OverView: A Framework for Generic Online Visualization of Distributed Systems

Travis Desell Harihar Narasimha Iyer Carlos Varela

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
http://www.cs.rpi.edu/wwc/

Abe Stephens

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112, USA

Abstract

Visualizing, testing and debugging distributed systems is a challenging task that is not well addressed by conventional software tools. OverView, an event-based Eclipse plug-in that provides runtime visualization of systems running on distributed Java virtual machines is presented. In the same way that the coding and debugging tools in Eclipse make writing software more accessible by visually representing both a program's static components: packages, classes, and interfaces, as well as a program's dynamic components: objects, threads, and invocation stacks; OverView intends to make distributed systems more accessible to programmers by creating an analogous visual workspace with appropriate abstractions for distributed component naming, state, location, remote communication, and migration. Overview is a generic visualization framework that uses an Entity Specification Language (ESL) to enable developers to map high-level concurrency and distribution abstractions into lower-level Java threads, network connections and objects.

Key words: distributed systems, online visualization, dynamic program reconfiguration

1 Introduction

Many distributed systems are designed using abstractions that create a unified view of the individual components independently of their locations in the system. This unified view is especially useful for transparency when mobility is considered and system components may be reconfigured during a program's

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Model	Entity	Container	Communication Event	Mobility Event
Actors	Actor	Theater	Message Passing	Actor Migration
Mobile Ambients	Process	Ambient	Ambient Input/Output	Process-Driven Migration
Petri Nets	Token	Place	n/a	Transition Firing
RMI/CORBA	Object	JVM	Method Invocation	n/a
J2EE	JavaBean	Container	HTTP Request	n/a

Table 1
Sample event types for distributed programming models.

life span. Conventional profiling and debugging tools for Java environments, including 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 state of a distributed system designed with high-level abstractions such as actors [1], processes [17,8], or mobile ambients [3]; since most tools lack first class support for these abstractions. Some sample abstractions are shown in Table 1. For instance, conventional profilers 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 at run-time represent their higher-level abstractions. This process is time consuming, and partially decreases the utility of using the higher-level abstraction; especially for visualization.

Most debugging and visualization systems instrument the code to be visualized, resulting in large decreases in program execution performance. This also leads to less accurate visualizations since often the errors and race-conditions that developers are trying to debug will not occur when the system is running at a slower speed. Compounding this performance hit over a large scale system, which may require synchronization between the visualization and visualized JVMs, can seriously reduce the viability of large-scale online distributed system visualization tool. Nonetheless, many distributed systems are long-lived and therefore require online visualization tools.

Our approach is to develop a language for describing a unified view of a

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

Table 2
Different types of program visualization.

distributed system's abstractions. Developers will be able to retain and utilize their high level abstractions when visualizing a distributed system for testing, debugging and optimization purposes. By having a mapping from high level abstractions to low level Java code, a visualization need not be limited to any particular abstraction paradigm. Furthermore, this mapping also enables the instrumentation of code to be targeted to events which are truly significant at the higher-levels of abstraction. This *selective instrumentation* helps to make less intrusive visualization tools, improving scalability.

In dynamically reconfigurable distributed systems, some components may be created, destroyed, or migrated to different locations due to manual commands by programmers and users, or autonomously by middleware layers for purposes such as load-balancing or tolerance to failures in the underlying network topology. Components also may communicate with each other, and the extent and destination of communication is valuable information to distributed systems developers. Providing an informative visualization of the location and behaviors of components in a distributed system can lead to more robust and higher performance systems.

Eclipse is an open source integrated development environment (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 is an Eclipse plug-in that provides dynamically reconfigurable distributed systems programmers and users with a tool to view the global state of the system at any point in 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.

2 Related Work

Considerable work has been done in the field of visualization and analysis of the execution of Java programs (for a survey see, *e.g.* [9]). Jinsight [12] does visualization of trace information produced by a special instrumented version of the Java Virtual Machine. Similarly, the system developed by Walker et al. [16] uses program event traces to visualize program execution patterns and event-based object relationships like method invocations viewed within cels. These systems support only an offline mode where the trace of the program execution gets visualized. Also they are not specifically designed for visualizing a distributed system.

Jive [10] does on-line visualization of Java programs. It is a software visualization framework for dynamic analysis of program data. Jive supports only non-distributed systems. Snodgrass presents a method targeted towards distributed debugging and monitoring in which a programmer uses relational algebraic queries to track run-time dynamics [13].

DejaVu is a Deterministic Java Replay Utility that supports understanding and debugging multi-threaded [2] and distributed [6] 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. Programs are instrumented so that critical scheduling checkpoints can be profiled and re-traced in subsequent analyses. The Hy+ system proposed in [4] helps to understand and debug a distributed program by replaying traces recorded at runtime.

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 [11] 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, with different granularities modeling different aspects of the architecture being visualized. Walker et al [16] describe a system which does visualization in terms of a high-level view of the system 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 Lencevicius [7] to debug online programs, by giving instant error alerts by continuously checking inter-object relationships during the program's runtime.

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 2 compares some of the different systems discussed above.

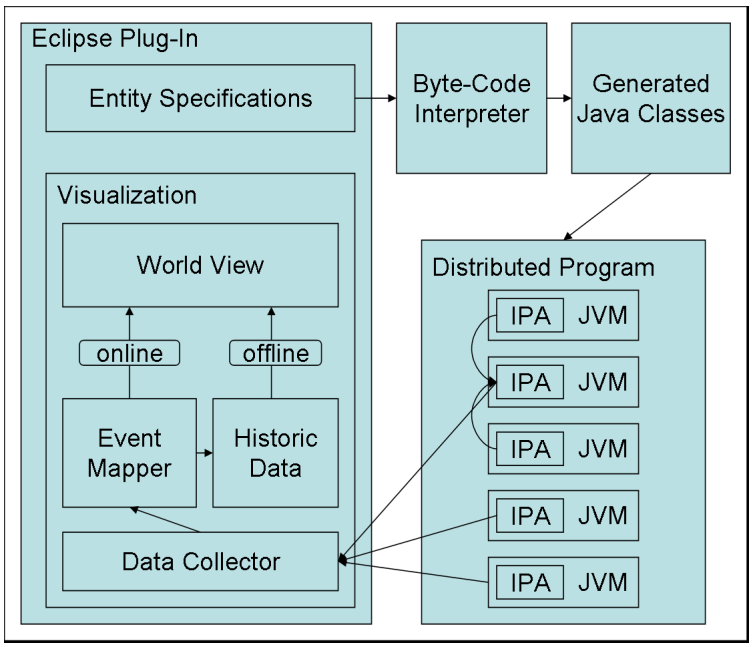

Fig. 1. OverView's Architecture

3 Visualization Architecture

A distributed systems visualization must be efficient and non-intrusive. Many race conditions will not appear when the speed of a program is decreased. We developed the Overview Plug-in Architecture, shown in Figure 1, a model for distributed systems visualization. This architecture attempts to achieve minimal intrusion through OverView's *entity specification language* (ESL). The ESL provides a simple language to map from high-level abstractions to low-level Java code. The ESL is described in depth in section 4. The entity specifications of a distributed system are used by the OverView plug-in to instrument the system's byte-code, inserting profiling statements only when needed and avoiding the large amounts of overhead incurred by other methods (such as the Java Platform Debugging Architecture).

The byte-code instrumentor creates a single instrumented profiling agent (IPA) for every JVM in the distributed program. The OverView plug-in will register it's *data collector*, the component which receives events and forwards them to the plug-in's visualization components, with all the IPAs at JVMs it wishes to visualize. Additionally the plug-in will register some IPAs with other IPAs, instead of with its data collector. These IPAs will act in a hierarchy to condense the information sent to the data collector, reducing the overall load. The IPAs collect profiling information and send this to every registered data collector or IPA. IPAs may also register to other IPAs, creating a hierarchy for condensing the amount of information eventually sent to the data collectors. This approach provides scalability, and allows for multiple OverView plug-ins to visualize a distributed system simultaneously.

```
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
                           | (this | IDENTIFIER) (. ValuePart)+
ValuePart             ::=  IDENTIFIER [( [Value (, Value)*] )]
```

Fig. 2. Entity Specification Language Grammar

The data collector receives the events and sends them to the event mapper, which periodically updates the world view and stores events to the historic data. The historic data can be saved to disk and used to replay the events in offline. The IPA, event mapper, historic data and world view are described in greater detail in section 5.

4 Entity Specification Language

OverView's Entity Specification Language (ESL) allows a developer to defines entities and events based on actions made by those entities. These entities are based on the concepts of distributed computing models; 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 language, as shown in Figures 2 and 3.

The ESL provides five types of declarations for an entity:

- **unique by** 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.
- **contained by** provides support in the visualization for one entity to be contained within another.
- **watch** specifies that a specific attribute at the visualization level should reflect all changes to a member of a specific Java object 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.
- **when** 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

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

   when finish bind(UAN uan, UAL ual) send Creation( ual.toString() );
   when finish bind(UAN uan) send Creation( this.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() );
}
```

Fig. 3. ESL example for SALSA Actors.

proceeds to describe the event sent to the visualization, as well as the values that the event will contain.

- **on exception** specifies what action to take if an exception is thrown by the watched method or constructor (optional).

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 occurred between two entities, and takes the entity communicated with as an argument.
- **error(Exception exception)** This specifies that an error occurred at some entity and takes the exception thrown as an argument.
- **update(String item, Object value)** 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 3 shows a sample entity specification for the actor model [1], using the SALSA programming language [15]. In the SALSA language, an actor *binds* to a location and enters the distributed system. After binding, the actor can *migrate* to other locations in the system. OverView requires IDs

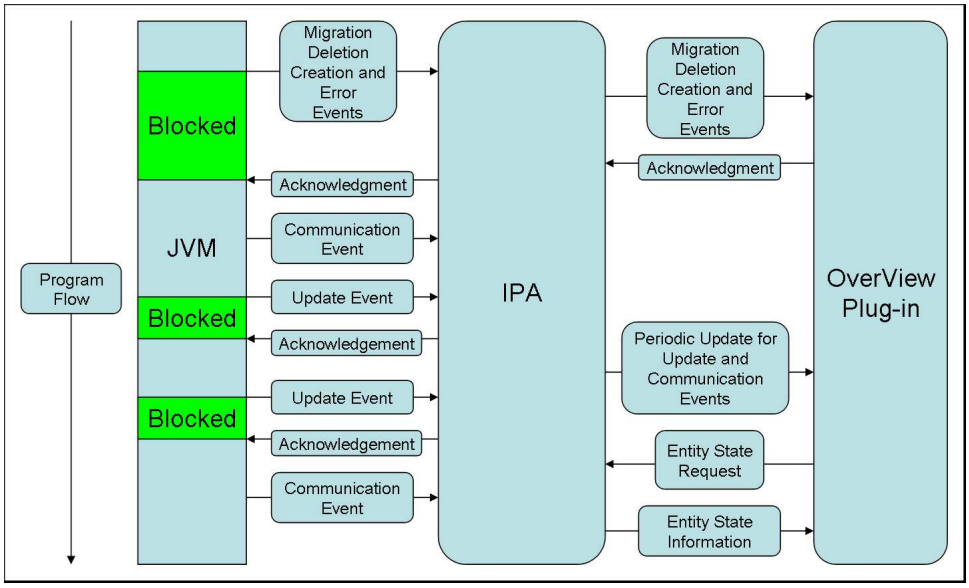

Fig. 4. Program flow for a JVM with a synchronous IPA

for entities be strings. The specification also designates how exceptions are sent to the visualization layers. The **this** keyword denotes that the following identifiers are values or method locals to that entity. If **this** isn't used, the first identifier is assumed to refer to the arguments of the method which sends triggers an event.

5 Preliminary Implementation and Results

5.1 ESL and Byte-code Instrumentation

The OverView plug-in utilizes the JJTree software to parse the ESL and instruments the appropriate Java classes, inserting the appropriate profiling statements for the various events in the ESL. These profiling statements consist of passing information about the above events to the IPA. This allows the OverView plug-in to visualize programs for which the source code is not readily available, as long as there are the appropriate entity specifications.

5.2 Instrumented Profiling Agent

The instrumented profiling agent accepts profiling information from the entities in its JVM through the *instrumented profiling statements*. It then forwards this information to every registered OverView plug-in. Entity creation, deletion and migration, as well as error reporting are all immediately forwarded to the OverView plug-ins to provide an accurate picture of the distributed system. These events are typically less frequent than entity updates and communication, which are queued at the IPA and periodically sent in groups to the plug-ins, to limit the load on the plug-ins. Upon connection, the OverView plug-in specifies the frequency at which communication events are sent. The

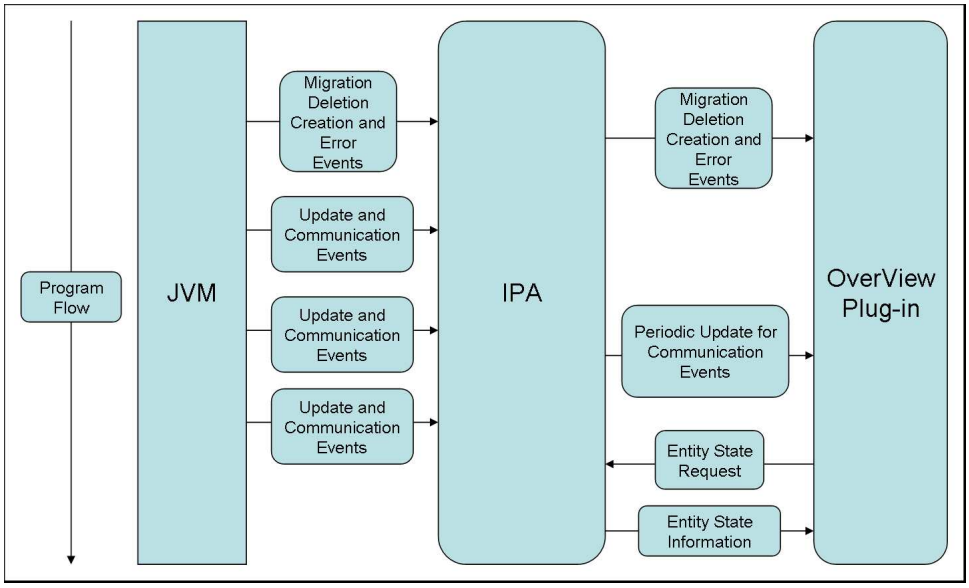

Fig. 5. Program flow for a JVM with an asynchronous IPA

OverView plug-in sends requests for the state of an entity as needed, retrieving the information from the IPA. This approach is very unintrusive as it does not block the JVM and only retrieves information about individual entities, rather than requiring every instance of an update event to be reported to the plug-in. Optionally, an OverView plug-in can request all update events to be forwarded to its data collector, so that the update events are stored when historic data is created.

The IPA and data collector are implemented using the SALSA programming language [15], and communicate using asynchronous message passing via SALSA's remote message sending protocol (RMSP). This allows for IPAs and data collectors to easily send events between themselves, providing synchronization only when it is required. SALSA programs are precompiled into Java code, and profiling statements consist of invoking static methods on the IPAs generated code.

When the byte-code is instrumented, the developer can specify the IPA to operate either synchronously or asynchronously. To maintain a consistent global state, creation, deletion, update, migration and error events must operate synchronously (shown in Figure 4), blocking until the plug-in has received the event until that JVM can continue executing the program. Most debuggers based on the JPDA operate in this manner; however it can severely impact the operation of the distributed program. To fix this problem, the IPAs can operate asynchronously (shown in Figure 5), sending the event and allowing the JVM to continue operating. This might result in the global state becoming inconsistent temporarily, if some creation, deletion, migration or error events are received out of order. These occurrences are rare and sending events asynchronously can result in a significantly less overhead in providing a visualization.

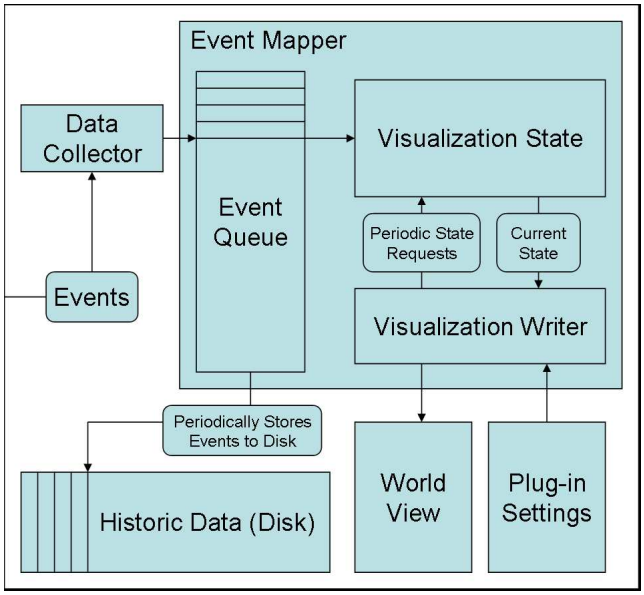

Fig. 6. OverView's Visualization Architecture

5.3 Event Mapper and Historic Data

Events are retrieved by the data collector, which forwards them on to the event queue. These events are passed on to the visualization state, which takes events and determines the current world view. Additionally, the event queue passes groups of events to the historic data and stored to disk. This is done periodically when load on the plug-in is low, so not to impede the speed of the visualization. At the visualization's refresh rate, the visualization writer retrieves the current global state of the distributed program. Then it paints the world view, representing the global state as determined by the plug-in's settings.

5.4 World View

The OverView plug-in can display a view of a distributed program in several ways. Entities are displayed as circles, which can be clicked on to display state data, determined by update events and retrieved from the IPA where that entity is located. Entities are located within other entities, displayed as squares. These act as containers for entities, which may migrate between these containers. Communication between entities is represented by lines, which are color mapped blue, for low amounts of communication, to red, for high amounts of communication. The width of a line determines the average time for communication between the entities: wide, for quick communication, to thin, for slow communication. Entities are colored blue when operating without error, and colored red when an error event has occurred. The entity may be clicked on to determine the error. An example of this world view is shown at Figure 7.

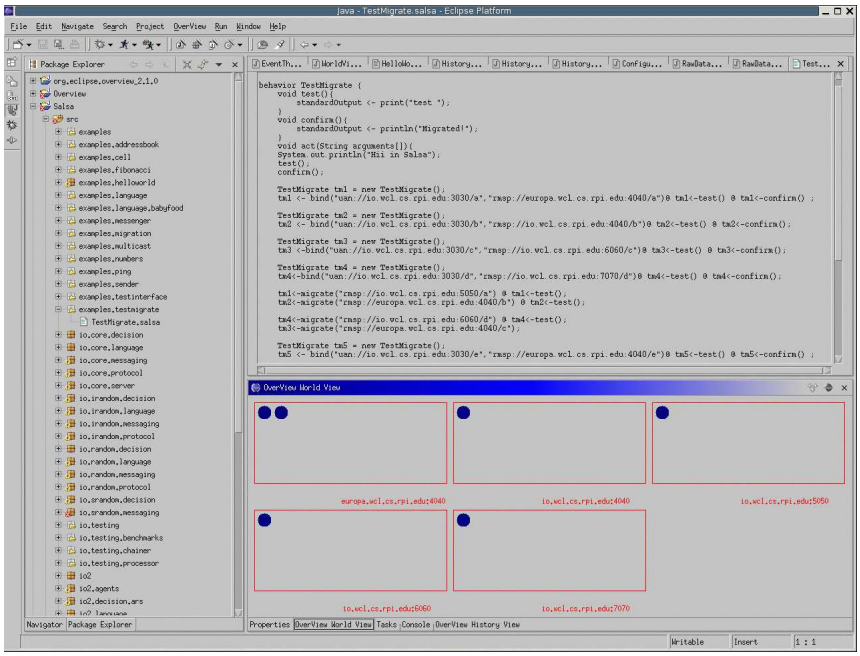

Fig. 7. Snapshot of the OverView World View.

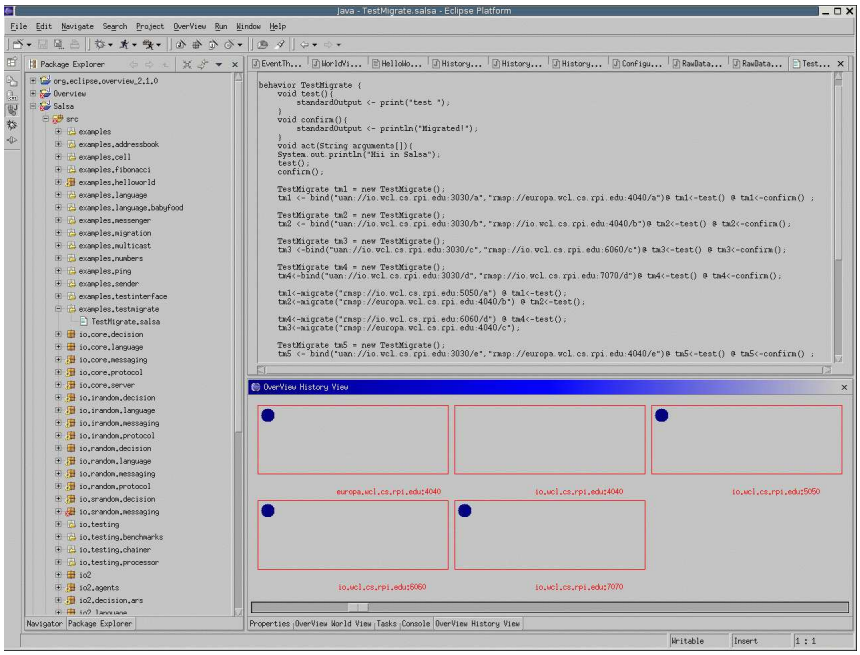

Fig. 8. Snapshot of the OverView History View.

As the OverView plug-in visualizes a distributed program, the developer can start and stop recording events to create historic data. This historic data can be viewed later in the same manner as the online visualization, as shown in Figure 8, with an additional scroll bar that can pause the visualization and move it forward or backward in time. This allows a developer to study what occurred in the distributed program in a slower more fine grained way,

especially if all update events are requested.

5.5 Actor Visualization with OverView

We tested our implementation on distributed programs written using the SALSA programming language [15]. The OverView prototype was able to successfully visualize actor creation, deletion and migration in the different trail runs. It could also capture the state of actors, which was specified as the actor's mailbox size. Figure 7 displays a screen shot of a SALSA program running on five different containers, specifically theaters, displayed as squares. The entities within this system are actors, displayed as circles.

Figure 8 shows a screen shot of the same SALSA program, visualized off of the historic data gained from the online visualization. The scroll bar at the bottom allows the developer to pause or move the visualization forward or backward in time. As the history view progresses, more actors are created and migrated resulting in the same view as in Figure 7.

6 Discussion

Dynamic visualization of distributed systems presents many challenges including:

- the ability to visualize *high-level concurrency abstractions* from events produced by lower-level objects received from distributed virtual machines.
- the need for *non-intrusiveness*, that is, any instrumentation inserted into distributed system components for visualization purposes should not significantly alter the behavior of the distributed system as a whole.
- the need to *scale* up to a large number of interconnected nodes but at the same time provide a unified view of the distributed system.
- the *synchronization* of the visualization tool and the distributed system, so that dynamic reconfigurations in the system performed as a result of load balancing or fault-tolerance policies get reflected appropriately and in a timely fashion at the visualization layer.

In this paper, we have described our preliminary work on OverView, an Eclipse extension providing a generic framework for online visualization of dynamically reconfigurable Java-based distributed systems. The key to genericity has been to provide an entity specification language enabling developers to map high-level abstractions into lower-level events. As a result of this generic language approach, visualization of distributed systems can be performed at a high-level and can scale up since only events which are relevant to the visualization purposes need to be instrumented.

We have tested our framework with distributed programs implemented in SALSA, a language based on the actor model that produces Java code. We have successfully visualized critical dynamic reconfiguration aspects of SALSA

programs such as actor creation and migration. Work remains to be done in developing and testing specifications for visualizing more fine-grained operations such as message passing. We also need to create entity specifications for additional concurrency models to test our framework's genericity. Future work includes evaluating in an objective and measurable manner the effectiveness, performance and scalability of the proposed visualization framework.

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 [14] and different notions of distributed transactions [5]. The ability to control and manipulate the components of these systems at run-time is also an important and desirable feature; the architecture described here presents a highly extensible first-step towards that eventual goal.

7 Acknowledgements

We would like to acknowledge Vivek Sarkar, Wim DePauw and Kaoutar El-Maghraoui for their help and comments on this research. This work was also supported in part by two IBM Eclipse Innovation awards.

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. In *15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 23–23, April 2001.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, pages 140–155. Springer-Verlag, Berlin Germany, 1998.

[4] M. P. Consens, M. Z. Hasan, and A. O. Mendelzon. Visualizing and querying distributed event traces with hy+. In W. Litwin and T. Risch, editors, *Applications of Databases, First International Conference, ADB-94, Vadstena, Sweden, June 21-23, 1994, Proceedings*, volume 819 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 1994.

[5] 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.

[6] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 219–228, May 2000.

[7] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. *Lecture Notes in Computer Science*, 1628:135–160, 1999.

[8] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

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

[10] S. P. Reiss. Jive: Visualizing java in action, personal communication. In *ICSE*, May 2003.

[11] 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.

[12] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe 2001)*, March 2001.

[13] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions of Computer Systems*, 6:2:157–196, May 1988.

[14] 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.

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

[16] 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 Programming, Systems, Languages and Applications*, pages 271–283, 1998.

[17] P. Wojciechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.