Helios: 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

1. PROBLEM AND MOTIVATION

The event-based software architectural style [4] is widely used in the domain of user-interface software and wide-area applications (e.g., financial markets, logistics, and sensor networks). A Gartner study determined that the market size for message-oriented (a.k.a event-based) middleware licenses was about $1 billion in 2005 [2]. In event-based systems, components do not directly call other components, but rather indirectly using messages or events. However, this high decoupling and use of implicit invocations render an event-based system more difficult to analyze since, in the absence of explicit dependency information, an engineer has to assume that any component in the system may potentially interact with, and thus depend on, any other component.

One of the key techniques for effective software maintenance used to deal with such an analyzability challenge is *impact analysis*, which helps to determine the scope of a change request. In this work, I focus on static dependence-based impact analysis techniques that compute how changes to a source code element affect other source code elements.

Current dependence analysis approaches suffer from two limitations regarding dependency extraction in event-based systems. First, most impact analysis techniques compute a *system dependence graph* (SDG) [1] that represents the control-flow and data-flow dependencies of a whole system. An event-based middleware often adds 100,000 SLOC to an application. SDGs computed from systems of this size easily contain several million edges and vertices [1] which results in intractable dependence slices. Second, current dependence analysis techniques cannot recover the concept of a message from low-level source code facts.

To address these two limitations and enable impact analysis for event-based applications, I propose *Helios*, a novel approach to determine a dependence graph that captures message dependencies by sequentially analyzing the source code of each event-based component. A technique that produces such a *message dependence graph* does not currently exist.

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

2. BACKGROUND AND RELATED WORK

A message dependence graph requires recovering two types of message-based dependencies: (1) inter-component dependencies and (2) intra-component dependencies. An *inter-component dependency* describes how a sender component influences a receiver component by publishing a certain message. Figure 1 depicts such a dependency between the message source of Component A and the message sink of Component C. An *intra-component dependency* describes 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. Component A in Figure 1 depicts a control-flow-based intra-component dependency: the dependency of e3 on e0. Intra-component *state-based dependencies* occur when executions caused by different message types write to and read from the same component state. In Figure 1, the incoming message of type e4 causes an operation that modifies the state of Component C but does not publish any message. Another operation that reads the same state and publishes a message of type e6 executes as a result of e3.

There is a rich body of work [1] on analyzing the dependencies and the impact of changes in software systems that use explicit rather than implicit invocation. *Code*-based dependence analysis techniques share a key shortcoming, precluding their direct applicability in event-based systems: they are unable to recover the concept of a message from source code facts. On the other end of the abstraction spectrum, there has been research in analyzing event-based *modeling* approaches [4]. These approaches provide no guidance for analyzing implementation-level message dependencies.

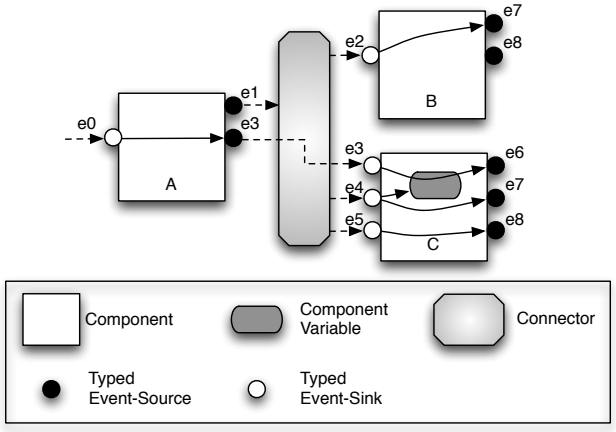

Figure 1: Inter- and Intra-Component Dependencies

3. APPROACH

This section describes how Helios creates a message dependence graph (MDG) from the source code of an event-based system. The MDG is needed to determine the impact of changes in such systems. Helios currently assumes that the event-based components utilize typed-based messages and that each incoming message can be dispatched to a unique message sink method. Helios also assumes that event-based components only communicate with each other using messages. All of the evaluated systems spanning four different middleware platforms were able to satisfy these assumptions.

Helios extracts a message dependence graph in four distinct phases. (1) Helios extracts the message source interfaces for each component, (2) the control-flow-based intra-component dependencies and (3) the state-based intra-component dependencies. Finally, (4), Helios adds the inter-component dependencies to the message dependence graph.

Extracting message source interfaces: As the first step, Helios computes an *interprocedural control-flow graph* (ICFG) of each component. Subsequently, for each method that implements the message sink interface, Helios analyzes the type of the method's message parameter, creates a message node for the message type and adds an edge from the message node to the method node representing the message sink interface. As the next step, Helios determines all message sources of the component. For each method `m`, Helios identifies whether `m` calls the message source interface of the middleware. In the case where a call to the message source is identified, Helios determines the type of the argument that is passed to the message source. Helios uses intra-method data flow analysis to determine where the argument variable was declared. The declaration signifies the type of the outgoing message. After identifying the message type, Helios creates a message node for the type and adds an edge from the method node, which represents the method calling the message source, to this message node.

Control-flow-based 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

State-based intra-component dependencies: To track state access, Helios utilizes *access permissions* that can express whether a reference (e.g., a pointer) allows modifying or only reading access to a referenced object.

Helios requires that all field variables of a component are mapped into a state. It also ensures that each calling method needs to own the permission that the called method or accessed field variable requires. As a consequence, permission annotations belonging to the methods, which implement the message sink interface, reveal whether an incoming message modifies a component's state, read from the state, or is independent of it. Therefore, all state-based dependencies can be determined by only inspecting the annotations of a component's message sink methods.

Inter-component dependencies: Helios creates an inter-component dependence graph by matching the recovered typed message sources of components with the message sinks of other components. Helios also utilizes any available structural configuration of the 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. 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.

4. RESULTS AND CONTRIBUTIONS

Helios was evaluated with existing Java-based applications that were written for four different event-based middleware platforms: c2.fw, Prism-MW, REBECA and JMS [3]. The evaluated systems include an architectural type checker, an arcade game, an emergency response system, a stock ticker system and the jms2009-PS benchmark for evaluating the performance of message-oriented middleware servers.

The objective of this study was to show that the precision of two alternative strategies for recovering a message dependence graph is significantly lower than the precision achievable by Helios. The first alternative strategy (s1) is called the *import heuristic*: Since the import heuristic is unaware of message sinks and sources, it assumes that a component consumes and publishes all, and only those, event types that are declared in files that are imported or included within it. The second alternative strategy assumes that an engineer can dynamically query the connector for each component's message source and message sink interface. This strategy is called (s2) the *black-box approach*: In the absence of other information, each message source is assumed to be dependent on each message sink within a component.

I briefly summarize the results of the evaluation which is available in [3]. The evaluation assessed two factors that affect the precision of dependence analysis: (p1) the precision of recovered message source interfaces and (p2) the precision of recovered intra-component dependencies. For each of the five heterogeneous systems, Helios outperformed both the black-box and import heuristics. Compared to the import heuristic (e.g., comparing (p1) to (s1)) using Helios yielded a median improvement in precision of 50%, and compared to the black-box approach (e.g., comparing (p2) to (s2)) Helios yielded a median improvement in precision of 33%.

Contributions: Software maintenance costs frequently dwarf development costs. Impact analysis is a key tool used by maintenance engineers. Helios is the first impact analysis technique specifically designed for event-based systems. Unlike the existing attempts, Helios is able (1) to detect the concept of a message within source code facts and (2) to cover the complete space of dependencies in an event-based system — inter-component control flow-based dependencies, intra-component control flow-based dependencies, and intra-component state-based dependencies.

5. REFERENCES

[1] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers: Advances in Software Engineering*, 62:105, 2004.

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

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

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