Helios: Impact Analysis for Event-Based Systems

Daniel Popescu[1] Joshua Garcia[1] Kevin Bierhoff[2] Nenad Medvidovic[1]

[1]Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{dpopescu,joshuaga,neno}@usc.edu

[2]Two Sigma Investments
379 West Broadway
New York, NY 10012, USA
kevin.bierhoff@twosigma.com

ABSTRACT

Event-based software systems contain highly-decoupled components that interact by exchanging messages via implicit invocation, thus allowing flexible system composition and adaptation. At the same time, these inherently desirable properties render an event-based system more difficult to understand and evolve 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. Software analysis techniques that have been used successfully in traditional, explicit invocation-based systems are of little use in this domain. In order to aid the understandability of, and assess the impact of changes in, event-based systems, we propose Helios, a technique that combines component-level (1) control-flow and (2) state-based dependency analysis with system-level (3) structural analysis to produce a complete and accurate *message dependence graph* for a system. We have applied Helios to applications constructed on top of four different event-based implementation platforms. We summarize the results of several such applications. We demonstrate that Helios enables effective event-based impact analysis and quantify its improvements over existing alternatives.

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 [8]. 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 [28] 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 [15], maintenance engineers need techniques that specifically address several challenges that are induced by the event-based architectural style.

In general, changing complex software systems often results in unintended effects that can be very costly or can have serious negative consequences [16]. At the same time, complex code changes require substantial engineering effort and can often only be justified when the added value exceeds the maintenance costs [5, 30]. To estimate the maintenance risks and costs of a given change request, researchers have developed two major impact analysis techniques [6]: (1) tracebility analysis and (2) dependency analysis. In this paper, we focus on dependency-based impact analysis techniques that show how changes to a source code element affect other source code elements. Most impact analysis techniques focus on procedural, functional, and object-based dependencies [2, 13, 14, 23, 24], while some techniques perform domain-specific analyses such as impact analysis of database schema changes [18]. No current impact analysis techniques can be readily applied to event-based applications because the event-based architectural style utilizes implicit invocations. In particular, the impact of changes on event-based system components is more difficult to analyze because these components, by design, do not know the receivers of the messages they send.

Most impact analysis techniques utilize dynamic or static slicing [29]. Dynamic slicing only considers dependencies based on a given input (e.g., a test case), while static slicing focuses on dependencies based on all potential inputs. In this work, we focus on static slicing. A program slice consists of program statements that could potentially affect the values computed at a specific program statement; all other statements of the program are "sliced away". The static slicing problem can be restated as a graph reachability problem of a program dependence graph (PDG). To solve slicing across procedure boundaries, Horwitz et. al. introduced the system dependence graph (SDG) [11]. Larsen and Harrold introduced class dependence graphs (CDGs) to account for object-oriented features [13]. Similar to the way the SDG and CDG needed to extend the PDG to fit their own respective domains, we do the same for the domain of

ICSE 2010 Capetown, South Africa

event-based systems. Specifically, we create a PDG that captures message dependencies. A technique that produces such a *message dependence graph* does not currently exist.

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 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 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 causality chains. They describe how outgoing messages of a component are dependent on incoming messages of the same component. These intra-component dependencies are caused by a component's internal control-flow and its state.

We present *Helios*, an approach to determine message dependence graphs. By slightly constraining the implementation of an event-based system, while retaining its adaptability benefits, Helios creates the pre-requisites for the computation of inter- and intra-component dependencies. We call an event-based application that follows these constraints *Helios-compliant* because this application can be analyzed for event-based change impact. An event-based application becomes Helios-compliant if it follows Helios's constraints:

1. Helios-compliant applications must use middleware platforms that support a standard message sink and message source interface for each component, as well as connectors that route messages from message source interfaces to appropriate message sink interfaces. This constraint is reasonable for many event-based systems [8].

2. Helios-compliant applications must use object-oriented programming languages that support strong static typing and either reflection mechanisms or multiple dispatch (i.e., methods that can be dynamically dispatched based on the runtime subtype of an argument of an method) [7]. Many modern OO programming languages such Java or C# fulfill these requirements.

3. Helios-compliant applications use type-based filtering in which message types are explicitly mapped to programming language types, allowing type-safe communication between event-based components.

For Helios-compliant systems, Helios enables the calculation of *intra*-component dependencies through (1) component variable access specifications [4] and (2) typing a component's incoming and outgoing interfaces. Helios calculates intra-component dependencies by producing a component call graph that is annotated with message types and access permission information. Helios determines *inter*-component dependencies based on the incoming and outgoing component interfaces and the system's overall structural configuration [19]. Finally, inter- and intra-component dependencies can be merged to form a complete message dependence graph on which an impact analysis algorithm can be executed.

We have evaluated Helios with existing event-based applications that were written for four different event-based middleware platforms. The evaluated systems include an architectural type checker [20], an arcade game [20], an emergency response system [28], a stock ticker notification system [21] and the jms2009-PS [25] benchmark that is based on the offical SPECJms2007 [26] benchmark for evaluating the performance of enterprise message-oriented middleware servers. For each system we are able to demonstrate that Helios enables an event-based impact analysis, and to quantify the degree to which extracted inter-component and intra-component dependencies correspond to dependencies in the event-based system. Our results indicate that, as long as an application is Helios-compliant, our approach produces neither false-positive nor false-negative results.

The remainder of the paper is organized as follows. Section 2 presents the background, including (1) the definitions of the terminology used in this paper, (2) an overview of different types of dependencies in an event-based system, and (3) a discussion of the choices available to an engineer in constructing an event-based system with an overview of the choices made in the related literature. Section 3 describes the approach taken in Helios, while Section 4 presents our evaluation results. The paper concludes with a summary of lessons learned and a discussion of future work.

2. BACKGROUND

2.1 Terminology

In event-based systems, components, i.e., the units of computation and data, communicate using *messages* which carry either *notifications* or *anonymous requests* [22]. An *event* is an important occurrence of anything that can be observed by a component (e.g., a change of a component's state). A *notification* is a datum that describes an event, and an *anonymous request* is a directive that expects a reaction from an unknown recipient. A *message* is a data container that conveys notifications and requests. Addition, removal, and updating of components during runtime can be achieved with relative ease because components do not have references to each other.

Event-based components can be classified into two types, *producers* and *consumers*. An event-based component can assume both roles simultaneously in a given system. Consumer components can explicitly subscribe to messages that they intend to process. When a producer publishes a message, a software connector routes the message to the appropriate subscribers based on the network configuration, routing policies, and message filters. *Message filters* are Boolean functions that check whether a component should process or publish a message. The filtering mechanisms help to reduce message load on the network by ensuring messages are only routed to the designated consumers. 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.

2.2 Classifying Message Dependencies

Figure 1 shows an example event-based system that will help to illustrate the inherent challenges and solutions throughout this paper. We will create a complete message dependence graph by extracting 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, we construct a complete graph of message dependencies.

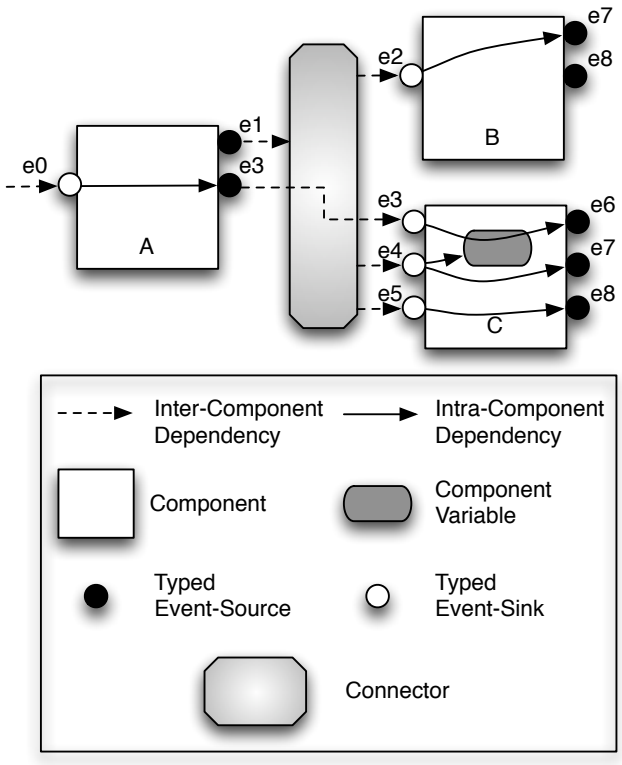


Figure 1: Inter- and Intra-Component Dependencies

2.3 Categorizing Event-Based Systems

This section contains a discussion of design choices available to developers of event-based systems and the specific choices we took to create applications that can be analyzed by Helios. Similar to the manner in which a statically typed programming language facilitates static analyses that are not possible with dynamically typed languages, design choices made about the event-based middleware platform and event-based application will facilitate or hamper the analysis of event-based systems. The three key dimensions along which event-based systems differ are the employed (1) filtering mechanisms, (2) communication styles, and (3) implementation languages.

In general, an event-based system needs a *message filtering mechanism* that prevents each component from having to process every published message. A component's message filter also reveals the component's message sink and message source interfaces since filters define what kind of messages a component may consume and produce. Common filtering mechanisms are (1) channels, (2) subject-based filtering, (3) content-based filtering and (4) type-based filtering [22].

Message *channels* are a filtering mechanism in which each component selects named channels for exchanging messages. A component can only receive messages from pre-selected channels. The disadvantage of channels is that a component has an explicit reference on the channel and therefore cannot be easily adapted to changing system configurations.

Subject-based filtering uses string matching to filter messages. In this filtering mechanism, each message is named by a string. A subject-based filter can use regular expressions to match messages, allowing complex message names and naming hierarchies (e.g., "/Weather/USA/LosAngeles"). The benefit of this filtering mechanism is that it is easy to implement in any common programming language. However, subject-based filtering hampers identifying message sink and message source interfaces in a component's source code. Both interface types need to be extracted from the subject-based filters and the message names that the implementation generates, but the latter are potentially generated in undecidable ways at runtime.

Content-based filtering allows the most refined filters by filtering over the whole content of a message. While being a powerful filtering mechanism, systems using content-based filtering complicate dependence analysis because the content of a message can be created in intractable ways. Therefore, content-based filtering causes similar challenges to subject-based filtering.

Finally, *type-based filtering* uses explicitly defined types to filter messages. In this filtering mechanism, message types can be directly mapped to programming language types. Consequently, type-based filtering enables type-safe communication between event-based components [9]. Moreover, since type-based filtering integrates well with programming language types, it facilitates identification of message types in the source code and therefore eases extraction of component interfaces. For these reasons, we focus on analyzing dependencies based on typed messages, typed message sinks, and typed message sources in Helios.

Components in event-based systems can rely solely on exchanging messages or they can utilize several *communication styles* in tandem. However, mixing communication styles can decreases the benefits of each individual style [10]. Consider the example in Figure 1. If Component A had an additional explicit reference to Component C (e.g., a pointer), adaptability of the system would decrease because modifications to Component C would likely affect the reference of Component A. As a consequence, the coupling between these components increases and the adaptability of the whole system decreases. 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.

Finally, we surveyed the 18 event-based middleware platforms covered in [22] regarding the implementation language each one supports. Statically typed OO programming languages such as Java, C# and C++ are used in the widest number of event-based middleware platforms. In addition, the static typing of the languages facilitates integrating type-based message filtering with an application's implementa-

tion language. Consequently, we focus on mainstream OO programming languages in Helios. In particular, our evaluation was performed on Java-based middleware systems.

3. APPROACH

In this section we describe how Helios creates a message dependence graph from the source code of an event-based system. The message dependence graph is needed to determine the impact of changes in such a system. Before we discuss each phase of Helios, we will first clarify the conditions that need to be fulfilled to analyze message dependencies with Helios. The conditions are based on the above discussion of design choices: (1) the system is implemented in an OO programming language, (2) the components comply to type-based filtering, and (3) each message type is bijectively mapped to a programming language type.

Whenever a connector wants to deliver a message to a component, the connector dispatches the message to one of the component's message sinks based on the type of the message. Figure 2 helps to clarify this mechanism. The figure shows a Java implementation of a simple Component C. In this implementation, the `consume` methods realize the component's message sink and the `publish` method realizes the component's message source. The annotations, denoted with @, will be explained in Section 3.2. The instance variable in line 7 realizes the component's state, while `E1`, `E2`, ... `E8` represent specific message types, which are subtypes of the general message type `EventMessage`. Component C has `consume` methods for message types `E3`, `E4`, and `E5`. When the connector receives a message of one of those three types, the message needs to be dispatched to the right `consume` method. Mainstream OO languages such as Java, C++, or C# cannot dispatch a method based on the runtime type of a method parameter. This is a well-known problem in programming languages, called the *Binary Method Problem* [7]. While Helios is able to incorporate any solution to this problem discussed in [7], our analyzed Helios-compliant applications all utilize *explicit type tests*: each component inspects the runtime type of an incoming message to decide whether it can process the message. When the component is able to consume the message's type, it explicitly casts the message to that type (Figure 2, lines 11-17).

As mentioned above, Helios assumes that event-based components only communicate using messages. In our example and in the later evaluation, we accomplish this communication constraint by ensuring that no reference to the component object escapes (i.e., is passed to) to other application objects or methods. The *component object* is the class instance that implements the message sink, parts of the component's application-specific logic, and the interface to the message source. Only the connector code can have a reference to a component object. This constraint can be guaranteed using static analysis [1].

Helios extracts a system message dependence graph in three distinct phases, detailed in the remainder of this section and depicted in Figure 3. Section 3.1 describes how Helios extracts message dependencies inside a component and composes them into an intra-control-flow message dependence graph (Figure 3a-c). Section 3.2 describes how annotating data access with permissions facilitates determining data-flow dependencies, which need to be added into the control-flow-based message dependence graph (Figure 3d). Finally, in Section 3.3 we describeğ how the intra-component

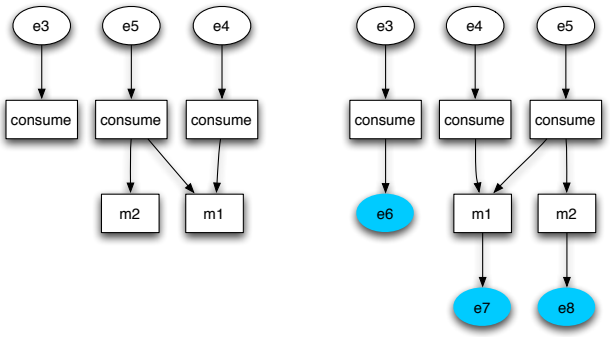
```
1  /*Component C in Java*/
2  @ClassStates({
3    @State(name = "counter",
4      inv = "share(counterObj)")})
5  @States(dim="counter")
6  public class C extends Component{
7    Counter counterObj = new Counter();
8
9    /* generated method */
10   @Share("counter")
11   public void consume(EventMessage e){
12       if (e instanceof E3){
13         consume((E3) e);};
14       if (e instanceof E4){
15         consume((E4) e);};
16       if (e instanceof E5){
17         consume((E5) e);};}
18
19   @Pure("counter")
20   public void consume(E3 e){
21     int i = counterObj.getState();
22     if (i > 0)publish(new E6(i));}
23
24   @Share("counter")
25   public void consume(E4 e){
26     counterObj.increment();
27     m1();}
28
29   public void consume(E5 e){
30     m1();
31     m2();}
32
33   private void m1(){publish(new E7());}
34
35   private void m2(){publish(new E8());}
36 }
37
38 class Counter{
39   @In("alive")
40   private int i = 0;
41
42   @Share("alive")
43   public void increment(){i++;}
44
45   @Pure("alive")
46   public int getState(){return i;}
47 }
48
49 public class Component{
50     /*Passes message to
51       its attached connector*/
52   public void publish(EventMessage e){
53     ...}
54   ...
55 }
```

Figure 2: Code of Component C in Java

message dependencies can be merged into a complete system message dependence graph (Figure 3e). Throughout this section, we will continue using the scenario depicted in Figure 1 and Component C's implementation depicted in Figure 2.

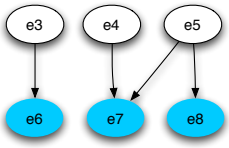
3.1 Intra-Component Dependence Graph

This section shows how an intra-component dependence graph can be created by extracting control-flow message dependencies. Helios starts the control-flow analysis of the source code by creating a program dependence graph of the

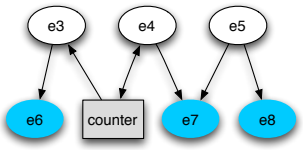


(a) Generating a component's callgraph and adding typed message sinks

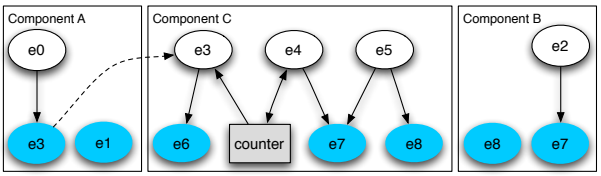
(b) Adding typed message sources



(c) Compressing paths between message sinks and message sources, by eliding methods



(d) Adding state-based dependencies



(e) Adding inter-component dependencies

Figure 3: Creating a System Message Dependence Graph

component [13]. In a program dependence graph, nodes represent statements, and directed edges represent data or control dependencies. As discussed in the previous section, to ensure that components only communicate via messages, Helios makes the following assumptions: (1) the component object uniquely owns the component's reference to the message source, (2) the component object implements the component's sink, and (3) no reference to the component object is passed to other application objects. As a consequence, no other application object can directly call the component's message sink or message source, and extracting a call graph of the component object results in all nodes of the program dependence graph that impact a component's message sink and message source. All of the systems used in our evaluation were able to satisfy these assumptions.

Helios is able to create an accurate call graph of the component object based on a class hierarchy analysis as described in [27].To extract the call graph, Helios creates a method node for every method of the component object. For each method `m`, Helios determines which other methods of the component object the method `m` is able to call. For each detected method call, Helios creates a directed edge from the node representing `m` to the node representing the called method. In addition, 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. Figure 3a shows the result of running these steps on the source code of Component C. For example, the method `consume(e4 e)` in line 25 of Figure 2 is represented by the right-most node labeled `consume` in Figure 3a. This node has an incoming edge from a message type node `e4` representing the method's parameter. Since `consume(e4 e)` calls the method `m1`, the right-most node labeled `consume` also has an outgoing edge to the method node representing `m1`.

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 (i.e., whether `m` calls `publish(...)`). In the case 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 control flow analysis to determine where the argument variable was declared. The declaration signifies the type of the outgoing message. After the message type is identified, 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. Figure 3b shows the added message nodes (depicted by shaded ellipses) that have been extracted from Component C's source code.

Helios creates the final intra-component message dependence graph by adding a directed edge for each existing path between a message sink node and a message source node. To find all paths, Helios performs a depth-first search on the graph starting on the message nodes representing message sinks (top of the graph in Figure 3b). At the end of this step, Helios elides all method nodes from the intra-component dependence graph to increase understandability of the message dependencies. The generated graph shows clearly the control-flow dependence relationships of the component. Figure 3c depicts the result of adding edges between Component C's message sink nodes and message source nodes, while eliding all method nodes.

3.2 State-Based Dependencies

A component's control-flow does not describe all potential dependencies between incoming and outgoing messages. An outgoing message might depend on a component's state that is updated by a method invoked as a result of an incoming message. In Figure 2 Component C's state is captured in the field `counterObj`, which captures the occurrences of messages of type `E4`: whenever the component receives an `E4` message, it increments the counter (Figure 2, line 26). At the same time, whenever Component C receives an `E3` message, it reads the counter. Depending on the value of the counter, Component C may publish an `E6` message. Therefore, although the message `E6` does not depend on the control-flow induced by `E4`, the shared state causes `E6` to depend on the occurrence of `E4`.

In order to check state-based dependencies we utilize Plural [4], our previously published static type system for OO programs that is based on access permissions. Access permissions enable tracking typestates (richer notions of states that are similar to statecharts) and aliasing information (objects being referenced from multiple locations). The approach is modular and has been proven sound (no false neg-

atives) for OO calculi [3]. In Plural, objects are seen as transitioning through developer-defined abstract typestates at runtime; methods perform these state transitions. While permissions allow tracking of object references to determine their typestate, permissions can also express whether a reference allows modifying or only reading access to the referenced object, which is what we are interested in for Helios. Plural supports independent state dimensions, which Helios uses to track access to individual fields in a component's state separately. Therefore, two key features of Plural are utilized in Helios: (1) programming-language annotations that specify individual fields or groups of fields as one or more independent state dimensions and (2) programming-language annotations that specify access permissions a method has to these state dimensions. These annotations must be performed manually by a developer during initial implementation or during maintenance. Our previous analysis [4] has shown that the overhead of adding annotations to applications is moderate, and our experience with the application of Plural in the context of Helios confirms that.

In Helios, we annotate each method that accesses component state with access permissions. For our example in Figure 2 and our evaluation, we utilize a Java implementation of Plural, which allows developers to specify permissions using Java 5 annotations on methods and classes. For example, in Figure 2, we annotate the `consume` methods for events E3 and E4 in Component C with access permissions. `@Pure` signifies read-access, while `@Share` signifies read-write-access. A calling method needs to own the permission that the called method requires. For example, the method `increment()` in the class `Counter` in Figure 2 requires that the caller has at least also a `@Share` permission. As a consequence, `counterObj.increment();` (Figure 2, line 26) needs also a `@Share` permission.

To track state access, Helios requires that all fields of the component object are mapped into a state dimension. Multiple related fields can be mapped into the same state dimension. Independent fields can be mapped into independent state dimensions. An example state declaration is shown in lines 3-5 of Figure 2. In this example, the dimension `counter` is declared for Component C (line 5), and the component will ensure the invariant of having a `@Share` permission on the `counterObj` field (lines 3-4). In addition to the explicitly declared state dimensions, Plural assigns to each class an implicit state called `alive`. This is the case with the class `Counter`, allowing the field `i` to be mapped into this default state (line 39). The mapping of a field into a state enables Plural to check whether all methods accessing that field have the appropriate access permissions. As a consequence, in our example the method `increment` requires a `@Share` permission because it modifies the value of the field `i`, while the method `getState` requires a `@Pure` permission because `i` remains unmodified.

The permissions annotations on the methods of the class `Counter` require that the method callers provide fitting access permissions. Since Component C is calling methods of its field `counterObj`, Component C's methods also need to be annotated with access permissions. As discussed earlier, only the method `consume(E3 e)` and `consume(E4 e)` access Component C's state. The method `consume(E3 e)` requires read access to the state `counter`. As a consequence, the method is annotated with `@Pure("counter")` (The parameter reflects the name of the accessed state). The `consume(E4 e)` needs a write access permission: `@Share("counter")`. The methods `consume(E5 e)`, `m1` and `m2` do not require access to the state `counter` and therefore do not require annotations. Methods can carry more than one annotation if they access fields from different dimensions, but we do not need this feature in our simple example.

Plural can automatically check whether the permission annotations express the needed permissions of the code. A program that passes Plural's checking analysis is called permission-checked. Proofs and detailed descriptions of the access permission tracking can be found in [3, 4].

After correctly annotating a component's state with access permissions, Helios can extract state-based intra-component dependencies. The permission annotations on the `consume` methods reveal whether an incoming message modifies a component's state, reads from the state, or is independent of it. Therefore, all state-based dependencies can be determined by only inspecting the annotations on a component's `consume` methods.

Helios extracts the state-based dependencies in two steps. First, for each state variable of the component, Helios adds a state node to the intra-component message dependence graph (depicted as a grey shaded rectangle in Figure 3d). Second, Helios checks the access permissions of the component's `consume` methods. Whenever a `consume` method requires a read-access permission (`@Pure`) to a state, Helios adds an edge from the state node to the node represening the `consume` method. Whenever a `consume` method requires a read-write-access permission (`@Share`), Helios creates a bidirectional edge between the node representing the `consume` method and the state node.

After adding these edges and state-nodes, the intra-component dependence graph includes all control-flow-based and state-based dependencies between the messages. Figure 3d shows the complete intra-component dependence graph of Component C from Figure 2. Traversing the dependencies of the depicted graph reveals the additional state-based message dependency. Specifically, by starting the traversal at the message node `e4`, we reach the message node e6 (via the state-node `counter` and message-node `e3`). The added path between the nodes `e4` and `e6` represents Component C's state-based dependency that could not be uncovered during the extraction of control-flow message dependencies.

3.3 Inter-Component Dependencies

Intra-component dependencies help to understand how a component reacts to a message and what messages a component emits. A component's reaction can either be a state change or the emission of one or more messages. While intra-component dependencies facilitate the understanding of a component, they also enable more precise inter-component dependence analyses. 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. Intra- and inter-component dependencies can be merged into a *system message dependence graph* that is able to show how changes to a component's message behavior impact other components.

Helios creates an inter-component dependence graph by matching the typed message sources of components with the typed message sinks of other components. Since the intra-component dependence analysis recovers the types of the

message sources, Helios generates an inter-component dependence graph after all intra-component dependence graphs have been generated.

Helios utilizes the structural configuration of an 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. 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. A message sink matches a message source if both share the same connector and if the type of the message sink is either the same type or a super type of the message source's type. For each found match, Helios creates a directed inter-component dependence edge from the message source to the message sink. Figure 3e shows the generated system message dependence graph of the example scenario from Figure 1. Note that there are no inter-component dependencies involving Component B since no other component generates events of type `e2`, which is the type of B's message sink.

4. EVALUATION

This section provides evidence that Helios reduces the effort required for software maintenance engineers to conduct impact analysis in event-based systems. An accurate automatic approach such as Helios can help to identify components that are independent of a particular source code change. Helios extracts accurate dependencies because its control-flow analysis is based on an accurate call graph [27] and its state-access analysis is based on our accurate access permission analysis [3]. Hence, such approaches allow an engineer to inspect fewer components. This potential effort reduction is based on the precision of the dependence analysis, i.e., the degree to which extracted inter-component and intra-component dependencies correspond to dependencies that can actually occur in the event-based system at runtime.

A maintenance engineer needs to inspect all message-based inter-component and intra-component dependencies in order to recover accurately the impact of changes in event-based systems. The overall precision of a message dependence graph depends on recovering precise edges (e.g., intra-component dependencies) and precise nodes (e.g., message source interfaces). We will show that the precision of the two best available alternative strategies for recovering a message dependence graph is significantly lower than the precision achieved by Helios. The first alternative strategy (s1) is called the *import heuristic* [12]: 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 (s2) is 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. We benchmark Helios against these alternative strategies assessing 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.

To assess the improvement in precision obtained by using Helios, we compare the increase in (p1) as compared to that yielded by (s1) and the increase in (p2) as compared to that yielded by (s2). We have conducted these comparisons on Helios-compliant event-based applications spanning four different middleware platforms. We should note that we do not assess empirically the precision of retrieving message sink interfaces because our subjects all explicitly declared their message sink interfaces.

We describe the details of the evaluation next. Section 4.1 introduces five benchmark applications on which we performed our analyses. Section 4.2 describes the experiments that helped to evaluate the precision of Helios's extracted message source interfaces, i.e., (p1). Section 4.3 describes the experiments that helped to evaluate the precision of Helios's intra-component dependence recovery, i.e., (p2).

4.1 Experimental Subjects

Table 1 gives an overview of the subject event-based platforms and applications. 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 of Helios, in Java, and their architectures have been described in prior publications [20, 21, 22, 25, 26, 28]. Initially, each application utilized subject-based filtering as its main filtering mechanism. Since event-based applications that employ the Helios approach must use type-based filtering, we converted each string-based message type into a class-based message type in Java. Even though it was conducted manually, this process was accomplished relatively easily by a single engineer; the process is fully automatable.

We now briefly overview each application. *KLAX* is a falling-tiles game. The originally version was developed by Atari Corp. We analyzed a version that was developed in the event-based C2 style [20]. In KLAX, ADT components capture the game's state, game logic components compute the next state of the game, and artist components compute abstract graphical objects that are sent to a general graphics component that is independent of KLAX. The next application is *DRADEL* [20], which is an environment that supports modeling, analysis, evolution, and implementation of C2-style architectures. DRADEL can analyze an architectural description and generate a skeleton implementation for that architecture. KLAX and DRADEL are desktop applications utilizing the c2.fw middleware platform. The *Emergency Response System* (ERS) application was developed in Java on top of the architectural middleware Prism-MW [17, 28]. The application helps to deploy and organize human resources during natural disasters. ERS is a distributed application running on multiple PDA and laptops. *Stoxx* is a stock ticker notification system developed using the middleware platform REBECA [21, 22]. Stoxx is able to monitor a stock portfolio based on the stock quotes that it receives via the Internet. As part of our evaluation, we analyzed the subsystem of Stoxx that manages database connections and monitors the minimum and maximum values of selected stock quotes.

App Name	App Type	SLOC	Comp	Msg Types	M/W Platform
KLAX	Arcade Game	4.5K	14	85	c2.fw [20]
DRADEL	Architectural Type Checker	10.8K	8	82	c2.fw [20]
ERS	Emergency Response	7.1K	11	56	Prism-MW [20]
Stoxx-Sub-system	Stock Ticker Notification	1K	4	14	REBECA [21, 22]
jms2009-PS	Standard Benchmark for JMS-Providers	18.6K	4	19	JMS [25, 26]

Table 1: Studied Applications – Experimental Subjects

The final application is *jms2009-PS*, a performance benchmark designed for JMS-based publish/subscribe application servers [25, 26]. The benchmark jms2009-PS is built on top of SPECjms2007, the first industry-standard benchmark for evaluating the performance of enterprise message-oriented middleware servers based on the Java Messaging Service (JMS).[1] The jms2009-PS benchmark application simulates a distributed supply management system consisting of four component types: Distribution Center, HeadQuarter, Supermarket and Supplier components.

We selected these subjects to cover a wide range of event-based systems and to reduce domain-specific bias. The chosen subjects have been developed for four different event-based middleware systems (see the *Middleware Platform* column of Table 1) and they cover various domains such as gaming, distributed systems, financial information systems, supply management and enterprise systems (see the *Application Type* column of Table 1).

4.2 Recovered Message Source Interfaces

In this section, we address the research question, *RQ1*: How does (p1) the precision of recovered message source interfaces in Helios improve on the precision of (s1) the import heuristic? The import heuristic assumes that a component can consume or publish any of, and only those, message types that are declared in any of the component's imported or included files. The import heuristic is a state-of-the-art programming language-based heuristic [12] and, to our knowledge, the only alternative strategy to Helios available to a maintenance engineer who needs to extract typed message source interfaces in event-based systems. All our subject applications explicitly state imported message types, which allowed us to extract typed message source interfaces using the import heuristic.

Table 2 shows the results of analyzing the subject applications using the import heuristic and Helios. Column *Component Name* lists each analyzed component. The results of the import heuristic are shown in column *Imp Typ*, which shows the number of distinct message types that the component imports. Helios recovered the number of distinct typed message sources shown in column *Msg Src* and the number of distinct typed message sinks shown in column *Msg Sink*. The column *RQ1 Red* shows by how much Helios reduces the results of the import heuristic, i.e., how many of the imported message types were actually message sink types. The remaining columns are not relevant for this research question and will be discussed in Section 4.3.

Component Name	Imp Typ	Msg Sink	Msg Src	BB Dep	C Flow	Hel Dep	RQ1 Red	RQ2 Red
KLAX								
ChuteArtist	7	3	4	12	7	8	42.9%	33.3%
ChuteADT	7	3	4	12	6	12	42.9%	0.0%
Clock	10	6	4	24	5	15	60.0%	37.5%
MatchLogic	7	2	5	10	5	10	28.6%	0.0%
NextTileLogic	5	2	3	6	4	4	40.0%	33.3%
PaletteArtist	12	5	7	35	8	23	41.7%	34.3%
PaletteADT	12	6	6	36	7	26	50.0%	27.8%
RelPosLogic	6	3	3	9	3	6	50.0%	33.3%
StatusArtist	12	5	7	35	9	18	41.7%	48.6%
StatusADT	5	3	2	6	3	5	60.0%	16.7%
StatusLogic	16	11	5	55	11	35	68.8%	36.4%
TileArtist	3	3	3	9	3	3	0.0%	66.7%
WellArtist	5	2	3	6	4	4	40.0%	33.3%
WellADT	15	8	7	56	14	56	53.3%	0.0%
DRADEL								
ArchADT	93	19	74	1406	92	1043	20.4%	25.8%
ConstrCheck	4	2	2	4	2	4	50.0%	0.0%
CodeGen	6	4	2	8	2	8	66.7%	0.0%
Parser	30	3	27	81	28	58	10.0%	28.4%
Repository	11	7	4	28	8	28	63.6%	0.0%
TypeChecker	7	4	3	12	3	12	57.1%	0.0%
TypeMismatch	10	3	7	21	7	15	30.0%	28.6%
UserPalette	22	9	13	117	24	37	40.9%	68.4%
ERS								
Clock	2	1	1	1	1	1	50.0%	0.0%
DeployAdvisor	4	2	2	4	2	3	50.0%	25.0%
Map	19	12	8	96	10	52	57.9%	45.8%
Repository	10	6	4	24	5	13	60.0%	45.8%
ResrcManager	13	8	5	40	9	36	61.5%	10.0%
ResrcMonitor	12	5	7	35	7	7	41.7%	80.0%
SimulatAgent	9	4	5	20	6	18	44.4%	10.0%
StrategyKB	13	9	4	36	4	21	69.2%	41.7%
StrategAnalyzer	5	2	3	6	3	6	40.0%	0.0%
Weather	6	3	3	9	3	7	50.0%	22.2%
WeathAnalyzer	4	2	2	4	2	2	50.0%	50.0%
Stoxx								
DBAbsLimit	7	4	5	20	5	8	28.6%	60.0%
DBPortfolioItm	6	4	3	12	5	5	50.0%	58.3%
DBRelLimit	5	3	3	9	3	4	40.0%	55.6%
QuoteMinMax	6	5	4	20	4	4	33.3%	80.0%
jms2009-PS								
DistribCenter	15	7	8	56	8	8	46.7%	85.7%
HeadQuarter	8	5	3	15	3	3	62.5%	80.0%
Supermarket	11	7	4	28	4	4	63.6%	85.7%
Supplier	7	3	4	12	4	4	42.9%	66.7%

Table 2: Analysis Results

Additionally, Figure 4 depicts a histogram based on the column *RQ1 Red* The histogram classifies the analysis result of each analyzed component and therefore indicates Helios's overall performance in the context of RQ1. From the data, we can observe that using Helios typically yields significantly more precise message sources as compared to the import heuristic. The median reduction is 50%. Only two components in our subject applications obtain a 10% or smaller improvement in precision. In one of those two com-

[1]SPECjms2007 is a trademark of the Standard Performance Evaluation Corporation (SPEC). The results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjms2007 is located at http://www.spec.org/osg/jms2007.

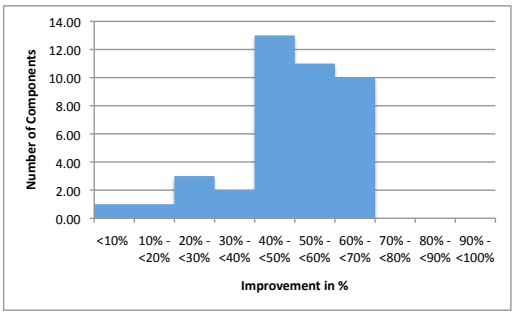


Figure 4: Helios's Improvement over Import

ponents, KLAX's *TileArtist*, Helios does not improve upon the result of the import heuristic at all. This is because the *TileArtist* only transforms the content of received messages and then forwards them. Consequently, both the import heuristic and Helios achieve the maximum precision for *Tile Artist*'s message source interfaces. The other component obtaining only a small improvement in precision from applying Helios is DRADEL's *Parser* component. This because the *Parser* mainly imports message types for use in its message source interfaces: it consumes only 3 but publishes 27 message types. Therefore, the two components for which Helios incurs no substantial improvement in precision already obtain high precision with the import heuristic. However, in the systems we have studied, this is a rare exception, as evidenced both by Table 2 and Figure 4.

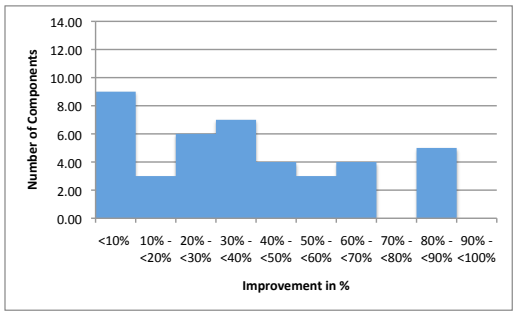
4.3 Recovered Intra-Component Dependencies

The second research question we address, *RQ2*, can be formulated as follows: Does (p2) the precision of recovered intra-component dependencies improve when using Helios as compared to the precision obtained when using (s2) the black-box approach? The black-box approach over-approximates the dependencies within a component by assuming that every message source of a component is dependent on every message sink. Without Helios, we are unaware of a better strategy to allow for the recovery of accurate intra-component dependencies.

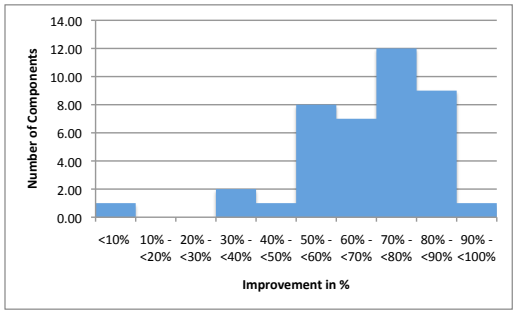
We should note that, in addition to modifying our subject applications so that they all use type-based filtering, we also had to annotate state dimensions of components using Plural annotations. This required devising a mapping between state dimensions and member variables of the class(es) that constitute a component in the manner introduced in [4].

We again refer to Table 2, focusing this time on the results of analyzing the intra-component dependencies in the subject applications using the black-box approach and Helios. The results of the black-box approach are shown in column *BB Dep*. The column *RQ2 Red* shows how much Helios increases the precision of the uncovered intra-component dependencies as compared to the black-box approach. Figure 5a depicts a histogram based on the column *RQ2 Red*.

The data demonstrates that Helios typically extracts more precise intra-component dependencies than the black-box approach. The median reduction is 33%. The smaller median as compared to RQ1 is due to components in the subject applications that have mostly state-based intra-component dependencies. Since in such components most (sometimes, all) typed message sinks can affect most (sometimes, all) typed message sources, *RQ2 Red* tends to approach 0%.



(a) Including State-Based Dependencies



(b) Only Control Flow-Based Dependencies

Figure 5: Helios's Improvement over Black-Box

The sole responsibility of a number of components in our subject applications was to manage system state. This was the case, e.g., in KLAX's *WellADT* and *ChuteADT* components, in which each incoming message starts an operation that can access and modify the game's state and all outgoing messages notify the rest of the system about state changes. Hence, every typed message sink can affect every typed message source, which is precisely what the black-box approach assumes. Therefore, both the black-box approach and Helios obtained the highest possible precision for these components.

If we set aside the state-based dependencies, Helios achieves a median dependence reduction of 71% over the black-box approach. In Table 2, column *C Flow* details the control-flow-based intra-component dependencies that Helios was able to extract. Figure 5b depicts a histogram that summarizes the reduction in control-flow-based intra-component dependencies. Even considering state-based dependencies, Table 2 indicates that for 76% of the components, Helios still reduces dependencies and, thus, increases precision by 10% or more.

5. CONCLUSION

There is a rich body of work on analyzing the dependencies and the impact of changes in traditional software systems, which rely on explicit invocations. This is not the case with event-based systems, however. The sophistication of existing program analysis techniques provides little benefit when applied to event-based systems, and engineers are left to rely on more primitive aids (such as generic lexical analysis tools used to try to uncover message declarations and usage) and/or overly conservative information (such as file import or include statements).

In this paper we presented Helios, an analysis technique that specifically targets event-based systems. Helios exploits the characteristics of such systems to uncover and combine both intra- and inter-component dependencies. While Helios requires event-based applications to satisfy certain conditions (the most important being type-based filtering of messages) and imposes some additional burden on the engineer (specifically, in annotating components for state variable access permissions), the resulting benefits are significant. Our studies have demonstrated that Helios can yield large savings in the effort required to understand and assess the impact of changes to an event-based application.

While our results to date are indicative of Helios's benefits, more empirical data with different applications can further increase the confidence in Helios's utility. We continue to actively search for such applications. However, our experience to date strongly indicates that, unlike traditional software systems and even event-based middleware platforms, of which many are freely available, event-based *applications* tend to be closely guarded by their owners. Other avenues of future work include assessing Helios's applicability to event-based applications that use filtering mechanisms beyond type-based. Our work with the applications detailed in the previous section suggests that expanding Helios to subject-based message filtering systems would be relatively easy, as message strings can be converted to programming language types. A bigger challenge will be incorporating content-based filtering systems into Helios. Finally, an interesting opportunity is presented by event-based systems that also in part rely on explicit invocation: We hypothesize that, in such systems, Helios can be used effectively in tandem with existing code analysis techniques.

6. REFERENCES

[1] J. Aldrich. Using types to enforce architectural structure. In *Working IEEE/IFIP Conference on Software Architecture*, pages 211–220, 2008.

[2] T. Apiwattanapong et al. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. 27th ICSE*, 2005.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. OOPSLA '07*, pages 301–320, 2007.

[4] K. Bierhoff et al. Practical API Protocol Checking with Access Permissions. In *Proc. ECOOP 2009*, pages 195–219. Springer, July 2009.

[5] B. Boehm. Software engineering economics. *IEEE TSE*, 10:4–21, 1984.

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

[7] C. Clifton et al. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3), May 2006.

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

[9] P. T. Eugster et al. On objects and events. In *Proc. OOPSLA '01*, pages 254–269, 2001.

[10] J. Garcia et al. Toward a catalogue of architectural bad smells. In *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, pages 146–162, 2009.

[11] S. Horwitz et al. Interprocedural slicing using dependence graphs. In *Proc. PLDI '88*, pages 35–46. ACM, 1988.

[12] B. Lague et al. An analysis framework for understanding layered softwarearchitectures. In *6th International Workshop on Program Comprehension.*, pages 37–44, 1998.

[13] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proc. 18th ICSE*, pages 495–505, 1996.

[14] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. 25th ICSE*, pages 308–318, 2003.

[15] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213 – 221, 1979-1980.

[16] J. Lions et al. Ariane 5 flight 501 failure. *Report by the Inquiry Board, Paris*, 19, 1996.

[17] S. Malek et al. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE TSE*, pages 256–272, 2005.

[18] A. Maule et al. Impact analysis of database schema changes. In *Proc. 30th ICSE*, 2008.

[19] N. Medvidovic et al. A language and environment for architecture-based softwaredevelopment and evolution. In *Proc. 21st ICSE*, pages 44–53, 1999.

[20] N. Medvidovic et al. The role of middleware in architecture-based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003.

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

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

[23] A. Orso et al. An empirical comparison of dynamic impact analysis algorithms. In *Proc. 26th ICSE*, pages 491–500, 2004.

[24] X. Ren et al. Chianti: a change impact analysis tool for Java programs. In *Proc. 27th ICSE*, pages 664–665, 2005.

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

[26] K. Sachs et al. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, 2009.

[27] V. Sundaresan et al. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

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

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

[30] S. Wagner. Using economics as basis for modelling and evaluating software quality. In *ESC'07. First International Workshop on the Economics of Software and Computation, 2007*, 2007.