Form: A Framework for Creating Views of Program Executions

Tim Souder, Spiros Mancoridis, and Maher Salah
Drexel University
Department of Mathematics and Computer Science
3141 Chestnut Street, Philadelphia, PA 19104, USA
{gtsouder|smancori|gmsalah}@mcs.drexel.edu

Abstract

Form is a framework used to construct tools for analyzing the runtime behavior of standalone and distributed software systems. The architecture of Form is based on the event broadcast and pipe and filter styles. In the implementation of this architecture, execution profiles may be generated from standalone or distributed systems. The profile data is subsequently broadcast by Form to one or more views. Each view is a tool used to support program understanding or other software development activities.

In this paper we describe the Form architecture and implementation, as well as a tool that was built using Form. This tool profiles Java-based distributed systems and generates UML sequence diagrams to describe their execution. We also present a case study that shows how this tool was used to extract sequence diagrams from a three-tiered EJB-based distributed application.

1. Introduction

Much of the research efforts in the area of program understanding have concentrated on extracting high-level models of software systems from information found in their source code. Source code analysis tools for a variety of programming languages (e.g., C, C++, Java, COBOL) provide researchers with infrastructure on top of which they can develop tools for automatic modularization, design extraction, metrics, and so on.

Source code analysis, however, does not provide enough information to understand a program completely because there are components and relations that only exist during its runtime. For example, the *Factory* object-oriented (OO) design pattern [8] is used to "manufacture" objects and make these objects accessible to client objects through an abstract interface at runtime. A static analysis tool would only reveal part of the complete design. Specifically, it would reveal the relationship between the client class and the Factory class but not between the client and the objects that were created by the Factory at runtime.

Program understanding, therefore, is greatly enhanced if views produced by static analysis are augmented by views that expose the dynamic (i.e. runtime) aspects of a system. Hence, there has been a considerable interest in extracting models of program execution (i.e. dynamic models) from the run-time data produced by profiling tools. This interest has produced a need for infrastructure, analogous to that for source code analysis, to support research in dynamic program understanding. Below are several requirements that this infrastructure must meet:

1. Interesting systems are complex and so the models of their activity are correspondingly complex. Thus, the framework must be able to handle large volumes of data (e.g.thousands of object creation and method invocation events).

2. Since the system is executing in real-time, its corresponding model may also need to be adjusted in real-time. Performance meters are a good example of tools whose model is adjusted in real-time.

3. Perhaps only a subset of the system activity is interesting to the software engineer. Therefore, the framework should have some mechanism to filter the volumes of data passing through it.

4. There may be a need to perform multiple analyses on the same runtime data in parallel, thus requiring the framework to have a provision for attaching multiple views to a single data stream.

5. The modeled system may be distributed and, hence, may consist of interacting components that reside on several, possibly heterogeneous, computers. To model the activity of such systems, the framework must support the coordination of data streams from multiple computers (or virtual machines) into a single logical model of the overall system activity.

6. As many modern systems consist of components that are implemented in different programming languages, the framework must support profilers for a variety of languages.

In this paper we describe the Form framework, which was designed to provide solutions to all six of the aforementioned requirements. Our objective was not to develop a dynamic analysis tool, but a reusable and extensible framework that researchers can use to develop sophisticated tools that analyze runtime data from standalone and distributed applications. In addition to describing the Form framework, we describe a tool that uses Form which we developed. This tool profiles Java-based distributed systems and generates UML sequence diagrams to describe their execution.

The structure of the rest of this paper is as follows. Next, we provide an overview of related work in the Background section. The Architecture section covers Form's architectural design. The Implementation section describes our implementation of the Form architecture. The Sequence Diagram Tool section demonstrates one view created using Form that creates sequence diagrams of distributed Java programs. We use a case study to show how this tool was used to extract the sequence diagrams from a three-tiered EJB-based distributed application. In the final section, we conclude the paper and provide some information about the future direction of this work.

2. Background

There are four major areas related to this work. The first two, dynamic analysis and profiling, are related to the Form framework itself. The other two, Enterprise Java Beans and sequence diagrams, are related to the view that will be described in the fifth section of this paper.

2.1. Dynamic Analysis

Dynamic analysis is used to describe the runtime activity of a system. This activity may be the method calls in a computer program, the flow of packets in a network, the movement of cars in traffic, *et cetera*.

Dynamic analysis has been used to describe the runtime activity of computer programs. As we describe next, the data produced by the dynamic analysis process has been used in a variety of ways.

Some early work on dynamic analysis focused on the performance testing and debugging of parallel systems. For example, in Choi and Stone [4], the source code of parallel systems was instrumented to assist in their debugging. Krishnan and Kale extended this idea [10]. Specifically, by analyzing profiling information that was gathered from a parallel program, they optimized the program's parallelization and achieved as high as a two-fold speed improvement in the program's time of execution.

A common use for dynamic analysis is in the area of software testing. Programs are first instrumented. Then, a set of test cases is executed against the program. Finally, the instrumentation returns the segments of the program that were exercised by the test cases [9].

Another area of research that employs dynamic analysis is program decomposition [1]. This decomposition identifies related functions and computations, and can relate those computations to given sets of inputs and outputs. This is useful when the source code is severely obfuscated and needs to be reengineered into readable code.

A more exotic use of dynamic analysis is in the area of computer immunology. As described in Forrest, *et al.*, profiles of the programs' interaction with the operating system are gathered [6]. These profiles are then used to determine the typical behavior of the program. Major deviations from this behavior may indicate an attempted security breach.

As more work was performed in dynamic analysis, it became apparent that a generic framework for dynamic analysis tools would be helpful to the software development community. Bruegge, *et al.* designed the BEE++ system as a generic framework for dynamic analyzers [2]. Similar to Form, this C++-based framework forwards runtime data from multiple profilers to multiple views. Form differentiates itself from BEE++ in several areas. First, it includes multiple levels of filters, its filters can be regular expression based (rather than event type-based), it supports multiple controllers, and it is implemented as a pure-Java program (i.e., it is accessible on many heterogeneous platforms). Second, the BEE++/C++ analyzer requires the source code to be instrumented before analysis, while Form does not require any modifications to the source code. Third, Form is designed to be an open architecture available to the software engineering community, while the BEE++ system does not appear to be readily available. Forth, Form provides a simple API for view development (as we describe in the forth section of this paper).

Dynamic analysis differs from static analysis in that it requires an active system to model. That is, dynamic analysis is performed on a running system, while, static analysis is performed on system artifacts (e.g. source code).

In the static analysis field, the Ciao system, created by Chen, *et al.* provides a framework for performing static analysis of source code [3]. This fundamental technology has facilitated the creation of many static program analysis tools, such as our own Bunch tool [11]. Our Form framework aspires to fill a similar role in the dynamic analysis field to the role played by Ciao in the static analysis field.

Our discussion will now continue with descriptions of profiling, in general, and Java's JVMPI (a profiler inter-

face), in particular, as they relate to the Form framework.

2.2. Profiling

Profiling is at the heart of dynamic analysis. Dynamic analysis depends on creating an image of the running system which can then be analyzed as described in the previous section. To profile a system, its activity (such as function calls) is recorded. Many dynamic analysis systems use instrumentation to create this profile. This instrumention is performed by inserting instructions into the program's source code (as done by BEE++) or directly into the program's binary code (as done by Miller, *et al.*'s Paradyn [13]).

The Form Java profiler uses the Java Virtual Machine Profiler Interface (JVMPI) to provide its profiling information. The JVMPI is a C/C++ interface to the Java virtual machine [19]. To use it, one registers interest in a set of system actions (such as entering a method). When the event occurs, a callback is made to the profiler library that references the event's identifier. The Form profiler has handlers for most of the JVMPI events. These handlers generate XML code that encapsulates information about the event (e.g. the name of the method that was called) and pass it to the Form controller as is described later in the Form Implementation section.

Now that we have discussed dynamic analysis and profiling, which are the basis of the Form framework, we continue the discussion with descriptions of Enterprise Java Beans and sequence diagrams, which are the basis of the view we developed using the Form framework.

2.3. Enterprise Java Beans

Enterprise Java Beans (EJB) is a framework for server-side Java components [14]. The EJB framework comprises EJB components, EJB containers, and EJB application servers. Components execute within a container. The container is responsible for registering components, providing a remote interface to components, coordinating distributed transactions, and object life cycle management. An application server hosts one or more containers. The server provides resource management services such as database and network connection pooling, security, distributed transactions, and persistence.

A client does not interact directly with an EJB bean. Instead, it interacts with two wrapper interfaces provided by the EJB container as shown in *Figure 1*. The first wrapper is the `EJBHome` interface, which clients use to create and destroy instances of a bean and to identify the bean to the Java Naming and Directory Interface. The second wrapper interface is the `EJBObject` interface, which clients use to invoke bean methods.

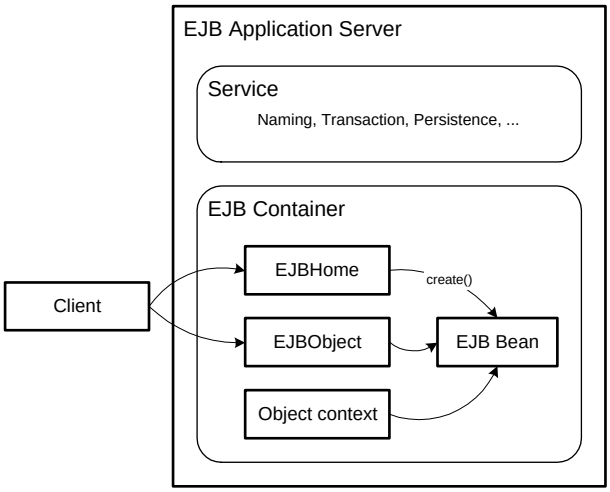

Figure 1. Enterprise Java Beans Architecture

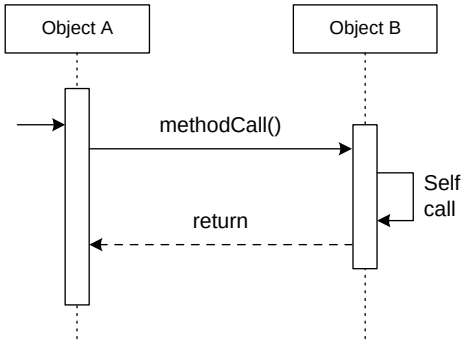

Figure 2. Event Sequence Diagram

2.4. Sequence Diagrams

Object interaction diagrams are views or models that describe how a set of objects collaborate at runtime. UML defines two types of interaction diagrams: sequence diagrams and collaboration diagrams.

Figure 2 shows a simple example of a sequence diagram. In this diagram, *Object A* executes the `methodCall()` method on *Object B*. *B* then calls a method within itself (the *self call*). *B* returns control to *A* (the *return* edge) when it has finished operation.

In the next section, we describe the architecture of the Form Framework.

3. Form Architecture

The Form architecture is an open architecture for passing, filtering, and broadcasting dynamic data as a set of events. The current implementation of the Form architecture, which is described in the next section, uses events to describe the activity of Java programs.

The Form architecture is a combination of the *pipe and filter* and *event broadcast* architectural styles [16]. As shown in *Figure 3*, the profilers are the sources of events and the views are the destinations of events. That is, events originate at the profilers $p_1 \ldots p_4$ and are then forwarded to controllers c_1 and c_2. The controllers filter the events and broadcast them to the interested views $v_1 \ldots v_4$.

If a view needs an unfiltered event stream (such as p_1 to v_1), it can connect directly to the profiler. A typical reason for directly connecting a view to a profiler is to create an event database for later analysis.

If a controller wants to receive a filtered event stream, it can connect to another controller (this is not shown in the figure). A typical reason for connecting controllers is to share the filtering load across several computers.

An event is a message that encapsulates information about a single action of the system being modeled. It contains several attributes:

1. A string that describes the type of the event (e.g. *MethodEnter*, *BeginThread*),
2. A set of key-value pairs that describe each event's parameters,
3. A number that identifies the profiler that sent the event,
4. The time the event occurred, and
5. A serial number that orders events within the context of its profiler (i.e. event 145 occurred before event 146).

When an event is received, the controller identifies the set of views that have registered interest in the event. For each of these views, the controller passes the event through a set of second order filters specific to the view. The second order filters are regular expressions that are tested against the values of the event's parameters. These filters include or exclude events before forwarding them to the view.

4. Form Implementation

The Form system is an open system that implements the Form architecture described in the previous section. This system can be partitioned into four primary components: the RMI interface (object dictionary), profilers (runtime data providers), the controller (event switching fabric), and the views (consumers of events). This section describes

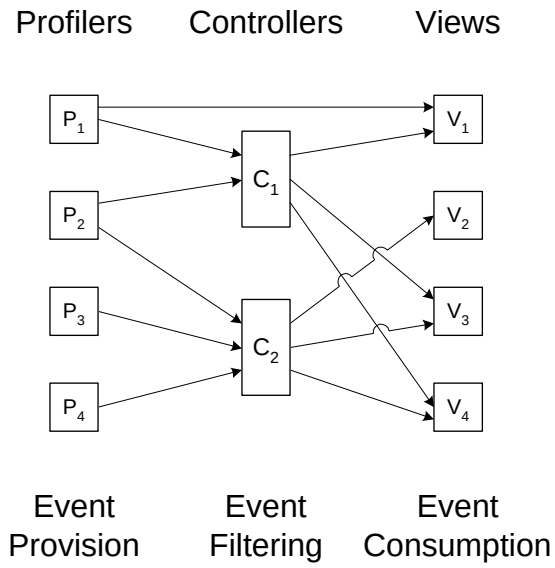

Figure 3. Form Architecture

these four components, followed by how a software engineer can extend the system to create novel views.

4.1. System Components

The Form system was designed to be a distributed, Java-based system. To this end, each element of the system may execute on a separate Java Virtual Machine. Java RMI enables the components of the system to communicate.

4.1.1. RMI Interface

Java Remote Method Invocation (RMI) is a remote procedure call interface for Java objects [20]. Through RMI, objects running in one Java Virtual Machine (JVM) can invoke methods on objects that exist on local or remote JVM's.

Objects register themselves by name in the RMI Registry, which is a centralized directory that enables local and remote objects to locate each other. For security reasons, Sun Microsystems chose to prevent objects on one system from registering in another system's registry. Since Form is designed to run across many systems, the RMI Registry was reimplemented to provide a centralized directory of the objects in the Form system. The directory we developed is called the *Form RMI Registry*.

The Form RMI Registry stores object references for all of the Form objects that communicate across RMI. Permitting all hosts to access the registry would be a security risk, because any computer that has access to the RMI registry would be able to participate in the Form system. If the system is manipulating sensitive data, this access should be re-

stricted. To address this need for security, Form provides several layers of protection:

1. The Form RMI Registry provides host-based access protection. That is, it has a set of access control lists to govern the set of systems (by hostname and IP address) that are permitted access to the Form objects.

2. Java's RMI supports an open socket architecture which can be used to add link-level security features (such as RMI over SSL) [18].

3. Since the profiler may be running on a system that is remote to the Form controller and views, the profiler architecture is designed to support an open security model. In this model, messages passed between the profiler and controller may be protected using a secure transport such as MIT's Kerberos [12].

To summarize, the Form RMI Registry is the central repository for the Form objects. It features a simple security model to protect it from being accessed by unauthorized systems.

4.1.2. Profiler

The profiler is the source of Form data. It provides a stream of XML data that describes the activity of the system it is profiling. We have defined a set of XML tags used to profile program activity. These tags are described on the Form web site [17].

Since a distributed system will have multiple streams of data (one for each element in the distributed system), each profiler is assigned a unique identifier. With this identifier, the multiple streams of data may be distinguished from one another. In this manner, the activity of the distributed systems can be modeled.

The current version of the Form system has a profiler for Java programs. The system is designed so that it is easy to implement additional profilers to support other programming languages (e.g. C/C++) and other sources of dynamic data (e.g. network traffic).

A profiler implementation must generate a stream of XML data that is recognizable by the controllers and views. This XML stream is sent to the controller across a network socket.

4.1.3. Controller

The Form Controller handles event filtering and broadcasting. Events are received from a set of profilers. These events then pass through two sets of filters. The first order filters map events to interested views. The second order filters are more complex. They are regular expressions with a syntax similar to the Unix `egrep` command [5].

There are two subclasses of second order filters, *include* and *exclude* filters. If the include filter is satisfied, the event is sent. If the exclude filter is satisfied (and an the include filter is not present or is unsatisfied), the event is not sent.

For example, perhaps a software engineer is only interested in an event called *MethodEnter*, where one of the parameters of the event is *name* (the name of the class and method entered separated by a period). The engineer is interested in the methods of a class called `A`. To get only these events, the engineer would specify two filters. The first filter is an *exclude filter* that excludes all $MethodEnter$ events in the tuple $(MethodEnter, name, \text{".*"})$. The second filter is an *include filter* that includes all $MethodEnter$ events whose value of the $name$ parameter begins with "`A.`" is specified by tuple: $(MethodEnter, name, \text{"A\..*"})$.

4.1.4. Views

The views accept data from the controller and use this data to model the activity of the profiled system. A view may be connected to one or more controllers. In addition to receiving filtered events through the controller, the view can connect directly to the profiler.

Implementing a view is a fairly simple process. To implement a view, one must implement the `FormViewClient` interface. This interface specifies three functions that are used by the `FormView` component to register the view in the Form system. These methods specify the name of the view and small and large pictures representing the view (to be used in a future pictorial browser of available views). After the view is created, it registers interest in a set of events with the controller (to get filtered events) or directly with the profiler (to get unfiltered events). The view loops through calls to `getEvent()` in the `FormView` object which blocks and returns the next available event. When the event is received, the time of the event is normalized using a simple distributed time algorithm similar to those described in the systems literature [15] [21]. With the distributed time algorithm, as multiple JVMs may be generating events, the relative times of the events may be compared. For further details, see the `FormViewTest` class in the Form source distribution (available at http://serg.mcs.drexel.edu/form/download).

Now that we have described the Form Architecture and its implementation, we will continue with a description of the *Sequence Diagram* (SD) tool. SD is used to create sequence diagrams of distributed systems using Form.

5. The Sequence Diagram Tool as a Form View

The components of distributed systems typically do not share memory. Rather, the system's state is distributed across multiple components that are executing on one or

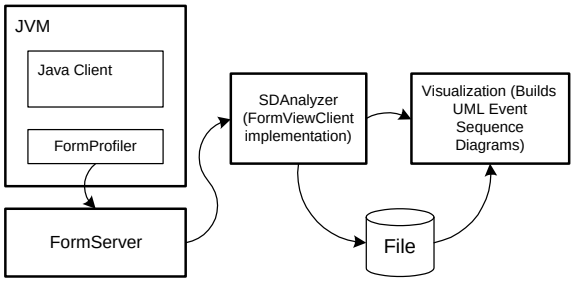

Figure 4. Form/SD Integration

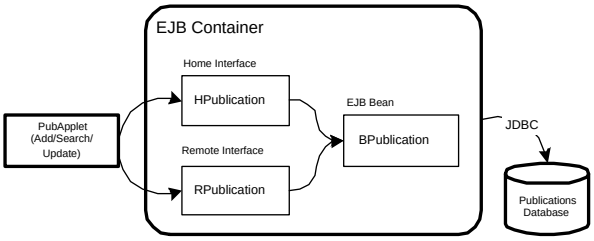

Figure 6. TRS Architecture

more computers. Monitoring a complete thread of execution that crosses process boundaries is difficult without the help of distributed monitoring or profiling tools. Such tools can capture the runtime state from multiple parallel portions of the system.

We used the Form System to construct a tool called SD. SD profiles Java-based distributed systems and generates sequence diagrams of the execution of threads of a profiled system. The sequence diagrams are represented using the Unified Modeling Language (UML) notation for sequence diagrams [7].

As shown in *Figure 4*, SD consists of three subsystems. The first subsystem is the Form system. The second is the SD analyzer, which is an implementation of the `FormViewClient` interface. The third subsystem is the sequence diagram component itself.

SD accepts profiling data from the Form controller. The `SDAnalyzer` (a subclass of `FormView`) gathers traces from multiple JVMs and can output them to either a file or directly to the visualization component. After the trace is complete, the visualization component builds and displays the sequence diagrams.

A sample of the SD tool's interface is shown in *Figure 5*. At the top-left of the window, the play, pause, and stop buttons allow the user to control the recording of profiling data from the distributed system. The Trace panel allows the user to select a feature of the program and view it. When the user clicks the *View* button, a window is opened to display the sequence diagram for the feature.

To profile a Java-based distributed system, each participating JVM must be profiled. As described in the previous section, the profiler connects to the Form controller, which is responsible for filtering and ordering events, and broadcasting them to interested views, such as SD.

SD identifies full execution traces of a profiled system. In this context, a full execution trace is defined as a sequence of method calls. The sequence begins with the start of a new thread. It ends with the thread's termination, or triggered by a callback from the JVM such as `EventListener.actionPerformed()` as a result of pressing a button. Many full traces, as we will later demonstrate, represent scenarios or use cases of an application's operational profile, or a slice of the system's execution.

Now that we have described the SD system, we next show how the activity of a distributed EJB system can be modeled using SD.

5.1. Modeling the Behavior of a Technical Report System Using SD

In this case study, SD is used to recover the sequence diagrams of a technical report system, called *TRS*. TRS has three features: add, search, and update a publication. TRS is a three-tiered system. The frontend of TRS is a Java client, the middle tier comprises a single EJB session bean. The backend is an Oracle database.

Figure 6 depicts the TRS architecture. The home interface `HPublication` extends `EJBHome` interface, and specifies one method, `create()` that returns a reference to the remote interface `RPublicaiton`. The `RPublication` interface extends `EJBObject` interface, and specifies the access points to the `BPublication` bean's business methods: `AddPublication()`, `UpdatePublication()`, `SearchByTitle()`, `SearchByAuthor()`, and `GetLastSQLException()`. Database access is performed through JDBC (Java Database Connection).

Using SD we identified sixteen full traces. Three traces for the *add* feature, three traces for the *search* feature, one trace for the *update* feature, and one trace for system *startup* and *login*. The remaining eight traces are for Java library callbacks. For example, two of the Java library-related traces are `ListSelectionModel.valueChanged()` (for the table that shows search results) and `ChangeListener.stateChanged()` (for the table in the main window). The maintainer can annotate and store interesting traces and sequence diagrams, so that they become part of the system documentation.

Figure 7 shows the sequence diagram for the *Search Publication* TRS feature. The feature requires four calls that

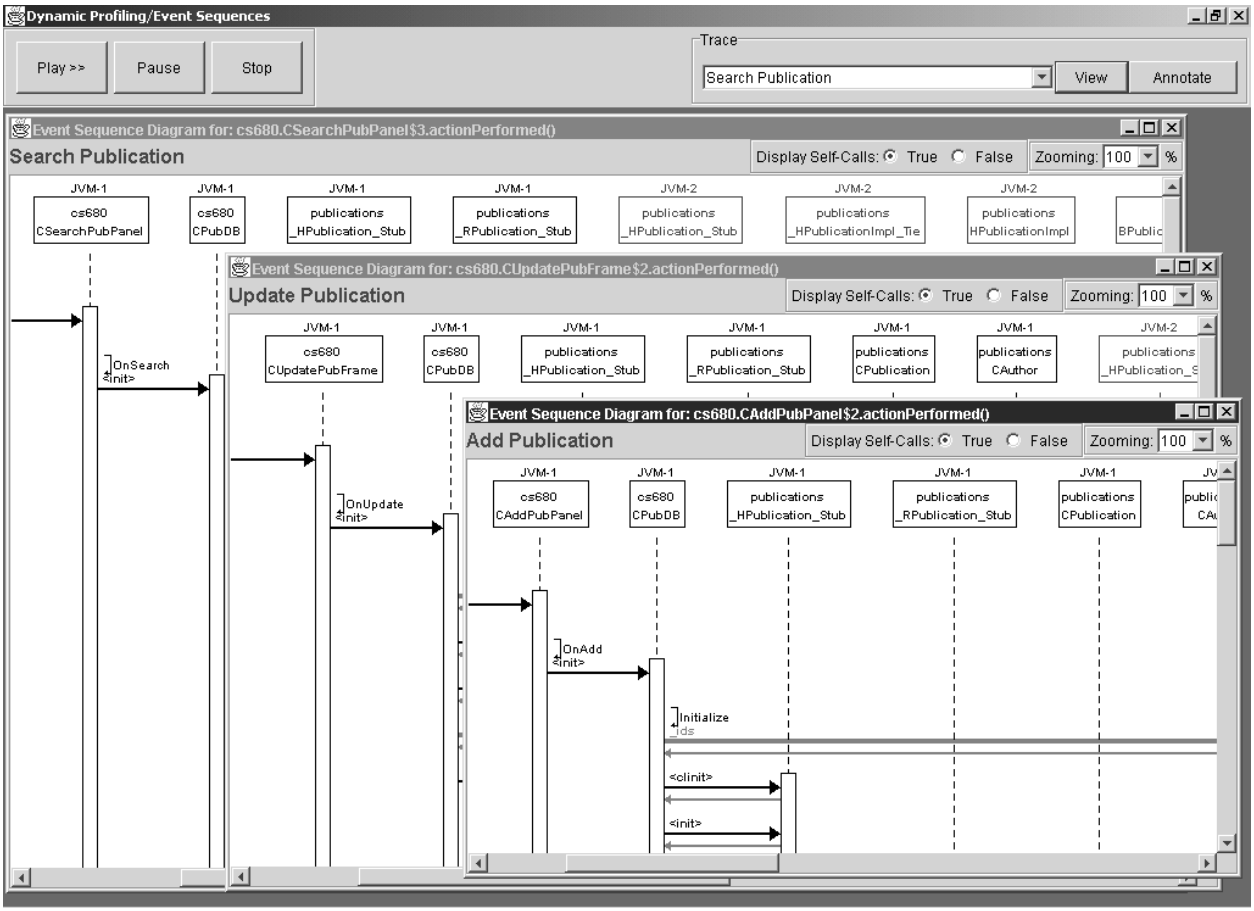

Figure 5. SD Main Window

cross between JVMs. A series of boxes along the top of the diagram identify the classes involved in the trace. Above each trace is the number of the JVM where the class is running. Each JVM participating in the system is assigned a unique number beginning at 1. Calls that cross JVM boundaries (e.g. `_all_interfaces`, `_idc`, and `_private`) are displayed in red (a thick, dark grey line in the figure). A light grey edge links the two classes in places where control returns to the calling function.

5.2. Integration with Form

SD is implemented using the Form framework. The `SDAnalyzer` class is an implementation of the `FormViewClient` interface as described in the previous section. Thus, it receives filtered profiling events from the Form controller and uses them to build a model of the program's operation. It requests thread- and method-related events from the Form controller.

Modeling the operation of a system is based on the path of execution of each thread in the system. When the analyzer receives a begin thread event from the Form controller, it creates a new trace for the thread. The trace is marked complete when the thread terminates. The analyzer associates each incoming event with its trace based on the thread generating the event. For each identified trace, a sequence diagram is generated.

SD operates in two modes, online mode and offline mode. In the online mode, SD builds the sequence diagrams while the application is being executed. In the offline mode, SD reads traces from a file and generates sequence diagrams for each full trace. Figure 4 illustrates the architecture for the SD tool.

5.3. How SD Assists in Program Understanding

SD provides three useful features for software maintainers. First, it helps in design recovery by producing execution traces, represented as sequence diagrams, from the system run-time profile. Since each execution trace cap-

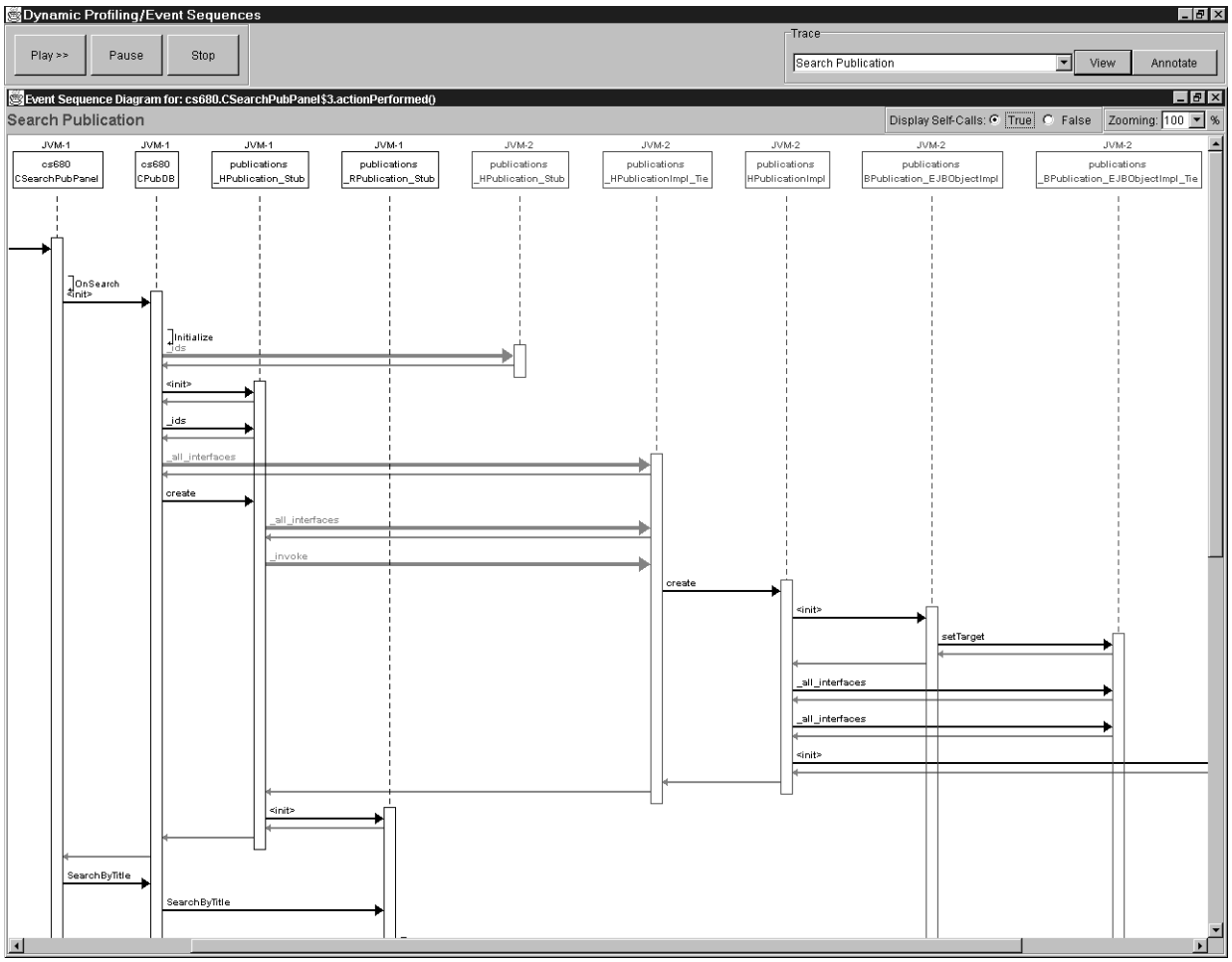

Figure 7. Search Publication Event Sequence Diagram

tures the dynamic behavior of a set of collaborating objects, representing the trace as a sequence diagram provides the maintainer with a good starting point to recover the actual use case or operational profile of an application. Second, each diagram represents a precise dynamic slice of program execution. Such dynamic slices can be used by maintainers, as well as testers, to determine the objects and methods that participate in a system feature. This information is useful when designing testing cases for a given use case. Third, it improves the visibility of a distributed application by providing a window in which system run-time behavior can be observed as sequence diagrams.

6. Conclusions and Future Work

In this paper we reviewed the Form architecture, its implementation, and a tool that uses the implementation to create sequence diagrams of distributed Java programs. Our thesis is that a distributed system's operation cannot be described in a single view or model. The Form architecture is thus designed to meet this need by facilitating the development of multiple views on data gathered from many sources.

The Form architecture is a unique three-tiered broadcast model that supports two levels of event filtering. It is a unique technology for advancing research in dynamic program understanding.

The framework itself is implemented as a distributed Java system with a Java profiler. SD uses the framework to create models of distributed Java programs.

Future work will concentrate on four areas. Initially, we will focus on performance issues. Secondly, we will provide more capabilities for the profiler (such as to allow it to start and stop tracing a program at the profiler level). Third, we will create additional profilers to provide runtime data for other languages (such as C/C++ and Fortran) and other non-prorgammatic dynamic systems. While this work

on the infrastructure continues, we will also work on novel views to help in program understanding.

The first release of our tool is available from Form's web page at http://serg.mcs.drexel.edu/form. We hope that other researchers will use the Form system to create their own views.

Acknowledgments

This research is sponsored by two grants from the National Science Foundation (NSF): a CAREER Award under grant CCR-9733569 and an Instrumentation Award under grant CISE-9986105. Additional support was provided by grants from the research laboratories of AT&T and Sun Microsystems.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, the U.S. government, AT&T, or Sun Microsystems.

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Engineering Conference*, Toulouse, France, 1999. ACM.

[2] B. Bruegge, T. Gottshalk, and B. Luo. A framework for dynamic program analyzers. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, 1993. ACM.

[3] Y.-F. Chen, G. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of the 1995 International Conference on Software Maintenance*, pages 66–75. IEEE, 1995.

[4] J.-D. Choi and J. M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In *Conference Proceedings on ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, 1991. ACM.

[5] D. Dougherty. *sed & awk*. O'Reilly & Associates, Inc., Sebastophol, CA, 1992.

[6] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[7] M. Fowler and K. Scott. *UML Distilled: a brief guide to the standard object modeling language*. Addison-Wesley, 2nd edition, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. In *ACM Transactions on Software Engineering and Methodology*, number 4, pages 370–397. ACM, Oct. 1997.

[10] S. Krishnan and L. V. Kale. Automating parallel runtime optimizations using post-mortem analysis. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 221–228, Philadelphia, PA, 1996. ACM.

[11] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings Sixth International Workshop on Program Comprehension*, Ischia, Italy, 1998. IEEE Computer Society Press. http://serg.mcs.drexel.edu/bunch.

[12] Massachusetts Institute of Technology, Cambridge, MA. *Kerberos: The Network Authentication Protocol*, 2000. http://web.mit.edu/kerberos/www.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[14] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 2nd edition, 2000.

[15] M. Raynal and M. Signhal. Logical time: A way to capture causality in distributed systems, Jan. 1995. http://www.irisa.fr/bibli/publi/pi/1995/900/900.html.

[16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[17] T. Souder and S. Mancoridis. *Form Developer Documentation: Java Profiler Events*. Drexel University Software Engineering Research Group, Philadelphia, PA, 2000. http://serg.mcs.drexel.edu/form/doc/dev/JavaProfEvents.html.

[18] Sun Microsystems, Palo Alto, CA. *The Scoop on RMI and SSL*, 1999. http://java.sun.com/j2se/1.3/docs/guide/rmi/SSLInfo.html.

[19] Sun Microsystems. Java virtual machine profiler interface. http://java.sun.com/products/j2se/1.3/docs/guide/jvmpi, Feb. 2000.

[20] Sun Microsystems, Palo Alto, CA. *Java® 2 Remote Method Invocation (RMI)*, 2000. http://java.sun.com/products/jdk/rmi.

[21] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 1994.