ViVA: A Visualization and Analysis Tool for Distributed Event-Based Systems

Youn Kyu Lee, Jae young Bang, Joshua Garcia, and Nenad Medvidovic
University of Southern California
Los Angeles, California, USA 90089
{younkyul, jaeyounb, joshuaga, neno}@usc.edu

ABSTRACT

Distributed event-based (DEB) systems are characterized by highly-decoupled components that communicate by exchanging messages. This form of communication enables flexible and scalable system composition but also reduces understandability and maintainability due to the indirect manner in which DEB components communicate. To tackle this problem, we present *Visualizer for eVent-based Architectures, ViVA,* a tool that effectively visualizes the large number of messages and dependencies that can be exchanged between components and the order in which the exchange of messages occur. In this paper, we describe the design, implementation, and key features of ViVA. *(Demo video at http://youtu.be/jHVwuR5AYgA)*

1. INTRODUCTION

Distributed event-based (DEB) systems, which are often built using message-oriented middleware (MOMs) platforms, are widely used in many application domains, including enterprise management, large-scale data dissemination, and real-time monitoring. In 2005, the market size for MOM licenses was about one billion USD [2]; by the end of the decade, the market for all middleware licenses was nearly 20 billion USD, with MOM among the fastest growing middleware platform types [1]. Unlike traditional systems where components directly invoke each other, DEB systems use a form of implicit invocation where components communicate by exchanging messages over connectors. This form of communication decouples the DEB systems' constituent components, and in turn facilitates development of highly flexible, resilient, scalable, concurrent, and heterogeneous distributed applications.

While DEB systems enable the desirable features previously mentioned, these systems are also comparatively harder to comprehend because the exchanged messages are obscured by the *ambiguous interfaces* of DEB components [3]. These interfaces do not reveal the specific types of messages that may be sent or received by a component. Thus, determining causality relationships between messages or what parts of a system may be affected by a maintenance task is challenging in a DEB system [7, 4]. This reduces the understandability of the system and hampers development activities such as debugging and refactoring. The reduced understandability may also hamper engineering teams that experience high turnover rates as it is likely to increase the initial learning curve.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICSE '14 May 31 - June 07 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2768-8/14/06 ... $15.00.

We posit that engineers can improve their understanding of the DEB systems on which they are working with the aid of proper visualization. Although some existing techniques are able to identify the types of messages exchanged between components [4], no technique exists that can effectively visualize (1) the large number of messages and dependencies between them that exist in DEB systems and (2) the order in which the exchange of messages occur.

In this paper, we present *Visualizer for eVent-based Architectures, ViVA,* a tool that aids engineers in understanding how messages are exchanged between a DEB system's components. ViVA has four key features. First, ViVA initially depicts the *entire* architecture including the key architectural elements, such as components, connectors, and the relationships among them. Second, ViVA presents the messages that are exchanged between the components, along with the order in which these messages are generated during runtime. ViVA also supports monitoring whether particular messages or a particular order of message exchanges occur. Third, to deal with the overwhelming number of messages and the dependencies between them that DEB systems commonly have, ViVA provides a feature called *filtered visualization* that focuses only on information regarding components and messages in which engineers state an explicit interest. Finally, ViVA can depict both static and dynamic dependencies between components. To this end, we have integrated an existing static analysis tool, to be used in concert with ViVA's native dynamic analysis capability.

The rest of this paper is organized as follows. Section 2 enumerates the techniques for improving DEB system comprehension that ViVA adopts. Section 3 depicts the overall ViVA architecture along with the description of each major component, as well as the details of ViVA's implementation. Section 4 highlights ViVA's key features. Section 5 concludes the paper.

2. OVERVIEW OF ViVA

ViVA relies on the integration of three key techniques: runtime visualization, message-log replay, and combined static and dynamic analysis. We elaborate on each in this section.

Runtime Visualization. Runtime Visualization is a powerful method for understanding a software system as it can transform the raw data generated by a running system into a form that is more intuitive for human comprehension. Runtime visualization can help software engineers in different ways. For instance, engineers can much more readily grasp the system's overall architecture by studying the visualization than by relying on the original source code or the raw runtime data. Moreover, an appropriate visualization can inform engineers interested in outliers by identifying and highlighting them, which could be costly with raw data [5].

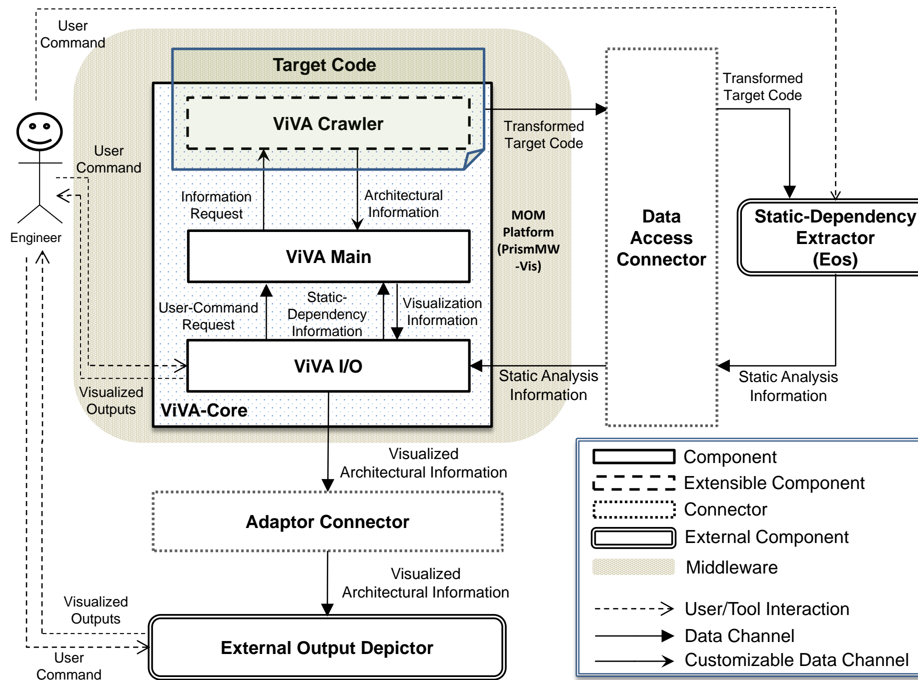

Figure 1: Overview of ViVA's Architecture

ViVA is able to visualize the architecture of a DEB system and illustrate its key architectural elements: components, connectors, their interdependencies, and their exchange of events (i.e., messages). ViVA also implements a visualization of message chronology [5]: it displays messages transferred between components during execution time in the order in which they are generated.

Message-Log Replay. ViVA can also visualize recorded messages after the fact, in the order they were sent. We refer to this as *message-log replay*. ViVA implements message-log replay by recording the message exchange between components and connectors, and by providing the engineer a "remote controller" to replay the message-exchange history forward and backward. Engineers may exploit this feature to identify unexpected message-exchange behavior or to establish that desired behavior is missing. Furthermore, if the execution fails, a replay execution can be used to help track down the failure and its causes.

Combined Static-Dynamic Analysis. Static and dynamic analysis have been shown to be effective software change impact analysis methods [10] that can complement each other. Static analysis techniques can consider all possible control-flow paths, which can result in an over-approximation of the dependencies in a system. In contrast, dynamic analysis techniques can be more precise, but only consider information along paths that have been executed. ViVA adopts both methods and provides the engineers with the intersection of their results. In turn, this has the potential to yield more accurate impact analysis.

3. ViVA ARCHITECTURE

Figure 1 depicts ViVA's architecture. The architecture comprises three major subsystems: *ViVA Core*, *Static-Dependency Extractor*, and *Output Depictor*. *ViVA Core* is responsible for collecting and processing the analysis information to be visualized, and for managing the integration and the interaction with the other two subsystems, both of which have been integrated off-the-shelf. *ViVA Core* relies on a MOM platform that supports recording of messages exchanged between components. In the case of message-log replay, the MOM platform continuously records exchanged messages until it receives a stop request. Afterwards, the MOM platform transfers the recorded messages to *ViVA Core*. In the rest of this section, we describe the details of ViVA's three subsystems and their implementations.

3.1 ViVA Core

ViVA Core comprises three distinct components: *ViVA Main*, *ViVA Crawler*, and *ViVA I/O*.

ViVA Main runs the dynamic analysis computation, sorts and searches the collected application messages, and manipulates the information to be presented to engineers. *ViVA Main*'s dynamic analysis monitors messages at the incoming and outgoing interfaces of a DEB component to determine their inter-dependencies. ViVA's MOM platform records all messages at these interfaces and makes them accessible to *ViVA Main*. The MOM platform adds a unique ID to each message that is sent by a component. *ViVA Main* determines dependencies by checking the IDs of sent and received messages: it identifies the component where each message has been generated and the components that receive that message. *ViVA Main* also sorts recorded messages in temporal order and generates visualization information to be presented via *ViVA I/O* or *Output Depictor*.

ViVA Crawler collects the structural information of the DEB application's target code. This information includes how each component and connector of the application are initialized and composed.

ViVA I/O interacts with the engineer and the two external subsystems. *ViVA I/O*'s GUI visualizes architectural information received from *ViVA Main* and shows ViVA's status. This information includes coordinates, colors, types of architectural elements and messages, as well as their relationships. *ViVA I/O* also forwards the analysis results received from *Static-Dependency Extractor* to *ViVA Main*. In case *Output Depictor* has been connected, *ViVA I/O* forwards the visualization information to the *Output Depictor*.

3.2 Static-Dependency Extractor

Static-Dependency Extractor identifies message-based dependencies using static analysis. These dependencies include the types of messages that each component sends and the corresponding components that receive those message types. Through the use of an explicit data-access connector [9], ViVA is able to integrate with different *Static-Dependency Extractor*s.

3.3 Output Depictor

In addition to its default visualization, ViVA also supports visualization through *Output Depictor*, an external and pluggable component. *Output Depictor* allows engineers to customize notations in their visualization and renders architectural information sent from *ViVA I/O* in a predefined format specified through a metamodel.

3.4 ViVA Implementation

Both *ViVA Core* and the target applications it visualizes are implemented in Java and run on PrismMW-Vis, a specialized version of Prism-MW [8]. Prism-MW is an extensible MOM platform that allows a developer to implement applications using predefined architectural constructs, such as components and connectors. Prism-MW is chosen because of its extensibility and support for different architectural styles, types of connectors, and implementation languages. PrismMW-Vis extends Prism-MW by additionally providing the ability to record information about message exchanges as an array structure and to share them with *ViVA Core*. Prism-MW provides implementations of message interfaces. Messages in Prism-MW follow a predefined format that includes a name and a time stamp, which are used for routing. During runtime, PrismMW-Vis records the information about messages into two separate fixed-size arrays, one for sent and the other for received messages. Since *ViVA Core* runs on PrismMW-Vis, *ViVA Main* can directly access the arrays for the recorded information about messages.

ViVA Crawler must be embedded in the target code so that it can access and analyze the architectural configuration, i.e., the set of associations between components and connectors, of a DEB application. The architectural configuration is explicitly specified using PrismMW-Vis. To embed *ViVA Crawler* in the target code, we add a pre-compiled Java class file that implements *ViVA Crawler* to the target code.

We selected Generic Modeling Environment (GME) [6] as ViVA's *Output Depictor*. GME is a domain-specific modeling tool that supports meta-modeling, provides APIs for model manipulation, visualizes models, and supports facilities for users to interact with the model. We created a new modeling notation for ViVA by specifying a meta-model in GME. This notation includes only the essential information of the architecture visualization. ViVA invokes the model manipulation APIs of GME through the adaptor connector (shown in Figure 1) to present the output of the requested ViVA function from *ViVA I/O*.

For ViVA's *Static-Dependency Extractor*, we chose Eos [4]. Eos is a summary-based iterative data-flow analysis that computes the types of messages in a DEB application, the message types' constituent attributes, and the message-flow dependencies between components. *ViVA Core* interacts with *Static-Dependency Extractor* (i.e., Eos) via a data-access connector: one subsystem writes the relevant information to a file, and the connector enables the other subsystem to access that information without directly coupling the two subsystems.

4. KEY ViVA FEATURES

In this section, we describe ViVA's four key features. These features are intended to help engineers understand complex system structures that have large numbers of components, connectors, and message dependencies. The features also give engineers the ability to track sequences of message exchanges, in order to find errors or simply to improve their understanding of how the system works. Each feature consists of one or more associated operations accessed via the ViVA console as shown in Figure 2.

Visualizing Entire Architecture. When first encountering a DEB system, an engineer can benefit from seeing all of its components, connectors, and message dependencies. To this end, ViVA provides an operation, called `visualize`, that displays the entire architecture of the target system. The `visualize` operation depicts the system's components as light blue rectangles, connectors as gray rounded rectangles, internal connections as black lines, and external connections as blue lines. Internal connections are associations between components and connectors that are deployed on the same host. External connections are associations between components and connectors that are deployed across different hosts.

As part of the `visualize` operation, an engineer can request to see all messages exchanged during a particular run of the application. In response, ViVA displays the messages as scrolling lists inside the rectangles of the sending and receiving components. The shading of the components that did not send or receive any messages during this run changes to dark blue. Figure 2 shows an example of this visualization.

Visualizing Partial Architecture. A target system may have a large number of messages, dependencies, components, and/or connectors. Displaying the system in its entirety may result in an incomprehensible visualization. To deal with this situation, ViVA provides an operation called `filter`. This operation allows engineers to focus on a particular part of the target system. Once an engineer inputs the names or identifying keywords of the desired components and/or connectors into ViVA's console, ViVA responds by depicting only the requested part of the architecture. Analogously to the `visualize` operation, `filter` can also show the messages exchanged among the selected components during a given run.

Tracking Message Exchanges. Filtering for particular components and connectors can help in developing an understanding of a DEB system. Another level of understanding is achieved by considering the messages exchanged between components. For example, engineers may be interested in whether any of the expected messages are missing, whether unexpected messages have occurred, and whether the order of message exchanges is different than expected.

ViVA supports tracking message exchanges and their order via an operation, called `track`, that displays the messages exchanged during a run in chronological order. Engineers can monitor message exchanges in, both, forward and reverse orders. For a message m that has been exchanged between components, `track` (1) changes the color of the component that generated m to green and shows m inside that component, (2) changes the color of the connector through which m has been routed to red, and (3) changes the color of the receiving component to green and shows m inside it. If an engineer tracks the message exchange backwards, ViVA performs the above three steps in reverse order.

As another aid in focusing on specific messages, ViVA provides an operation called `search`. This function goes through the message-exchange history to find whether messages with a particular name have been generated or exchanged during runtime. ViVA highlights the components through which the searched messages have traveled by changing the color of the components to yellow. Engineers may exploit this function to identify missing messages or unintended message exchanges.

Additionally, ViVA provides an operation called `check_flow`,

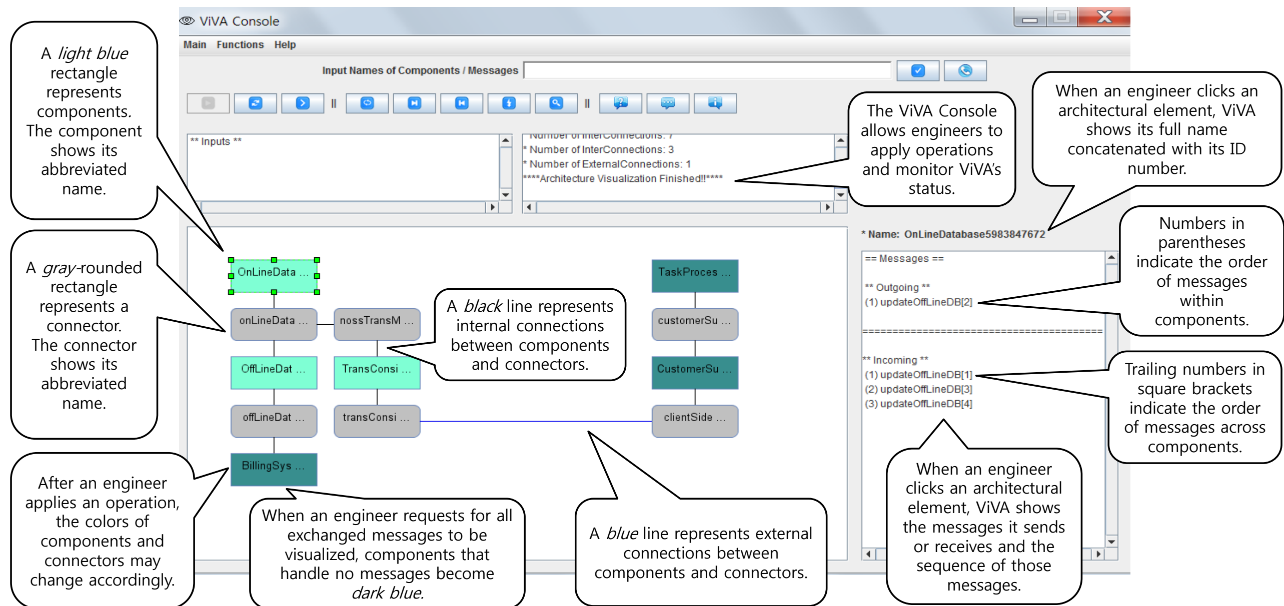

Figure 2: ViVA's User Interface

which compares a message sequence occurring at runtime with a sequence specified by an engineer. ViVA displays any mismatches and highlights the involved components in red.

Visualizing Component Dependencies. Checking dependencies between components helps engineers determine the potential impact of changes to a system. However, the accuracy of the extracted dependencies is a function of the quality of the employed analysis techniques. For example, a static analysis used by ViVA may identify spurious dependencies, and (although this is not the case with Eos [4]) it may miss actual dependencies. Complementing such statically extracted dependencies with dependencies obtained by a dynamic analysis may help an engineer differentiate between *potential* dependencies and those that *actually occur* during runtime. As a result, the engineer can find potential errors or decide how to restructure a system. To help engineers with such activities, ViVA provides an operation called `show_dependencies`. This operation enables component visualization based on three types of message dependencies: static analysis-based, dynamic analysis-based, and overlapping dependencies. Overlapping dependencies are those that are identified by, both, the static and dynamic analyses.

To use the `show_dependencies` operation, an engineer accesses the ViVA console, first to input the name of a desired component *Comp* and then to invoke the operation. This invocation causes ViVA to change the colors of components that depend on *Comp* based on the type of dependency through which they have been identified: those components that have been identified by static analysis are colored blue; those that have been identified by dynamic analysis are colored red; finally, those that are involved in overlapping dependencies are colored purple.

5. CONCLUSION AND FUTURE WORK

Our experience with applying ViVA to several DEB systems has shown it to be promising for their understanding, debugging, and refactoring. We are currently pursuing a user study in which ViVA is used in the context of different development tasks, with the goal of assessing the extent to which ViVA helps engineers successfully complete those tasks. Another avenue of recent work has been leveraging ViVA to detect and visualize instances of architectural decay [3] in a system.

6. ACKNOWLEDGEMENTS

This work has been supported by the National Science Foundation under award numbers 1117593, 1218115, and 1321141. The work is also supported by the Intelligence Advanced Research Projects Activity (IARPA) under contract number number N66001-13-1-2006 and the Defense Advanced Research Projects Agency (DARPA) under contract number N66001-11-C-4021. Finally, the work has been supported in part by Infosys Technologies Ltd.

7. REFERENCES

[1] F. Biscotti and A. Raina. Market Share Analysis: Application Infrastructure and Middleware Software, Worldwide, 2011. *Gartner Market Research Report*, 2012.

[2] J. Correira et al. Market Share: AIM and Portal Software, Worldwide, 2005. *Gartner Market Research Report*, 2006.

[3] J. Garcia et al. Toward a Catalogue of Architectural Bad Smells. In *Conf. on Quality of Software Architectures*, 2009.

[4] J. Garcia et al. Identifying message flow in distributed event-based systems. In *ESEC/FSE*, 2013.

[5] K. S. Gatiin. Trials and tribulations of debugging concurrency. *ACM Queue*, 2004.

[6] Generic Modeling Environment (GME). `http://isis.vanderbilt.edu/projects/gme/`.

[7] T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *ICSE*, 2010.

[8] S. Malek et al. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE TSE*, 2005.

[9] R. Taylor et al. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2008.

[10] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.