Toward an environment for comprehending distributed systems

Maher Salah and Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
msalah@cs.drexel.edu, Spiros.Mancoridis@drexel.edu

Abstract

Many modern software systems are often large, distributed, written in more than one programming language, and developed using pre-built components. This paper presents the results of the first phase of a project to develop an environment that supports the comprehension of distributed systems.

The environment has a layered architecture consisting of three subsystems: data gathering, data repository, and modeling/visualization. The first phase of the project focuses on the design and implementation of the data gathering and data repository subsystems. The key requirements of the environment are to support: (a) static and dynamic analysis, (b) multiple languages, (c) distributed systems, and (d) component-based models.

1 Motivation

Modern software systems are often large, distributed, written in more than one programming language, and developed using distributed component-based frameworks such as CORBA/CCM [27, 28], EJB [36], and DCOM/COM+ [23]. The comprehension of distributed systems introduces new challenges. First, the source code for many of the components of a distributed system is unavailable and may be written using more than one programming language [37]. Second, the components that constitute a distributed system do not execute on a single computer but, rather, on a network of computers. Third, although the need for human-generated code decreases when pre-built components such as EJB and COM+ are used, the overall complexity of the system increases. Specifically, many components are used partially, thus making the unused functionality part of the system [37, 38]. Also, the many layers of software infrastructure (*e.g.,* shared facilities and libraries) of the component framework reduce the visibility of the software's internals. The unavailability of the source code and the multi-language issues of modern software systems have been characterized as practice patterns for architecture reconstruction (*i.e.,* binary component, the mixed-language patterns [32]).

To address the above issues, we are working on a project to develop an environment for comprehending distributed systems. The architecture of the environment comprises three subsystems: data gathering, data repository, and modeling/visualization (see Figure 1). The first phase of the project, whose results are the topic of this paper, focuses on the data gathering and the data repository subsystems. The second phase focuses on the modeling, visualization and empirical evaluation of the environment's usability in an industrial setting. Some of the key technical problems we have addressed are:

1. The construction of distributed profilers that collect run-time data from components that execute across a network.
2. The design and implementation of a single data repository for storing static and dynamic data from multi-language applications.
3. The management of the enormous volume of data produced during dynamic analysis.
4. The decoupling of the environment's subsystems to enhance the extensibility and ease the integration of new tools.

The organization of the rest of the paper is as follows: Section 2 describes related research, Section 3 describes the architecture of the environment, Sections 4 and 5 emphasize the data gathering and repository subsystems, and Section 6 uses a small example to illustrate how the environment is used to analyze a distributed systems.

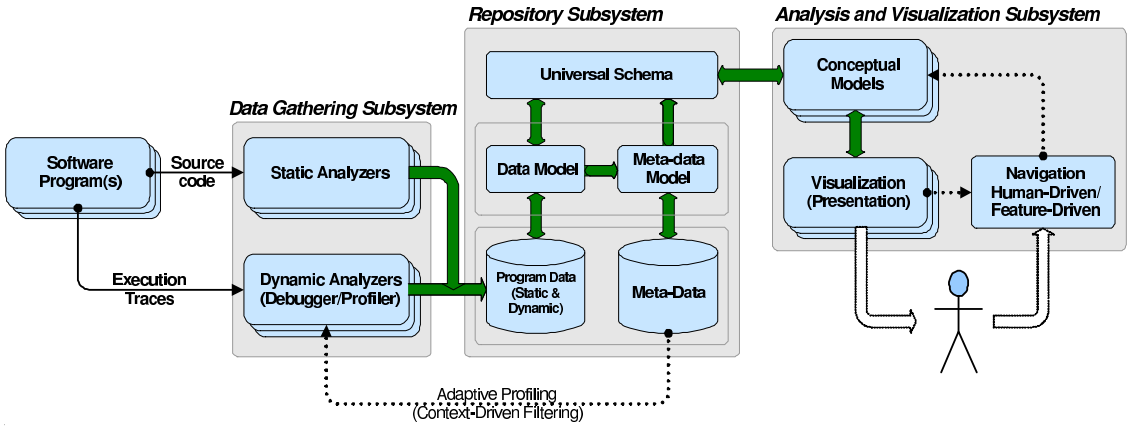

Figure 1. Architecture of the software comprehension environment

2 Background

This work is related to three research areas. Namely dynamic analysis, the analysis of multi-language applications, and the analysis of distributed systems.

2.1 Dynamic Analysis

Dynamic analysis is used to study the behavior of software systems. There are three methods for collecting run-time data. The first method is source code instrumentation. Bruegge *et al.* designed the BEE++ system [1] as a framework for monitoring systems written in C/C++. In this system, event generation is achieved by instrumenting the program source code. Another system that uses instrumentation is SCED [18]. This system uses run-time data to create dynamic models of object-oriented programs, which are visualized as state diagrams or state charts. SCED only collects data from stand-alone applications, while BEE++ can also collect data from distributed applications.

The second method is pre-complied code instrumentation. This method is widely used to instrument Java bytecode [20]. The third run-time event collection method is based on debugging and profiling. Modern development frameworks provide interfaces to facilitate the collection of run-time data. Examples of such interfaces include JVMDI and JVMPI for Java [33, 34], CLR Profiling Services for Microsoft .NET [25], and COM+ instrumentation services for COM+ applications [24]. Debuggers have been used to emulate profiling interfaces by automatically inserting breakpoints and manipulating the stack frame of the executing program. Drexel University's GDBProfiler [5] uses the GNU debugger interface to profile C programs.

The key difference between the three data collection methods is that the source code is required for the source code instrumentation method, but is not required for the debugging/profiling and bytecode instrumentation methods.

2.2 Analysis of Multi-language Applications

Traditional software comprehension tools assume that the software is written in a single programming language and that the source code is available for analysis. For many modern systems these assumptions are not valid, because the source code of the components may not be available, and is often written in multiple languages. For example, in web-based information systems, the source code typically comprises a mixture of scripting languages such as Active Server Pages (ASP), Php, Perl or Java Server Pages (JSP). In addition, the use of pre-built components, such as COM and EJB, is common. Multi-language development is common in today's development environments as is indicated by a recent survey by Evans Data Corporation [10] showing that 60% of the enterprise developers surveyed use multiple languages.

These issues have been recognized in recent research [4, 14]. The multi-language aspect of web applications has been studied by Hassan and Holt [14]. Their approach uses source and binary extractors to perform static analysis. The extracted data is consolidated into a common fact base.

Acacia [4] supports the static analysis of several languages including C/C++, Java [16], Html [3] and ksh. In Acacia, a multi-language application could be analyzed piecemeal for each language. However, a different repository for each language is created, and, hence, queries that cross language boundaries are not possible.

There are two essential requirements for multi-language analysis: (a) the data collection tools for different languages, and (b) the data model (schema) of the common repository. The second requirement has received little attention compared to the first one. The importance of data modeling has been recognized [6, 8, 29] as a key requirement for exchanging data between different tools. Koschke *et al.* [17] introduced a set of requirements for the intermediate

representation of data for reverse engineering tools. Kullbach *et al.* [19] developed an integrated conceptual model for multiple languages (MVS/JCL, COBOL, PL/I and others) and a graph-based query language. Our approach differs from Kullbach's approach in three ways. First, our data repository consists of both static and dynamic data about the programs being analyzed. Second, the data repository is extensible so that new languages can be added easily. Third, there is no need for translation between the data gathering tools and the repository.

2.3 Analysis of Distributed Systems

The maintenance of distributed systems is inherently difficult because of issues such as: asynchronous communication, network delays and crashes, lack of a common physical clock, and the lack of a global system state [9, 30]. In a distributed environment, system components do not share memory, and system state is distributed across multiple components running on a variety of computers and operating system platforms. Monitoring a complete thread of execution that crosses process boundaries is difficult or impossible without the help of distributed monitoring or distributed profiling tools.

The issues pertaining to the physical distribution of components have been studied previously [1, 21]. The BEE++ system [1] uses source code instrumentation to monitor the execution of distributed systems written in C/C++, and dispatches run-time events to various distributed software comprehension tools. The X-Ray system [21] relies on the analysis of C/C++ source code to recover the architecture of distributed systems. The client-server relationships are identified using clustering techniques and clues from the source code. For many modern distributed systems, the reliance on source code may not be sufficient for an accurate and complete analysis. First, the naming and registration services, which are commonly deployed by many frameworks (*e.g.*, CORBA, DCOM and EJB) make the identification of interrelationships between remote components impossible by just studying the source code. This is because the location of the name server and the network ports are determined at run-time. Second, the source code of many components may not be available for analysis or instrumentation, which limits the portions of the system that can be analyzed.

Our approach differs from BEE++ and X-Ray in three ways. First, dynamic data is the primary data source and static data is gathered only if the source code is available. Source code is still important, because dynamic traces may exercise only portions of the system under study. In addition, the use of both static and dynamic data enhances the quality of analysis [7, 35]. Second, no instrumentation and recompilation is needed. Our distributed profiler does not use the source code, it operates on byte-code for Java and executable objects (EXE or DLL) for C/C++ and Visual Basic (VB). Third, our environment is not limited to a single programming language or framework.

3 Software Comprehension Environment

Figure 1 illustrates the architecture of our software comprehension environment. The main subsystems are:

1. **Data gathering**. This subsystem defines the interfaces and data formats of the data collection tools (*i.e.,* static analyzers and dynamic profilers). At the core of this subsystem is the distributed profiler, which consists of three basic components. The first component is a local profiler, which is attached to each running process. The second component is an event ordering service, which is essential since the components of a distributed system do not share a common clock. The third component is a data collection manager that receives data from the local profilers. This effort is a continuation of our previous work on the Form architecture [31].
2. **Repository**. This subsystem defines the data and meta-data models, as well as the data manipulation and query language. The data repository stores the program entities, relationships, and run-time events that are gathered from the subject system. The universal schema is a composite model of both data and meta-data models along with a query and data manipulation language. The repository is manipulated using standard SQL and is accessed using either SQL or our own SMQL (described later).
3. **Analysis and visualization**. This subsystem is responsible for the creation and visualization of software views. The subsystem has three components: conceptual modeling, visualization, and navigation. The conceptual modeling component defines the set of filters and analyzers for deriving software models. The models are a function of data and context. The data part of the model is the result of a query, and the context is the meaning associated with the data. The visualization component is responsible for the presentation of the models, and the navigation component is responsible for exploration of the views.

The next two sections describe the data gathering and data repository subsystems in detail.

4 The Data Gathering Subsystem

The architecture of the data gathering subsystem is influenced by four design goals. The first goal is to support multiple languages. The second goal is to gather static

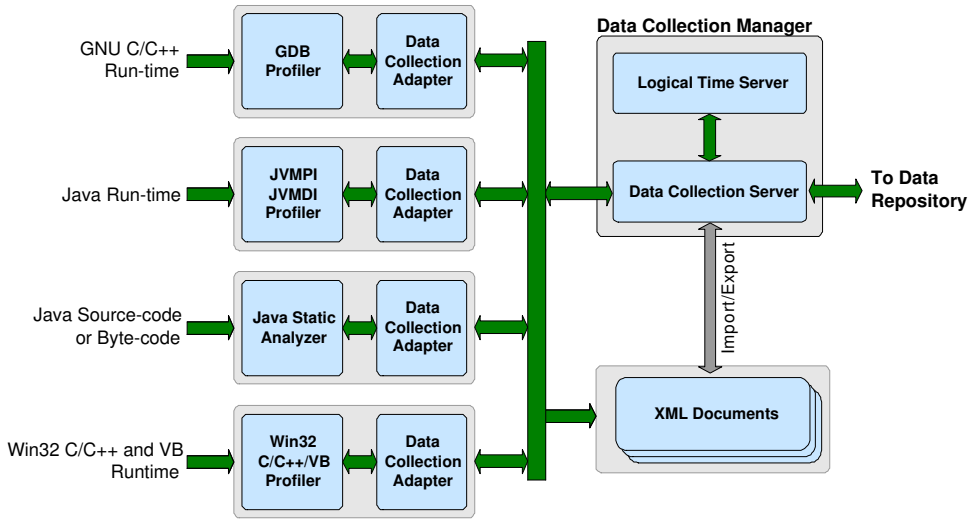

Figure 2. Data Gathering Subsystem

and dynamic data, because both kinds of data are needed for a comprehensive analysis of the structure and behavior of distributed systems. The third goal is to decouple the data collection component from the rest of the environment's subsystems. The tight-coupling between data collection tools and the rest of the tools is a common drawback of many popular tools [13] such as Acacia [2]. The final goal is to support commonly used scripting languages and component-based frameworks.

Figure 2 illustrates the architecture of the data gathering subsystem. The language-dependent data extractors are responsible for extracting program artifacts from the software's source code or execution. The data collection adapters encode source code facts and run-time events and route them to the data collection manager. The adapters provide a common interface for transporting the collected data as serialized Java objects or as XML messages. The time server provides a shared logical clock to enable the correct ordering of the run-time events when profiling distributed systems. The data collection manager routes data to the data repository and provides access to the language meta-data models.

A key requirement for analyzing distributed systems is the identification of remote interactions between components. The profiler must be able to distinguish between local interactions and remote ones. Remote interactions occur when a component invokes special functions that result in a request/reply messaging protocol between the two distributed components. For example, in Java, the special functions include the `read()` and `write()` methods of the `java.net.Socket*Stream` class. In C/C++, the special functions include the `recv()` and `send()` functions of the `winsock.dll` library.

To support the analysis of distributed systems, the data collection and repository subsystems model the concept of communication **endpoint** entities, and the **connects** relationship between endpoints.

Figure 3 shows portions of a UML event sequence diagram of two distributed components of a distributed systems. In the diagram the solid diamonds highlight *endpoints*, the gray thick arrows show the *connects* relationships between the components. A method from the `userClass` class invokes the `write()` method of the `SocketOutputStream` class to send a message to the `userModule` module. The `userModule` uses the `recv()` function (in `ws2_32.dll` library) to read the message. The `userModule` uses the `send()` function to send a reply to `userClass`.

An endpoint is defined as a tuple with four elements: network address of the local host, local network port, network address of the remote host, and the remote network port. The *connects* relation between two endpoints A and B is valid if all of the following conditions are satisfied:

$$
\begin{aligned}
EP_{localHost}(A) &= EP_{remoteHost}(B) \\
EP_{localPort}(A) &= EP_{remotePort}(B) \\
EP_{Time}(B) &- EP_{Time}(A) < Threshold \\
EP_{Time}(B) &> EP_{Time}(A)
\end{aligned}
$$

Where,

$EP(A)$	:	An endpoint from application **A**
$EP(B)$	:	An endpoint from application **B**
$EP_{localHost}(A)$	:	The domain name or IP address of the machine hosting application **A**
$EP_{localPort}(A)$	:	The TCP/IP port number for application **A**

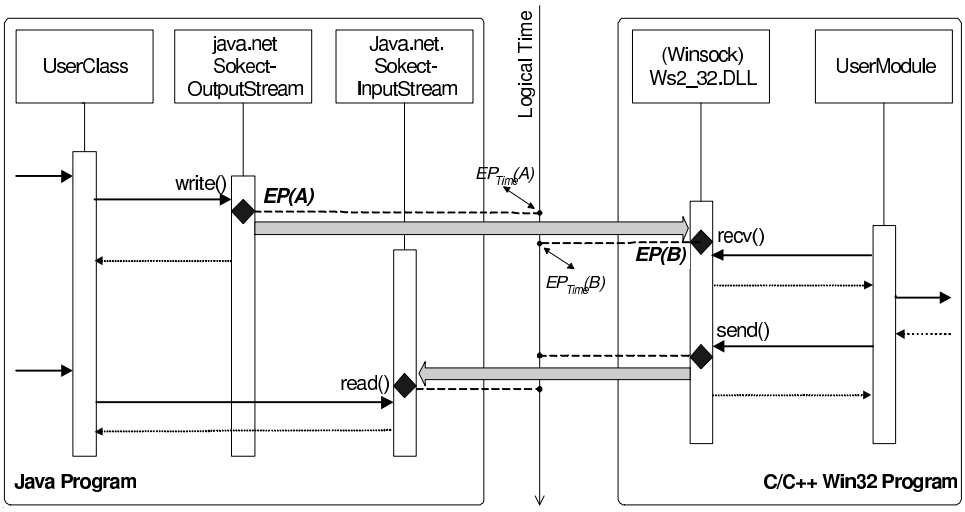

Figure 3. Communication Endpoints

EP_{Time}	:	The logical time-stamp when the endpoint was detected
$Threshold$	:	The acceptable time difference between the detection of the two endpoints, ideally its value is one logical-time tick

The above conditions ensure that the correct pair of endpoints are associated to establish the remote interaction between two distributed components. The *endpoint* entity is identified by the distributed profiler using network interceptors, while the *connects* relation is determined by analyzing the event trace.

Currently, the environment supports the following data extractors:

- **wdbg**: A Win32 C/C++/VB profiler that is implemented using the Win32 debugging API.
- **jvprof**: A Java profiler that is implemented using JVMD and JVMPI interfaces.
- **jsa**: A Java static analyzer that is implemented using the BCEL [11] package.
- **gdbprofiler**: A GNU C/C++ Profiler [5] that is implemented using the GNU debugger library.

Modeling the run-time interactions between more than one application requires profilers that can intercept network calls (*e.g.,* socket or SocketStream calls) to determine the communication endpoints between the participating distributed components.

We implemented *network-call interceptors* in the **wdbg** (for Win32-based applications) and **jvprof** (for Java applications) local profilers. Running either or both profilers in conjunction with the data collection manager forms a distributed profiler for collecting run-time data from distributed applications written in Java and Win32 C/C++/VB.

The Java-based local profiler (**jvprof**) uses the JVMDI and JVMPI interfaces [33]. The profiler watches for method call events from instances of the `java.net.SocketInputStream` and `SocketOutputStream` classes. The endpoint parameters are determined by reading the field values of the `SocketImpl` class. The instance of `SocketImpl`, which is declared in the `SocketInputStream` and `SocketOutputStream` classes, holds the values of the remote and local network addresses and the port numbers. Note that Java RMI and EJB use these classes to implement remote invocations.

The C/C++ local profiler (**wdbg**) uses the Windows debugging API [22], which is limited and cumbersome to use compared to JVMDI. **wdbg** watches for `Winsock` library function calls. The `Winsock` library is the Windows implementation of the TCP/IP protocol API (known as the Socket Interface). The profiler acts as a debugger process for Win32 binary executables. First, it reads the debugging symbol table and all of the imported dynamic link libraries (DLL), then it sets breakpoints on all identified functions. Inclusion and exclusion filters on source code files, DLL modules, and functions are supported to limit the amount of run-time data that is gathered. By default, if the `winsock` library is used, the profiler sets breakpoints on the `connect`, `accept`, and all `read/write` functions. The socket value, remote host, and remote port number are read from the stack frame of the watched functions. The local port value is read by intercepting the `getsockname` function. The local port cannot be read from all winsock-based applications, since the call to `getsockname` is optional, and not every winsock-based program uses it. To overcome

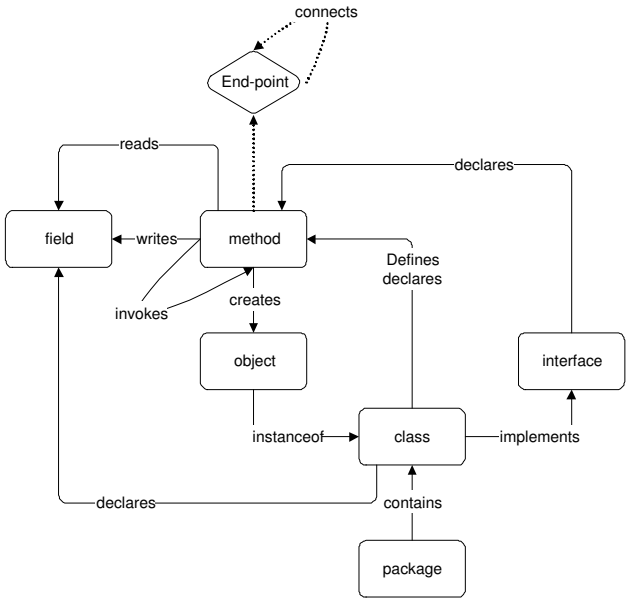

Figure 4. Java Language data model

this problem, we minimally instrument the code by replacing calls to the `connect` and `accept` functions by two wrapper functions `dbg_connect` and `dbg_accept`, respectively. The wrapper functions call the original functions, then call the `getsockname` function to read the local port.

To avoid instrumentation and recompilation, we are considering an alternative approach to implementing the network interceptor for our Win32 binary-based profiler. The new approach will implement the necessary functionality in a separate DLL, which the profiler loads at run-time into the profiled-program process memory. The profiler then detours selected calls to this library.

5 The Data Repository

The schema of the data repository comprises three models. The first is the Language Definition model, which defines the entities and relations for each programming language. The second is the Program Data model, which stores the data gathered (entities, relations and run-time events) about the programs under study. A run-time event is defined as a relation with a run-time context. The run-time context includes the time when the event occurred, the process, and thread that generated the event. The third model is defined as a set of database views to simplify querying the repository. The SMQL query language (described later in this section) uses these database views.

The Language Definition model is a graph, where nodes define programming language entities, and edges define relations between the entities. Formally,

$$Model(General) = Graph(E_g, R_g)$$

Where,

E_g	is the set of all entity types supported by the repository.
R_g	is the set of all relation types supported by the repository.

Each element in R_g is a tuple of three elements ($relationType$, $sourceEntity$, $destinationEntity$). A specific programming language model (*e.g.,* Java) is defined as the projection of the general model over the set of entities and relations supported by the language. For example, the Java language can be defined as:

$$Model(Java) = Graph(E_{java}, R_{java})$$

Where,

E_{java}	$\subset$	E_g	is the set of all entity types supported by the Java language
R_{java}	$\subset$	R_g	is the set of all relation types supported by the Java language, such that $source$ and $destination$ entities $\in E_{java}$

A new language model can be instantiated from the general model or from an existing language model, For example, a C++ model can be defined from the Java model as:

$$Model(C++) = Graph(E_{c++}, R_{c++})$$

where,

$$E_{c++} = E_{java} \cup \{\texttt{template,struct,typedef,function}\} - \{\texttt{interface,package}\}$$

$$R_{c++} \subset R_g \text{ such that } source \text{ and } destination \text{ entities} \in E_{c++}$$

The graph in Figure 4 illustrates the Java language data model. Note that the edges to and from the endpoint entity (diamond shape) are represented in dotted lines to emphasize that these edges are determined by analyzing the event trace and are not directly extracted from the source code.

5.1 Query Language: SMQL

Even though the repository can be queried using SQL, designing queries for comprehending software systems using SQL is cumbersome. In addition many of the queries that are of interest to an engineer, for example queries that involve the transitive closure of a relation, are not supported directly by SQL. To overcome this limitation, we developed a higher-level query language called SMQL (Software Modeling Query Language). SMQL is a set-based language that facilitates the definition of queries about entities, relations and events by translating the SMQL code into SQL query statements. SMQL provides a built-in `closure` function, which computes forward and reverse reachability, as well as binary operators such as union, intersection and join. SMQL is similar to `grok` [15] for manipulating binary relational algebra expressions. Unlike `grok`, SMQL can be extended to support additional operations that are implemented in Java.

The purpose of SMQL is query processing and data retrieval. The creation of software views is through extensions, to SMQL written in Java, using the `filter` function. SMQL maintains its data in memory as Java objects that are passed to the user-defined filters for analysis.

To introduce SMQL, we use an inheritance hierarchy view. The following SMQL code gets the inheritance hierarchy rooted at the `myClass` class using the `closure` function.

```
// Example 1: Inheritance view
// Declare an entity set
Entity S
  {
  // Include entities of type class and
  // interface from myPackage.myClass
  type = {class, interface} ;
  include(name) = {"myPackage.myClass"};
  }
// Use closure to get all subclasses
I=closure(null, S, {subclass}) ;
// Produce the ouput in Dot-Format
ouput(I,
    "java:serg.sc.filter.DotOutputFilter");
```

The `closure` function returns a set of relations. The semantics of the `closure` function is as follows:

$$\textbf{closure}(source, destination, relationTypes)$$

where, *source* is the source entity-set, *destination* is the destination entity-set, and *relationTypes* is the set of relation types over which the closure is computed. If the *source* is an empty set (`null`), `closure` returns the reverse reachability from any entity to all entities in the destination set. If the *destination* set is empty, then `closure` returns the forward reachability from the source-entity set to any entity. If both are provided, `closure` returns a set of relations that contains the paths between the source and destination sets. Note that $source \cup destination \neq null$.

In the `output` statement, the second argument instructs SMQL to forward the (`I`) relation to the Java-based output filter `DotOutputFilter`, which converts the relation (`I`) into a dot graph, which is saved as a file that can be viewed using dotty tool [12].

The next section describes an example of a distributed system, where the server is a `Java` program running on Linux and the client is a `C/C++` program running on Windows 2000.

6 Example: Distributed Interaction Diagram

Figure 5 shows the deployment architecture for profiling a distributed system. The client (**Tear.exe**) is a simple web browser that takes an HTTP URL and connects to the web server and receives the content of the HTTP request. This example is one of the samples distributed with Microsoft MSDN Library. Tear.exe is written in C/C++ and runs on Windows 2000. The server is the **Jetty** Web Server [26]. Jetty is written in Java and runs on Linux and JDK 1.4.2. **jvprof**, the Java profiler is used to collect run-time events from the web server (Jetty), and **wdbg**, the windows profiler is used to collect event from the client (Tear.exe). The data collection manager and the repository run on the same Windows 2000 machine as the server. The solid arrows in the diagram show the data flow (run-time events), and the dotted arrow show the virtual communication link between Jetty and Tear programs. In this example, we show how with one simple query we can obtain an object interaction diagram from a multi-language distributed application.

The following SMQL code defines the event set `E`, which contains selected events from both applications (Tear and Jetty). For the purpose of this example, the set of event-types is {`method-entry`, `method-exit`, `endpoint`, `thread-start`, `thread-end`}. Also, the example illustrates the use of the `Filter` function, which uses the `DistCGraph` class to extract the interaction diagram from the run-time data collected from the distributed system. It

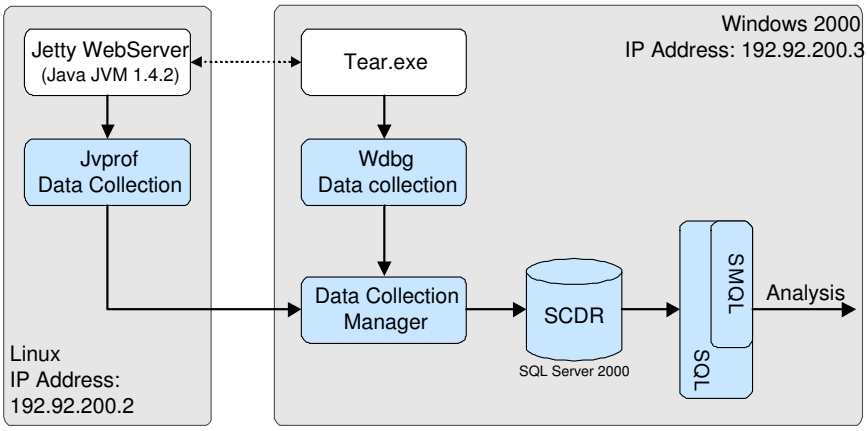

Figure 5. Deployment architecture of `Jetty-Tear` example

is worth noting that without a common data repository for `Java` and `C++`, the following query would not be possible.

```
// Example 2: Distributed interaction
// diagram
// Defining the set of events
Event E
  {
  type = {method-entry,
          method-exit,
          endpoint,
          thread-start,
          thread-end } ;
  // Select events from Jetty Web Server
  // and Tear applications
  include (application) =
         {"Tear", "Jetty"} ;
  }
// Use DistCGraph to Process events
Filter(
   "java:serg.sc.filter.DistCGraph", E) ;
```

The `DistCGraph` class processes the sequence of events from the `Jetty` and `Tear` programs and builds an interaction diagram. In addition, it searches for the communication endpoints, and for each matched pair of endpoints it adds the *connect* relation to show the communication between the client and server. Figure 6 shows the interaction diagram (call graph) of the two programs as well as the communications between them. The rectangles represent classes (for Java) or compiled-modules (for C/C++). The edges represent one or more *invoke* relation between classes or modules. Endpoints are represented as diamonds with a (local-host:port => remote-host:port) label.

Each of the subgraphs can be constructed separately by analyzing the programs individually using tools such as Acacia [4] and Chava [16]. However, the remote connectivity between the programs can only be identified by dynamic analysis using a distributed profiler.

7 Conclusions and Future Work

In this paper we describe an environment for comprehending distributed systems. Specifically we focus on the design and implementation of the data gathering and repository subsystems. The environment addresses the unique challenges for comprehending distributed systems such as the dynamic analysis of distributed systems and the analysis of multi-language applications. Our solution includes a distributed profiler with network interceptors that identify communication endpoints, a common repository for storing data from multiple languages, and the SMQL language for simplifying queries that cross multiple applications and programming language boundaries. Through an example of a distributed system, we demonstrate the ability of the environment to collect data and analyze a multi-language distributed system.

The first phase of the project must still address the performance issues of the distributed profiler, the static analysis of VB programs, and the support for the dynamic analysis of component-based systems developed using component-models such as EJB, COM+, and Microsoft.Net.

The second phase of the project involves the design, construction, and visualization of a set of software views for comprehending distributed and component-based systems. During the second phase, we will perform an empirical evaluation of the environment in an industrial setting with the help of Cigna Corporation, one of the largest insurance companies in the United States.

References

[1] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analysis. In *Conference on Object-*

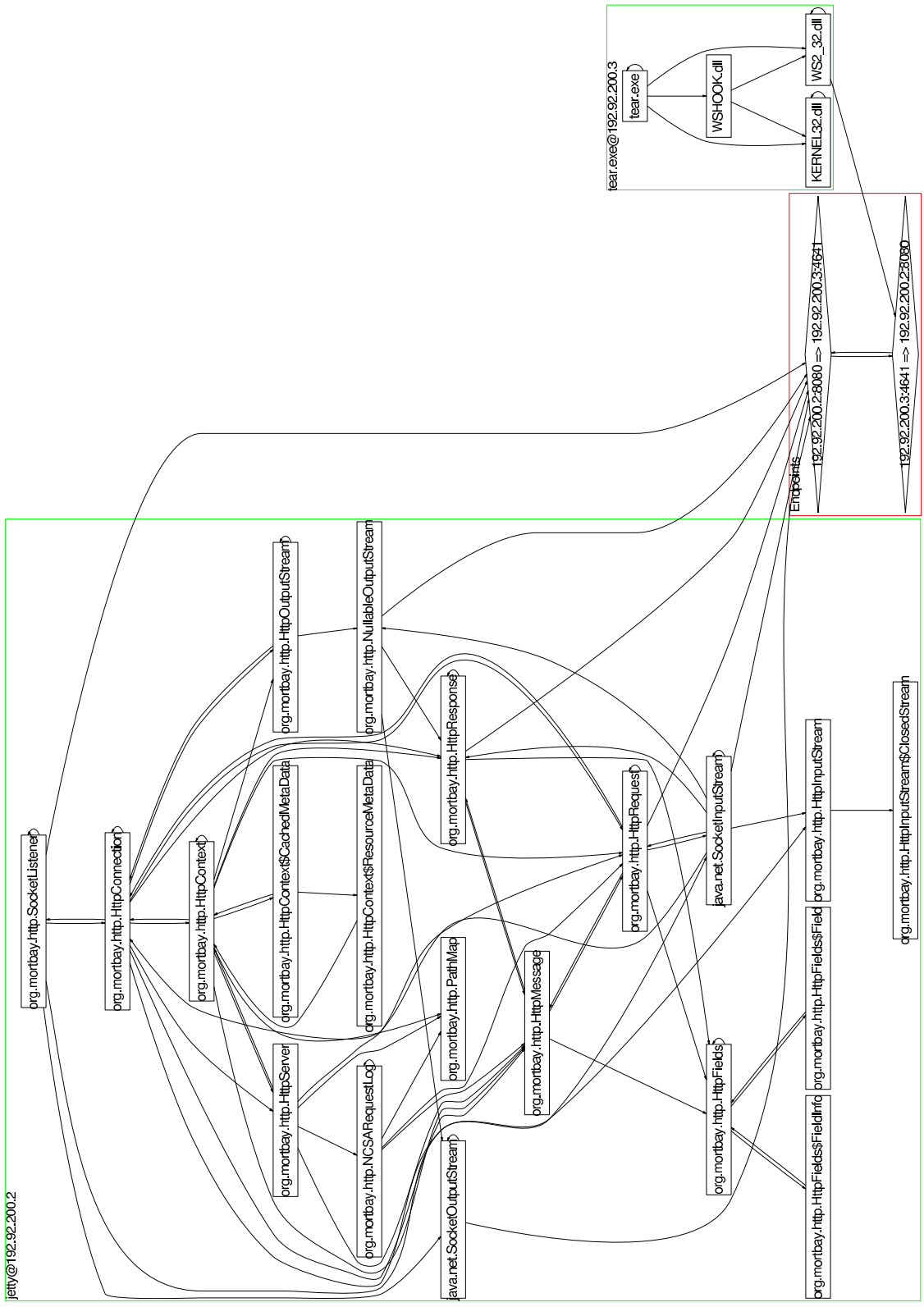

Figure 6. Interaction diagram of `Jetty-Tear` example

Oriented Programming Systems, Languages, and Applications (OOSLA93), Washington, USA, September 1993.

[2] Y.-F. Chen, E. Gansner, and E. Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9), 1998.

[3] Y.-F. Chen and E. Koutsofios. Webciao: A website visualization and tracking system. In *Proceedings of WebNet97*, Toronto, Canada, October 1997.

[4] Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of International Conference on Software Maintenance (ICSM)*, Nice, France, October 1995. IEEE.

[5] C. Dahn and J. Penrose. GDBProfiler for GNU C/C++. http://serg.mcs.drexel.edu/gdbprofiler.

[6] S. Demeyer, S. Tichelaar, and P. Steyaert. Famix 2.0 - the famoos information exchange model. Technical report, University of Berne, August 1999.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, Florence, Italy, November 2001. IEEE.

[8] Electronic Industries Alliance and International Standards Organization (EIA/ISO). *CDIF - Framework for Modeling and Extensibility*, 1994.

[9] P. Enslow. What is a 'distributed' data processing system? *IEEE Computer*, 11(1), January 1978.

[10] Evans Data Corporation. *Enterprise Development Management*, volume 1 edition, 2002.

[11] Free Software Foundation. *Byte Code Engineering Library (BCEL)*. http://jakarta.apache.org/bcel.

[12] E. Gansner, E. Koutsofios, and S. C. North. *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ, February 2002.

[13] M. W. Godfrey and E. H. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *In Proceedings of the 2nd International Symposium on Constructing Software Engineering tools (CoSET-2000)*, Limerick, Ireland, June 2000.

[14] A. E. Hassan and R. C. Holt. Architecture recovery of web applications. In *24th International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA, May 2002.

[15] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report CSRI-345, University of Toronto, March 1996.

[16] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proceedings of Sixth Working Conference on Reverse Engineering (WCRE)*, Atlanta, GA, USA, October 1999. IEEE.

[17] R. Koschke, J.-F. Girard, and M. Wurthner. An intermediate representation for reverse engineering analysis. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE)*, Honolulu, HI, USA, October 1998.

[18] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modeling of object systems. Technical Report A-1996-4, Department of Computer and Information Sciences, University of Tampere, Finland, 1996.

[19] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE)*, Honolulu, HI, USA, October 1998.

[20] H. Lee and B. Zorn. *BIT: Bytecode Instrumenting Tool*. http://www.cs.colorado.edu/ hanlee/BIT.

[21] N. C. Mendonca. *Software Architecture Recovery for Distributed Systems*. PhD thesis, Department of Computing, University of London, November 1999.

[22] Microsoft Corporation. *Microsoft Debugging Tools*. http://www.microsoft.com/whdc/ddk/debugging.

[23] Microsoft Corporation. *DCOM Technical Overview*, 1996.

[24] Microsoft Corporation. *COM+ SDK Documentation: COM+ Instrumentation*, 1999.

[25] Microsoft Corporation. *.NET Framework: Runtime profiling*, 2001.

[26] Mort Bay Consulting. *Jetty Web Server and Servlet Container*. http://jetty.mortbay.org.

[27] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, version 2.2 edition, February 1998.

[28] Object Management Group. *CORBA Component Model*, version 3.0 edition, June 2002.

[29] Object Management Group. *XML Metadata Interchange (XMI)*, version 1.2 edition, January 2002.

[30] M. Singhal and C. Thomas. Distributed computing systems. *IEEE Computer*, 24(8), August 1991.

[31] T. Souder, S. Mancoridis, and M. Salah. Form: A framework for creating views of program executions. In *International Conference on Software Maintenance*, Florence, Italy, November 2001.

[32] C. Stroermer, L. O'Brien, and C. Verhoef. Practice patterns for architecture reconstruction. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, Richmond, VA, USA, October 2002.

[33] Sun Microsystems, Inc. *Java Platform Debugger Architecture*, 1999.

[34] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*, 1999.

[35] T. Systä. *Static And Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Department of Computer and Information Sciences, University of Tampere, Finland, 2000.

[36] A. Thomas. Enterprise javabeans technology: Server component model for the java platform. Technical report, Patricia Seybold Group, December 1998.

[37] M. Vigder. The evolution, maintenance, and management of component-based systems. In G. Heineman and W. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 29, pages 527–539. Addison-Wesley, 2001.

[38] E. J. Weyuker. The trouble with testing components. In G. Heineman and W. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 28, pages 499–512. Addison-Wesley, 2001.