graphic design of user interfaces. In *1st Annual ACM SIGGRAPH Symposium on User Interface Software*, 1988.
12. Rajiv Maheswaran, Craig Rogers, Romeo Sanchez, and Pedro Szekely. Distributed coordination in uncertain multiagent systems. In *6th. International Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*, 2007.
13. Marvin Minsky. *The Psychology of Computer Vision*, pages 211–277. New York: McGraw-Hill, p. winston edition, 1975.
14. Hyacinth S Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: a toolkit and approach for building distributed multi-agent systems. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 360–361, 1999.
15. Fikes R.E. and T. Kehler. The role of frame-based representation in knowledge representation and reasoning. *Communications of the ACM*, 28(9):904–920, 1985.
16. W. Schroeder, L. Avila, and W. Hoffman. Visualizing with vtk: A tutorial. *IEEE Computer Graphics and Applications*, 20:20–27, 2000.
17. John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.
18. P. Szekely, M. Becker, S. Fitzpatrick, G. Gati, D. Hanak, J. Jin, G. Karsai, R.T. Maheswaran, R. Neches, C.M. Rogers, R. Sanchez, and C. VanBuskirk. Csc: Criticality-sensitive coordination. In *5th. International Conference on Autonomous Agents and Multiagent Systems (AAMAS'06), Demo Session*, 2006. Demo at: <http://www.ist.edu/~szekely/csc/aaamas06/csc-demo-v01.html>.
19. 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.
20. 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.
21. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems Volume II: The New Technologies*, chapter 16, pages 982–983. W.H. Freeman and Company, 1989.