OverView - Dynamic Visualization of Java-Based Highly Reconfigurable Distributed Systems*

Harihar Narasimha Iyer, Abe Stephens†, Travis Desell and Carlos Varela
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
http://www.cs.rpi.edu/wwc/

Abstract

Online visualization enables developers to test, debug, and monitor the behavior of distributed systems, while they are running. While important in software development, online visualization of distributed systems is largely unaddressed by conventional tools. Distributed systems are often programmed using high-level abstractions that facilitate reasoning about them, e.g., actors, processes, sessions, or ambients. OverView is an entity specification language-driven Eclipse plug-in for visualization of distributed systems that preserves the high level of abstraction, and enables online visualization of critical distributed system properties such as component naming, location, remote communication, and migration. OverView's architecture is generic in that different abstractions can reuse the visualization module requiring changes only in the entity specifications that drive the visualization process.

1 Introduction

Visualizing, testing, and debugging distributed systems is a challenging task that is largely unaddressed by conventional software tools. Many distributed systems are designed using abstractions that create a unified view of the individual components and their locations in the system. This unified view is especially useful when mobility is considered and system components may be reconfigured during a program's lifespan. Conventional profiling and debugging tools for Java environments, such as those distributed with the Eclipse development platform restrict a developer to examine one virtual machine at a time. To examine an entire distributed program with these tools a developer must use multiple debugger instances, individually attaching each to a single virtual machine. This division makes the global state of the system difficult to intuitively understand and complicates testing because the unified view provided by the developer's original design is either obscured or not present.

Several problems arise when attempting to visualize the online state of a distributed system designed with high-level abstractions such as actors, processes, or mobile ambients; since most tools lack first class support for these abstractions. Conventional profilers, for instance provide numerical summaries of dynamic information such as the length of time spent executing a method or information at the level of objects and classes. In order to analyze the system at the higher level, developers must determine which lower level objects represent their higher-level abstractions at runtime. This process is time consuming, and partially decreases the utility of using the higher-level abstraction; especially if a considerable amount of visualization is conducted.

Eclipse is an open source IDE that has an extensible architecture. The IDE can be extended by providing modules called plug-ins, which provide the developer with a specific tool. Plug-ins are coded in Java and integrate well into the Eclipse Platform. OverView provides distributed systems developers and users with a tool to view the global state of the system at any point of time. It supports both online dynamic visualization as well as an offline approach where the events in the distributed system can be recorded and replayed at a future time.

OverView tries to address the challenge of visualizing, testing, and debugging distributed systems by providing a consistent global view of the entire system. In dynamically reconfigurable distributed systems, some components may be migrated (manually by programmers, or autonomously by middleware layers) to different locations due to load-balancies policies or failures in the underlying network topology.

*Work partially supported by an IBM Eclipse Innovation Grant

†Currently affiliated with the University of Utah.

Visualization Tool	*Distributed Program Execution*	*Online Visualization*	*High Level Abstractions*
DejaVu	Y	N	N
Jinsight	N	N	N
Jive	N	Y	N
Hy+	Y	N	N
OverView	Y	Y	Y

Table 1: Different types of program visualization.

Model	*Entity*	*Communication Event*	*Container*
Actors	Actor	Message Send	Theater
Mobile Ambients	Process	Channel	Ambient
Petri Nets	Token	Transition	Place
RMI/CORBA	Object	Method Invocation	JVM
J2EE	JavaBean	HTTP Request	Container

Table 2: Sample event types for distributed programming models.

If the components of the distributed system migrate from one location to the other, the plug-in will reflect these changes, allowing the user to analyze and study the system. In the same way that the coding and debugging tools in Eclipse make writing software more accessible by visually representing a program's static information, such as packages, classes, and interfaces, as well as a program's dynamic information, such as objects, threads, and invocation stacks; OverView makes programming a distributed system more accessible to the programmer by creating an analogous visual workspace with appropriate distributed computing abstractions, including component naming, location, remote communication, and migration.

2 Related Work

Considerable work has been done in the visualization and analysis of the execution of Java programs (see, e.g. [7]). Jinsight [10] does visualization of trace information produced by a special instrumented version of the Java Virtual Machine. Similarly, Walker et al. [14] use program event traces to visualize program execution patterns and event-based object relationships like method invocations. All these systems support only an offline mode where the trace of the program execution gets visualized. Also they are not designed for visualizing a distributed system.

Jive [8] does on-line visualization of Java programs. It is a software visualization system that has a framework for dynamic analysis of program data.As in the systems mentioned before, Jive also supports only non-distributed systems.

DejaVu [2, 5] is a Deterministic Java Replay Utility that supports understanding and debugging distributed Java applications through deterministic replay of non-deterministic execution. DejaVu does not support dynamic visualization but rather a replay of the execution of the different VMs. The Hy+ system proposed in [3] helps to understand and debug a distributed program by replaying traces recorded at runtime. Snodgrass presents a method targeted towards distributed debugging and monitoring in which a programmer uses relational algebraic queries to track run-time dynamics [11].

Most of the visualizations mentioned above show fine-grained execution information about individual classes and objects. They lack a mapping from the low-level objects to high-level abstractions. Sefika et al [9] allow the developer to utilize coarse-grained system information to produce visualizations. In their technique, a developer may introduce abstractions into the system instrumentation process. The abstractions can then be used as a basis for several visualizations. Walker et al [14] describe a system, which does visualization in terms of a high-level view of the system that is selected by the user. It permits lightweight changes to the abstraction used for condensing the dynamic information. Queries of dynamic program information were used by Lencenvicius [6] to debug online programs.

Our work is unique in that it supports both an online and offline visualization of a distributed system. It also allows an easy mapping of the high-level abstractions through a specification language. Table 1 shows the different systems discussed above.

3 Entity Specification Language

OverView's Entity Specification Language (ESL) provides the ability to map high-level programming abstractions to low level Java code. The user defines entities, and events based on actions made by those entities. We based these entities on the concepts of distributed computing models, namely entity creation/deletion, containment, migration, state change, communication and errors. By mapping these concepts to low level Java code,the ESL provides a universal way to translate high level abstractions to low level Java code.This provides a homogeneous model for visualization. The simplicity of the ESL model is reflected in the simplicity of the language, as shown in Figure 1 and Figure 2.

The ESL provides three types of declarations for an

```
Entity              ::= entity IDENTIFIER is Name EntityBody
EntityBody          ::= { UniqueByDeclaration (WatchDeclaration | WhenDeclaration)* }
UniqueByDeclaration ::= unique by Value ;
WatchDeclaration    ::= watch IDENTIFIER is Value ;
WhenDeclaration     ::= when (start | finish) MethodSpecification send EventDeclaration
                        [ExceptionSpecification , ExceptionSpecification] ;
MethodSpecification ::= IDENTIFIER ( [Parameter  (, Parameter)*] )
ExceptionSpecification ::= on exception IDENTIFIER send EventDeclaration
Parameter           ::= Name IDENTIFIER
Name                ::= IDENTIFIER (. IDENTIFIER)*
EventDeclaration    ::= EventType (( [Value (, Value)*] )
EventType           ::= Creation | Deletion | Migration | Update | Communication | Error
Value               ::= LITERAL
                        exception
                        | (entity | IDENTIFIER) (. ValuePart)+
ValuePart           ::= IDENTIFIER [( [Value (, Value)*] )]
```

Figure 1: Entity Specification Language Grammar

```
entity Actor is salsa.language.Actor {
   unique by entity.uan.toString();

   when finish bind(UAN uan, UAL ual)
      send Creation( ual.toString() );
   when finish bind(UAN uan) send
      Creation( entity.ual.toString() );

   when finish finalize()
      send Deletion();

   when finish migrate(UAL ual)
         send Migration( ual.toString() )
      on exception MigrationException
         send Error( exception ),
      on exception MalformedUALException
         send Error( exception );

   when finish send(Message message)
      send Communication( message.getTargetString() );
}
```

Figure 2: ESL example for SALSA Actors.

entity:

- A **unique by** declaration provides the ability to declare multiple objects as a single identity, as an object in a distributed system may be represented by multiple objects across various JVMs during its life span. This declaration specifies a unique string identifier to describe the entity.
- A **contained by** declaration provides support in the visualization for one entity to be contained within another and be visualized as such.
- A **watch** declaration specifies that a specific attribute at the visualization level should reflect all changes to a member of a specific Java instance. This attribute will be monitored by OverView and events will be sent to the visualization layer whenever the monitored object or attribute is modified.
- A **when** declaration describes triggers for events, based on actions performed by the entity. **start** and **finish** designate if the event should be triggered at the beginning or end of the method invocation (or a constructor). The declaration then proceeds to describe the event sent to the visualization, as well as the values that the event will contain. The optional **on exception** designation specifies what action to take if an exception is thrown by the watched method or constructor.

The six events that can be specified with a when declaration are:

- **Creation(String containerId)** This specifies the creation of an entity, taking the container or entity that is contained by as an argument.

- **Deletion()** This specifies a deletion of an entity.
- **Migration(String targetContainerId)** This specifies that an entity has moved from one container to another, taking the container which the entity has moved to as an argument.
- **Communication(String entityId)** This specifies that communication occured between two entities, and takes the entity communicated with as an argument.
- **Error(Exception exception)** This specifies that an error occured at some entity and takes the exception thrown as an argument.
- **Update(String item, Object value)** This specifies that the state of an entity has been updated, along with the value that was updated and the identifier for that object. These events are usually used behind the scenes by the watch declaration. The arguments for this event are the name of the part of the state that was changed, as well as the value that is has now become.

Figure 2 shows a sample entity specification for the Actor Model [1], using the SALSA Language [13]. In the SALSA language, an actor *b*inds to a location, thus entering the distributed system. After this, the actor can *migrate* to other locations in the system. Currently, OverView requires IDs for entities be Java Strings, so the specification reflects this. The specification also designates how exceptions are sent to the visualization layers. The `entity` keyword denotes that the following identifiers are values or method locals to that entity, if this keyword isn't used, the first identifier is assumed to refer to the arguments of the method which sends a trigger.

4 OverView Architecture

OverView has a highly modular event-based architecture. It consists of four layers: a data collection layer, an event profiling layer, a data-mapping layer and visualization layer (see Figure 3). The entity specification language is used to map high-level concurrency abstractions into lower-level Java threads, network connections and objects.

The data collection layer (DCL) connects to the specified JVMs (Java Virtual Machines) using JPDA (Java Platform Debug Architecture) connections. The data collection layer is responsible for maintaining the connections throughout the course of the visualization. The event-profiling layer (EPL) communicates to the different JVMs through the data collection layer. EPL gets the information from the ESL and translates them

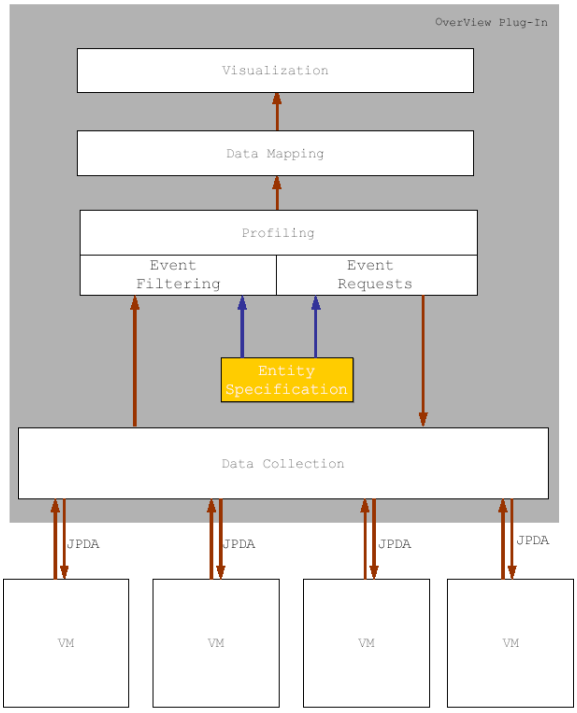

Figure 3: Layered Architecture of OverView.

into Java debugging events like method entry and exit events. Then it sets the requests for these events in the JVMs. When the desired event occurs in the JVM, it sends an event back to the EPL. This event is analyzed by the EPL and the required data is extracted from the JVM. The ESL again provides EPL with the necessary information as to how and what information to extract when the event occurs.

The data that is gathered from the events is passed to the Data Mapping Layer (DML), which modifies a data structure which we refer to as *RawData*. The RawData has data structures to hold the entity state attributes and the entity-entity relationships. The events that are passed on to the DML from the EPL get mapped to specific changes in the RawData structure.

The final layer in the architecture is the visualization layer (VL). The input to VL is the RawData, which gets modified by the DML. The visualization layer takes the RawData and presents it to the users in the form of diagrams. VL has two modes of presentation to the user: online and offline. In the first mode, the RawData gets visualized in real-time. The VL also provides the user with the option to record the events and to visualize them at a later point in time.

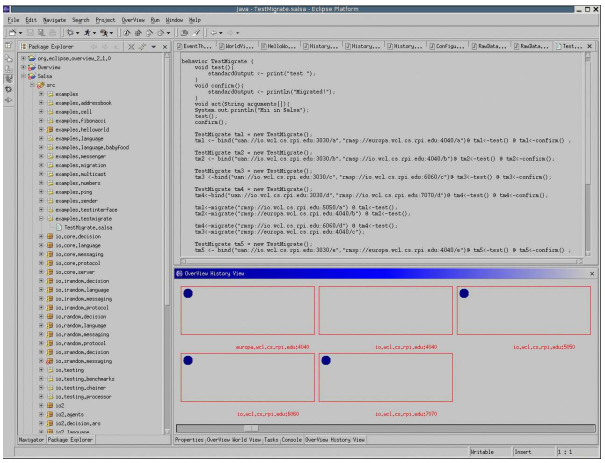

Figure 4: Snapshot of the OverView History View.

This second mode of visualization allows the user to analyze the events at a pace convinient to them.

5 Preliminary Implementation and Results

OverView has been implemented as an Eclipse Plug-In. The system developer uses the ESL to specify the events that he is interested in visualizing and how to get that information. Before program visualization can begin, the connections to the different VMs are made using the JPDA architecture. JPDA uses Java Debug Wire Protocol (JDWP) to communicate between the VM and the DCL. One of the main issues with the current implementation is the suitability of JPDA as a tool for collecting information from different VMs. From our experiments, we have found that JPDA fails to provide necessary information under certain circumstances. This could lead to the state of the system being misrepresented by OverView. Another factor was the speed of execution of the system. Using JPDA to gather information considerably slows down the speed of execution of the whole system. In order to improve on these aspects, we are considering Java byte-code instrumentation as an alternative to JPDA to get information from the various VMs. This would amount to inserting byte code into proper places in the user classes at load time to send the required information to OverView as specified in the ESL. The communication between the DCL and the VMs would then be handled by a custom protocol. However, an advantage of using JPDA is that we could more easily extend OverView as a debugging and distributed systems management framework.

The VL consists of two Eclipse views. The *World View* gives a high level view of the entire system. It visualizes the different containers with the entities inside them making it highly suitable to view entity migrations across the system. The containers are visualized as rectangles with their identifiers below them and the entities are shown as circles inside the containers. The color of the entity is a representation of the "state" of the entity. For example, an entity, which needs more computational resources, will be displayed as a red circle, and an entity which has low resource usage, will be displayed in blue. The World View also provides the user with the choice to record the events in the execution. A "Start Record" and a "Stop Record" option in the World View facilitates this. The *History View* can be invoked to visualize and analyze the program execution in an off-line mode. The History View can be used along with the World View even while the program is executing. Therefore it is possible to visualize the events that already occurred during the execution of the current program.

We tested our implementation on distributed programs written using the SALSA programming language [13]. OverView was able to visualize the actor creation and migration successfully in the different trail runs. It could also capture the state changes of actors, which was specified as the change in the mailbox size of the actor. Figure 4 shows a screenshot of the initial version of the plug-in.

6 Discussion

In this paper, a brief description of the OverView Eclipse plug-in has been presented. The initial prototype of OverView does visualization of creation and migration of distributed entities in the network. Understanding global state is a critical step in developing robust distributed systems with reliability guarantees. OverView not only enables the visualization of distributed systems during run-time, but also enables experimentation with new language constructs for high level distributed systems programming. Programming abstractions for coordination may include hierarchical actor groups [12] and different notions of distributed transactions [4]. The ability to control and manipulate the components of these systems at runtime is also an important and desirable feature; the architecture described here presents a highly extensible first step to that eventual goal.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] B. Alpern, J. Choi, T. Ngo, M. Sridharan, and J. Vlissides. A Perturbation-Free replay platform

for Cross-Optimized multithreaded applications. pages 23–23.

[3] M. P. Consens, M. Z. Hasan, and A. O. Mendelzon. Visualizing and querying distributed event traces with hy+. In *Applications of Databases*, pages 123–141, 1994.

[4] J. Field and C. Varela. Toward a programming model for building reliable systems with distributed state. In *Proceedings of the First International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA).*, Brno, Czech Republic, August 2002.

[5] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. pages 219–228.

[6] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. *Lecture Notes in Computer Science*, 1628:135–??, 1999.

[7] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Software visualization, state-of-the-art survey. *LNCS 2269*, 2002.

[8] S. P. Reiss. Jive: Visualizing java in action demonstration description. 2003.

[9] M. Sefika, A. Sane, and R. Campbell. Architecture-Oriented Visualization. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 31:10, pages 389–405, 1996.

[10] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. March 2001.

[11] R. Snodgrass. A relational approach to monitoring complex systems. 6:2:157–196, May 1988.

[12] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[13] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[14] R. J. Walker, G. C. Murphy, B. N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Conference on Object-Oriented*, pages 271–283, 1998.