Impact Analysis for Event-Based Components and Systems

Daniel Popescu
Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
dpopescu@usc.edu

ABSTRACT

In my dissertation, I aim to develop a dependence-based impact analysis technique for event-based systems and event-based components that communicate via messages. This paper motivates the problem, summarizes the open challenges and outlines proposed solution and evaluation strategies.

1. INTRODUCTION

In recent years, systems that have been developed using event-based (also referred to as message-oriented) middleware platforms have become widespread. A Gartner study determined that the market size for message-oriented-middleware licenses was about $1 billion in 2005 [5]. In event-based systems, components do not directly call other components via explicit references, but instead use implicit invocation to transfer data and notifications of events by sending messages. Software connectors, e.g., event buses or brokers, then route these messages to the correct recipients. Messages are either broadcast to certain component groups or sent to recipients who previously subscribed to them. Consequently, components in event-based systems are highly decoupled and allow highly scalable, easy-to-evolve, concurrent, distributed, heterogeneous applications. The event-based software architectural style [13] is especially used in user-interface software and wide-area applications such as financial markets, logistics, and sensor networks.

As successful software systems are typically maintained for many years and changed constantly during their lifetimes, maintenance engineers need techniques that specifically address several challenges induced by the event-based architectural style. One of the key techniques for effective software maintenance is *impact analysis* [4]. Impact analysis helps to determine the scope of a change request in order to estimate whether the value of the change request justifies the estimated costs. To compute the impact of change requests, researchers have developed two major impact analysis techniques [3]: (1) traceability analysis (e.g., tracing relationships between requirements and code elements) and (2) dependence analysis. In my work, I focus on static dependence analysis techniques that compute how changes to a source code element affect other source code elements.

Current dependence analysis approaches are restricted in the types of dependencies they can extract from event-based systems. First, most impact analysis techniques compute a *system dependence graph* (SDG) [14, 15] that represents the control-flow and data-flow dependencies of a whole system. These techniques assume that all source code of the analyzed system is available. However, in event-based systems, the source code of the off-the-shelf middleware infrastructure is typically unavailable. Since event-based components communicate with each other via the off-the-shelf middleware infrastructure, the missing middleware source code renders the SDG computation unobtainable. Second, assuming the source code of the middleware infrastructure is available, it would typically add 100,000 SLOC or more to the SDG computation. SDGs of such sized systems easily contain several million edges and vertices [2], resulting in intractable dependence slices that contain details that are irrelevant for analyzing the impact of changes to an event-based application. Third, current dependence analysis techniques cannot recover the concept of a message from low-level source code facts. Therefore, current techniques are unable to determine the impact of changes to the message communication.

To enable impact analysis for event-based systems, I propose *Helios*, an approach to determine an SDG that captures message dependencies. A technique that produces such a *message dependence graph* does not currently exist. Before Helios can compute message dependencies, it needs to recover the incoming and outgoing message interfaces for each component because current event-based components do not explicitly reveal these interfaces. I propose to create a technique that utilizes control-flow and data-flow analyses to extract the interfaces of an event-based component. This technique should support the three message interface types common to today's event-based systems [9]: (1) type-based, (2) subject-based and (3) record-based message interfaces.

A message dependence graph requires recovering two types of message-based dependencies: (1) inter-component dependencies and (2) intra-component dependencies [12]. An *inter-component dependency* describes how a component influences a receiver component by publishing a certain message. Inter-component dependencies by themselves are insufficient to determine change impact because two components might be dependent on each other through a chain of message dependencies. For example, Component A sends e1, which causes Component B to send e2, which changes

ICSE '10, May 2-8 2010, Cape Town, South Africa

the state of Component C; consequently the state of Component C is dependent on e2 and e1. *Intra-component dependencies* fill the missing link in a causality chain. They describe how outgoing messages of a component are dependent on incoming messages of the same component. Intra-component dependencies are caused by a component's internal control-flow and state.

Intra-component *state-based dependencies* occur when executions caused by different message types write to and read from the same component state. I propose to track state-based dependencies through component variable access specifications [1] that are enforced in the implementation. Helios will calculate control-flow- and state-based intra-component dependencies by producing an interprocedural control flow graph that is annotated with the recovered message types and access permission information. Helios will determine *inter*-component dependencies based on the incoming and outgoing component interfaces and the system's overall structural configuration [13].

The remainder of the paper is organized as follows: Section 2 presents the background, including (1) an overview of different types of dependencies in an event-based system, and (2) a discussion of three common message interface types. Section 3 describes the research hypotheses. Section 4 describes the proposed approach. Section 5 concludes with the proposed empirical evaluation.

2. BACKGROUND

2.1 Classifying Message Dependencies

Figure 1 shows an example event-based system that will help to illustrate the three types of message dependencies. (1) *Intra-component dependencies resulting from control flow* are dependencies that occur due to an operation whose invocation is caused by the receipt of a message at a component's message sink which, in turn, produces one or more messages at the component's message source. Component A in Figure 1 depicts such a dependency: the intra-component dependency of e3 on e0. (2) *Intra-component dependencies based on state* are dependencies of the kind depicted in Component C in Figure 1. The component variable in C is written to by an operation executed as a result of e4. Another operation that reads that component variable executes as a result of e3. (3) *Inter-component dependencies that occur across connectors* are dependencies of the kind depicted in Figure 1 between the message source of A and the message sink of C. By extracting these three different types of dependencies, Helios constructs a complete message dependence graph.

2.2 Message Interfaces

Event-based components can have two types of interfaces: A *message source* is a component's interface that a component invokes to publish messages, and a *message sink* is a component's interfaces that a connector invokes to transfer a message to the component. In current event-based systems, a component does not reveal the specific message types that the component is consuming or producing. Event-based components offer only *Ambiguous Interfaces* [6]. An Ambiguous Interface offers only one public message sink and message source interface, although its component consumes and publishes multiple specific message types. As a consequence, the component accepts all incoming messages

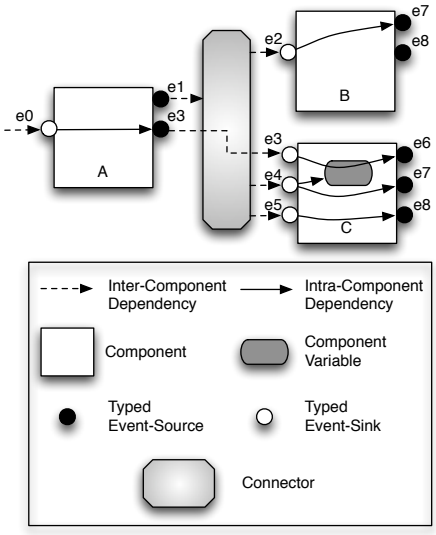

Figure 1: Inter- and intra-component Dependencies

through its single message sink and internally dispatches to other services or methods. In addition, since every component's interface only offers one message sink and message source, the accepted type is consequently overly general.

Helios needs to recover the message interfaces of each event-based component. Helios will support the three different message interface types common to today's event-based systems [9]: (1) type-based, (2) subject-based and (3) record-based message interfaces. *Type-based message interfaces* are directly mapped to programming language types. In *subject-based message interfaces*, a string names each message, allowing complex message names and naming hierarchies (e.g., "/Weather/USA"). A component that dispatches subject-based messages utilizes string comparisons to identify how a message should be processed. Finally, in *record-based message interfaces*, messages are records consisting of a set of name/value pairs called attributes. A component that dispatches record-based messages analyzes whether a message contains a certain attribute name or value.

3. HYPOTHESES

A maintenance engineer needs to inspect all message-based inter-component and intra-component dependencies in order to accurately recover the impact of changes in an event-based system. The goal of my research is to reduce the effort required in doing this. This potential effort reduction is based on the precision of the dependence analysis, i.e., the degree to which extracted dependencies can actually occur in the event-based system at runtime. The overall message dependence graph depends on recovering precisely the edges (e.g., intra-component dependencies) and nodes (e.g., message source interfaces).

My research aims to test the following hypotheses.

Hypothesis 1: Assuming the source code of the message-oriented middleware is unavailable, I hypothesize that a data-flow analysis technique can be devised that extracts more precise message interfaces than any published state-of-the-art heuristic. The technique will also extract more precise intra-component dependencies than any published state-of-the-art heuristic.

Hypothesis 2: I hypothesize that the distinct message

interface recovery approaches for (1) type-based, (2) subject-based and (3) record-based message interfaces will recover interfaces of similar precision.

Hypothesis 3: An approach can be devised that computes summary information for an event-based component. The summary information corresponds to a component's message dependence graph. The graph includes nodes for the component's state and the message interfaces, and the edges for the component's intra-component dependencies. I hypothesize that a component's summary information will consist of significantly fewer nodes and edges than the component's SDG.

Hypothesis 4: When the system-wide impact of changing one event-based component needs to be repeatedly determined, an approach can be devised that is able to reuse the previously computed summary information of each unchanged component. For such cases, I hypothesize that the summary information will help to substantially reduce the number of nodes and edges in the event-based system's SDG. Furthermore, adding components to a system will decrease the ratio depicted in Equation 1. This hypothesized trend would indicate that summary information improves the scalability of impact analyses for event-based systems.

$$\frac{|Component's\ SDG| + |Summary\ Information|}{|System's\ SDG|} \quad (1)$$

4. APPROACH

A preliminary version of Helios [10] confirmed the importance and feasibility of extracting a message dependence graph for event-based systems. In the preliminary version, each event-based application component explicitly exposed its message sink interface. For this relaxed problem, Helios was able to extract a message dependence graph from applications that used type-based message interfaces. The preliminary studies demonstrated that Helios can yield large savings in the effort required to understand and assess the impact of changes to an event-based application.

This section outlines the foundations of Helios and how the preliminary version is currently extended to solve the more complex open challenges. Helios extracts a message dependence graph in three distinct phases. Section 4.1 outlines how Helios extracts the message interfaces for each component. Section 4.2 outlines how Helios extracts intra-component dependencies. Finally, Section 4.3 describes how the intra-component message dependencies can be merged into a system message dependence graph.

Helios will assume that event-based components only communicate with each other using messages. Mixing an explicit invocation style with the event-based communication style decreases the adaptability benefits of the event-based style [6]. Since the event-based style is typically used to achieve loose coupling between components, Helios focuses on systems in which components only communicate through messages. All of the systems used in our preliminary evaluation were able to satisfy this assumption.

4.1 Extracting Message Interfaces

The preliminary version of Helios did not have to identify message sink interfaces, and it was not able to analyze subject-based or record-based message interfaces. The proposed extended Helios will overcome these limitations. Helios will start the analysis of a component's implementation by creating an interprocedural control-flow graph (ICFG) of each component. To extract message interfaces from a component's implementation, Helios will need to collect additional information about the incoming message. The type of information depends on the message interface type the system is using. In the case of type-based interfaces, Helios will try to identify locations where the incoming message is casted to a more specific type. In the case of subject-based interfaces, Helios will try to identify expressions that compare the name of an incoming message to a string. Finally, in the case of record-based interfaces, Helios tries to identify the locations on which a component accesses attributes of the incoming message. For each location, Helios will add annotations about the identified type information to the ICFG.

Helios will use a data-flow analysis technique on the ICFG to propagate the annotations to the message sink method. For example, if two different attributes are accessed on the same path that is reachable from the message sink, the data-flow analysis merges the two attributes to one record-based message. The ICFG contains also regions that consist of disjoint nodes that are independently reachable from the message sink. Annotations that reach such distinct regions indicate differing message types. For each independent region, Helios will create a message node for the merged reachable annotations and will add an edge from the message node to the entry node of the region.

Helios will have to perform a similar analysis for message source interfaces. The details are omitted because of space limitations. The added message nodes in the ICFG will denote the extracted message interfaces.

4.2 Intra-Component Dependencies

Helios will add control-flow-based intra-component dependencies by adding a directed edge to the ICFG for each existing path between a message sink node and a message source node. To find all paths, Helios will perform a depth-first search on the ICFG starting at each message sink node.

In order to add state-based dependencies to the component message dependence graph, Helios currently utilizes a previously published static type system for OO programs that is based on access permissions [1]. Access permissions enable modular tracking of typestates (richer notions of states that are similar to statecharts) and aliasing information (objects being referenced from multiple locations). Access permissions can also express whether a reference allows modifying or only reading access to a referenced object.

To track state access, Helios requires that all fields of the component are mapped into a state. It also ensures that each calling method needs to own the permission that the called method or accessed state requires. As a consequence, in the preliminary version of Helios, permission annotations on the `consume` methods, which implemented the message sink interface, revealed whether an incoming message modified a component's state, read from the state, or was independent of it. Therefore, all state-based dependencies could be determined by only inspecting the annotations on a component's `consume` methods. This approach was possible because each message type was mapped to a specific independent `consume` method. For the extended Helios, this approach fails because all message types share the same `consume` method. Consequently, the extended Helios will have to create a data-flow analysis that determines the permissions that reach each message sink node of the ICFG. Helios will then be able

to extracts the state-based dependencies from the identified access permissions.

4.3 Inter-Component Dependencies

While intra-component dependencies facilitate the understanding of a component, they also enable more precise inter-component dependencies. The intra-component analysis is able to reveal that possible inter-component dependencies do not manifest themselves because of extracted intra-component dependencies, message sinks, and message sources.

Helios creates an inter-component dependence graph by matching the recovered typed message sources of components with the recovered typed message sinks of other components. Helios utilizes the structural configuration of an event-based system to identify message sinks that could be reached from a message source. The structural configuration describes how components and connectors are connected to each other. In some event-based systems, components may be connected to multiple connectors, while in other systems all components communicate through the same connector. For example, in Figure 1, if Component A's message source were not connected to the same connector as Component C's message sink, the depicted inter-component dependency would not exist. If a system's structural configuration is unavailable, Helios assumes that all components can potentially exchange messages with each other.

5. EVALUATION

The preliminary version of Helios [10] was empirically evaluated for event-based applications spanning four different middleware platforms. For my dissertation research, I am planning to include additional larger subject applications. Table 1 gives an overview of the subject event-based platforms and applications that were used for the preliminary evaluation. Column *App Type* gives a short description of the application's domain; *SLOC* shows the source lines of code of each application; *Comp* shows how many component objects each application has; *Msg Types* totals the different message types in each application; and *M/W Platform* names the middleware that provides the connector services to the application. All applications have been developed independently, they are Java-based, and their architectures have been described in prior publications [7, 8, 11, 13]. The preliminary evaluation tested parts of Hypothesis 1. The evaluation showed that Helios typically extracts more precise message sources and intra-component dependencies than two alternative state-of-the-art approaches.

To test Hypothesis 1 and 2, I will run the extended Helios on the extended list of subjects applications. Since the applications of the preliminary evaluation were refactored to explicitly expose their message sinks, the refactored applications will help to evaluate the overall precision of the proposed message interface recovery techniques. The recovery techniques would achieve perfect precision if they recover the same interfaces as specified in the refactored applications.

To test Hypothesis 3, I will utilize state-of-the-art dependence analysis approaches [14, 15, 2] to create an SDG for each component of each subject application. Helios will then recover and add the message sink and source nodes to the SDG. As the next step, Helios will create the summary information for each component. Finally, I will compare the number of nodes and edges in the component's SDG and in the component's message dependence graph.

App Name	App Type	SLOC	Comp	Msg Types	M/W Platform
KLAX	Arcade Game	4.5K	14	85	c2.fw [7]
DRADEL	Architectural Type Checker	10.8K	8	82	c2.fw [7]
ERS	Emergency Response	7.1K	11	56	Prism-MW [7]
Stoxx-Sub-system	Stock Ticker Notification	1K	4	14	REBECA [8]
jms2009-PS	Benchmark for JMS-Providers	18.6K	4	19	JMS [11]

Table 1: Experimental Subjects

To test Hypothesis 4, I will try to identify trends for each subject application. Initially, I will evaluate Equation 1 assuming that each application only consists of two components. Subsequently, components of the same application are iteratively added to the system and Equation 1 is reevaluated for each iteration. The evaluation will also determine whether the order of adding components influences the identified trends significantly.

6. REFERENCES

[1] K. Bierhoff et al. Practical API Protocol Checking with Access Permissions. In *Proc. ECOOP*, July 2009.

[2] D. Binkley et al. Empirical study of optimization techniques for massive slicing. *ACM TOPLAS*, 30(1):3, 2007.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Pr, 1996.

[4] P. Bourque and R. Dupuis. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[5] J. Correira and F. Biscotti. Market Share: AIM and Portal Software, Worldwide, 2005. *Gartner market research report, Gartner, June*, 2006.

[6] J. Garcia et al. Toward a catalogue of architectural bad smells. In *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, 2009.

[7] N. Medvidovic et al. The role of middleware in architecture-based software development. *Int. J. of Softw. Eng. and Knowl. Eng.*, 13(4), 2003.

[8] G. Mühl. *Large-scale content-based publish/subscribe systems*. PhD thesis, Darmstadt University of Technology, 2002.

[9] G. Mühl et al. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., 2006.

[10] D. Popescu et al. Helios: Impact analysis for event-based systems. Technical Report USC-CSSE-2009-517, USC-CSSE, 2009.

[11] K. Sachs et al. Benchmarking of Message-Oriented middleware. In *SIGMETRICS/Performance 2009 Demo Competition*, June 2009.

[12] J. Stafford and A. Wolf. Architecture-level dependence analysis for software systems. *Int. J. of Softw. Eng. and Knowl. Eng.*, 2001.

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

[14] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[15] B. Xu et al. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2), 2005.