Impact Analysis for Distributed 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@cs.cmu.edu

ABSTRACT

Distributed event-based (DEB) 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 a DEB 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, DEB 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 message-oriented middleware platforms. We summarize the results of several such applications. We demonstrate that Helios enables effective impact analysis and quantify its improvements over existing alternatives.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

dependence analysis, distributed event-based systems, message dependence graph, program analysis, impact analysis, components

[*]Current affiliation: Google Inc, 340 Main St, Los Angeles, CA 90291, USA, popescu@google.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS 2012 Berlin, Germany
Copyright 2012 ACM 978-1-4503-1315-5 ...$10.00.

1. INTRODUCTION

In recent years, distributed event-based (DEB) systems that have been developed using message-oriented middleware (MOM) platforms have become widespread. The market size for MOM licenses was about $1 billion in 2005 [8]; the market for all middleware licenses was nearly $16 billion in 2009, and MOM were among the fastest growing middleware platform types [5]. In DEB systems, a component does not directly call other components via explicit references, but instead *implicitly invokes* other components by *publishing messages.* Software connectors, e.g., message buses or brokers, then route these messages to the correct recipients. Consequently, components in DEB systems are highly decoupled and allow highly scalable, easy-to-evolve, concurrent, distributed, heterogeneous applications. The event-based software architectural style [33] is especially used in user-interface software and in 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 [17], software engineers need techniques that assist them in investigating the impact of proposed changes. Changing a software system is typically a time-intensive, difficult, and error-prone task. Field observations show that software engineers spent 28% of their coding time proposing changes and investigating the impact of the changes [16]. Some investigation tasks become so overwhelming that engineers give up on implementing a given change request [15]. To assist software engineers, researchers have investigated dependence-based impact analysis techniques [4, 6, 27, 34] which show how changes to a code element affect other code elements.

The impact of changes on DEB system components is more difficult to analyze because components invoke each other implicitly and therefore, by design, do not know the consumers of the messages they publish. *Lexical source model extraction (LSME)* is the only existing technique that has been used for extracting implicit invocations from a DEB system [25]. LSME allows developers to specify (1) regular expression patterns for constructs of interest and (2) actions that execute when a pattern is matched (e.g., to record information or reject some matches). Although LSME enables an engineer to quickly extract implicit invocations from source code, it typically recovers implicit invocations that are imprecise and incomplete.

Traditional dependence-based impact analysis techniques such as program slicing techniques identify overly imprecise dependencies from DEB systems since they focus on state-

ment-level dependencies that do not capture implicit invocations. In addition, experiments show that existing program slicing techniques extract dependencies that are practically incomprehensible because, for large programs, they often contain over 100,000 low-level source code dependencies [4]. To enable more comprehensible analysis, our work introduces the *message dependence graph*, which explicitly captures implicit invocations.

A message dependence graph captures two types of message 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*, a technique for determining message dependence graphs. By slightly constraining the implementation of a DEB system, while retaining its adaptability benefits, Helios creates the pre-requisites for the computation of inter- and intra-component dependencies. We call a DEB application that follows these constraints *Helios-compliant* because the application can be analyzed for event-based change impact. A DEB application becomes Helios-compliant if it follows Helios's constraints:

1. Helios-compliant applications must use MOM platforms that support a standard message sink interface and a 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 DEB 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 a 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 DEB components.

For Helios-compliant systems, Helios enables the calculation of *intra*-component dependencies through (1) component variable access specifications [3] 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. It determines *inter*-component dependencies based on the incoming and outgoing component interfaces and the system's overall structural configuration [20]. Finally, inter- and intra-component dependencies are merged to form a complete message dependence graph on which an impact analysis algorithm can be executed.

We have evaluated Helios with existing DEB applications that were written for four different MOM platforms. The evaluated systems include an architecture-based modeling and analysis environment [21], an arcade game [21], an emergency response system [33], a stock ticker notification system [23] and the jms2009-PS [30] benchmark that is based on the official SPECJms2007 [29] benchmark for evaluating the performance of enterprise MOM servers. For each system we are able to demonstrate that Helios enables impact analysis

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 a DEB system, and (3) a discussion of the choices available to an engineer in constructing a DEB 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 distributed event-based (DEB) systems, components, i.e., the units of computation and data, communicate using *messages* which carry either *notifications* or *anonymous requests* [24]. 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.

DEB components can be classified into two types, *producers* and *consumers*. A DEB 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 DEB 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.

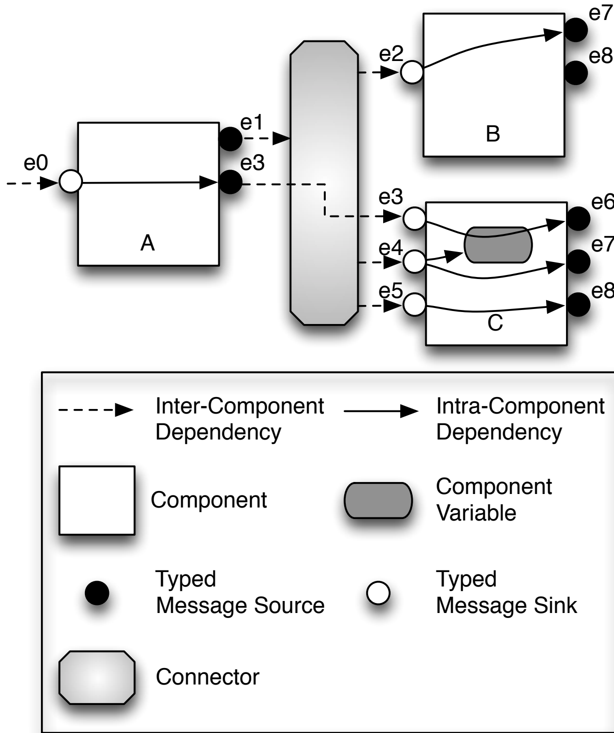


Figure 1: Inter- and Intra-Component Dependencies

2.3 Categorizing Distributed Event-Based Systems

This section contains a discussion of design choices available to developers of DEB 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 hindered by dynamically typed languages, design choices made about the message-oriented middleware platform and DEB application will facilitate or hamper the analysis of DEB systems. The three key dimensions along which DEB systems differ are the employed (1) filtering mechanisms, (2) communication styles, and (3) implementation languages.

In general, a DEB 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 [24].

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 to 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., "/Stock/Dow"). 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 code. Both interface types need to be recovered from the subject-based filters and the message names that the implementation generates.

Content-based filtering allows the most refined filters by filtering over the whole content of a message. While it is a powerful filtering mechanism, systems using content-based filtering complicate dependence analysis because the content of a message can be created in intractable ways.

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 DEB components. Moreover, applications that are based on subject-based filtering are often suitable for adaptation to type-based filtering and, therefore, can also benefit from Helios. Specifically, message names can be transformed into programming language types, and naming hierarchies lend themselves to type hierarchies. For these reasons, we focus on analyzing dependencies based on typed messages, typed message sinks, and typed message sources in Helios.

Components in DEB 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), the system's adaptability would decrease because modifications to Component C would likely affect the reference maintained by Component A. 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 MOM platforms covered in [24] 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 MOM platforms. In addition, the static typing of the languages facilitates integrating type-based message filtering with an application's implementation language. Consequently, we focus on mainstream OO programming languages in Helios. In particular, our evaluation was performed on Java-based middleware systems.

2.4 Related Work

A number of approaches for static and dynamic slicing have been developed (e.g., [4, 31, 32]). When slicing, a user has to specify a *slicing criterion* (P, V) in which P is a program point and V is a subset of program variables. A *backward slice* consists of the statements that could influence the values of variables in V at program point P. A *forward slice* consists of the statements that are affected by the slicing criterion. For *chopping*, a user tries to identify the statements that affect a given target element from a given source element [28]. Since slices are used to determine the impact of statements on other statements, slicing and chopping are dependence-based analysis techniques.

Most static slicing approaches either compute data-flow equations on a control-flow graph or compute a program dependence graph (PDG) [4]. A PDG is a directed graph in which the vertices correspond to statements and control predicates and the edges correspond to data-flow or control dependencies. The static slicing problem can be restated as a graph reachability problem on a PDG. Initial slicing approaches were not able to slice across procedure boundaries. To achieve inter-procedural slicing for data-flow-based approaches, Krinke suggested tracking the calling context of procedure calls [14]. To solve slicing across procedure boundaries utilizing a PDG, Horwitz et. al. introduced the system dependence graph (SDG) [12], which contains procedure summary edges that help to address the calling-context problem. Multiple researchers have since suggested how to extend an SDG to account for object-oriented [18] and concurrency [11] features.

None of the above *code*-based dependence analysis techniques are able to separate message dependencies from other source code dependencies. On the other end of the abstraction spectrum, there has been research in analyzing event-based *modeling* approaches. Unfortunately, these approaches share another deficiency: they provide no guidance for analyzing implementation-level event dependencies.

Stafford and Wolf [32] developed a dependence analysis technique for Rapide, a modeling language that allows one to specify and simulate the behavior of a DEB system. This approach accounts for inter- and intra-component dependencies, but fails to provide a mapping of the model to an implementation. Zhao introduced slicing for the software architecture modeling language Wright [35]. This technique is of limited use in our proposed work since Wright explicitly captures several relationships between DEB components that a programming language such as Java does not. Millett and Teitelbaum introduced slicing of Promela models [22]. However, it is unclear how Promela's channels could be mapped to event-based implementations that do not use channels, such as those of most current message-oriented middleware systems [24].

Helios is not the only program analysis technique that has been developed specifically for analyzing DEB systems. Jayaram and Eugster developed static program analysis techniques that improve the performance of DEB systems [13]. These analyses are devised specifically for the EventJava framework [9], which includes an event-based programming language and a compiler for that language.

3. APPROACH

In this section we describe how Helios creates a message dependence graph from the source code of a DEB 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 [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 DEB 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 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 describe how the intra-component message dependencies can be merged into a complete 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.

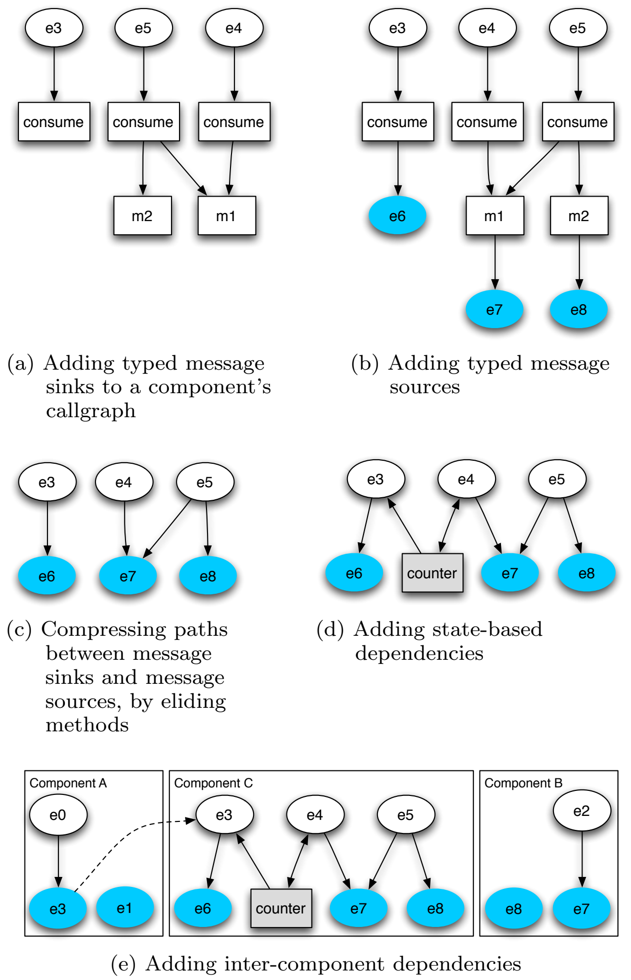
```
1   /*Component C in Java*/
2   @ClassStates({
3     @State(name = "counter",
4       inv = "share(counterObj)")})
5   @States(dim="counter")
6   public class C extends Component{
7     Counter counterObj = new Counter();
8
9     /* generated method */
10    @Share("counter")
11    public void consume(EventMessage 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

3.1 Intra-Component Dependence Graph

This section shows how an intra-component dependence graph can be created by extracting control-flow message dependencies. To compute the intra-component message dependencies, Helios analyzes and annotates the component's call graph, which captures the component's methods and its calling relationships.

For each method that implements the message sink interface, Helios analyzes the type of the method's message parameter, adds a node for the message type to the call graph and also attaches 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`.



(a) Adding typed message sinks to a component's callgraph

(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 the Final Message Dependence Graph

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 of a call to the message source, Helios (1) determines the type of the argument that is passed to the message source based on an intra-method data-flow analysis, (2) adds a message node for the type to the call graph, and (3) attaches an edge from the node representing the method `m` 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 at the message nodes representing message sinks (top of the graph in Figure 3b). At the end of this step, Helios elides from the call graph all nodes and edges that describe calling relationships between methods.The generated graph shows clearly the message-flow dependence relationships of the component. Figure 3c depicts the result of this step for Component C.

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 [3], 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 negatives) for OO calculi [2]. 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 [3] 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 [2, 3].

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 exchange 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 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 the DEB 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 DEB 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 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 maintenance engineers to conduct impact analysis in DEB systems. A semi-automatic approach such as Helios can help to identify components that are independent of a particular source code change, allowing 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 DEB system at runtime. Helios guarantees the extraction of *all* message dependencies from a Helios-compliant system because (1) all message sinks are explicitly defined, (2) its state-access analysis is based on our sound (no false negatives) access permission analysis [2], and (3) uses a sound call graph. However, Helios does not guarantee that there will be no spurious dependencies recovered (i.e., false positives are possible).

We investigate the following research question: To what extent does Helios extract fewer spurious message dependencies compared to the message dependencies that are extracted by existing techniques? We have conducted these comparisons on Helios-compliant DEB applications spanning four different MOM platforms.

We assessed the benefit of extracting intra-component dependencies by comparing Helios's analysis results with the results achieved by LSME. Recall that LSME is the only previously existing technique that has been used to explicitly extracts implicit invocations from the code of a DEB system. Since LSME does not extract intra-component dependencies, we assumed in our study that each message source was dependent on each message sink within a component. This assumption is safe because it ensures that LSME does not miss any intra-component dependencies. Although LSME typically extracts message source interfaces that are spurious and that are often incomplete, we assume the best-case scenario for LSME — that it extracts no spurious message interfaces and does not miss any actual message 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 empirical results that helped to evaluate the precision of Helios's intra-component dependence recovery.

4.1 Experimental Subjects

Table 1 gives an overview of the subject message-oriented platforms and applications. Column *Application Type* gives a short description of the application's domain; *SLOC* shows the source lines of code of each application; *Components* shows how many component objects each application has; *Message Types* totals the different message types in each application; and *MOM 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 [21, 23, 24, 30, 29, 33]. Initially, each application utilized subject-based filtering as its main filtering mechanism. Since DEB 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. 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 [3]. Even though it was conducted manually,

Application Name	Application Type	SLOC	Components	Message Types	MOM Platform
KLAX	Arcade Game	4.5K	14	85	c2.fw [21]
DRADEL	Architecture Modeling and Analysis Environment	10.8K	8	82	c2.fw [21]
ERS	Emergency Response	7.1K	11	56	Prism-MW [21]
Stoxx-Sub-system	Stock Ticker Notification	1K	4	14	REBECA [23, 24]
jms2009-PS	Standard Benchmark for JMS-Providers	18.6K	4	19	JMS [30, 29]

Table 1: Studied Applications – Experimental Subjects

this process was accomplished relatively easily by a single engineer.

We now briefly overview each application. *KLAX* is a falling-tiles game. The original version was developed by Atari Corp. We analyzed a version that was developed in the event-based C2 style [21]. 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* [21], 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 [19, 33]. The application helps to deploy and organize human resources during natural disasters. ERS is a distributed application running on multiple PDAs and laptops. *Stoxx* is a stock ticker notification system developed using the middleware platform REBECA [23, 24]. 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. The final application is *jms2009-PS*, a performance benchmark designed for JMS-based publish/subscribe application servers [30, 29]. 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 DEB applications and to reduce domain-specific bias. The chosen subjects have been developed for four different message-oriented middleware platforms (see the *MOM 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).

[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.

4.2 Evaluation Results

Table 2 provides detailed results that shows how Helios improves over LSME for each analyzed component. The two *Helios* columns show how many intra-component dependencies Helios extracts from the subject systems. The two *Improvement* columns show the extent to which Helios extracts fewer spurious intra-component message dependencies as compared to LSME. Both *Helios* and *Improvement* have *CF* and *CF+State* columns. Column *CF* captures the control-flow-based intra-component dependencies that Helios extracts from the subject systems, while column *CF+State* captures the control-flow-based and state-based intra-component dependencies extracted by Helios.

The data demonstrates that Helios reduces the number of spurious intra-component dependencies in comparison to LSME in a great majority of the cases. Figure 4 contains two histograms that help to visualize the data. The data in Figure 4a is based on Helios's extracted control-flow and state-based intra-component dependencies, while the data in Figure 4b only considers Helios's extracted control-flow-based intra-component dependencies.

Helios achieves a median reduction of 33% for control-flow-based and state-based intra-component dependencies. If we set aside state-based dependencies, Helios achieves a median dependency reduction of 71% over LSME. Figure 4 shows that components have mostly state-based intra-component dependencies in many subject systems. Since in such components most (sometimes all) message sinks can affect most (sometimes all) message sources, *Improvement: CF+State* tends to approach 0%. The sole responsibility of a number of components in our subject systems was to manage system state. This was the case, e.g., in KLAX's *WellADT* and *ChuteADT*, 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 message sink can affect every message source, which is precisely what LSME assumes. Therefore, both LSME and Helios obtained the highest possible precision for these components.

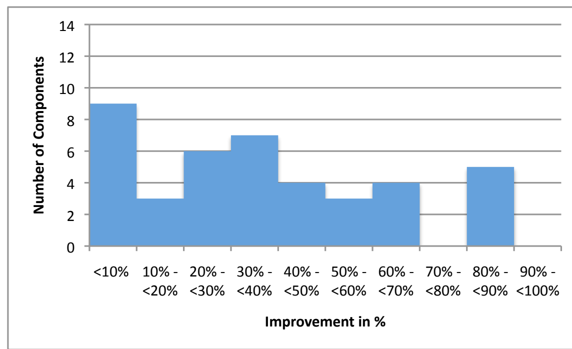
5. CONCLUSION

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 DEB systems, however. The sophistication of existing program analysis techniques provides little benefit when applied to DEB systems, and engineers are left to rely on more primitive aids such as generic lexical analysis tools used to try to uncover message declarations.

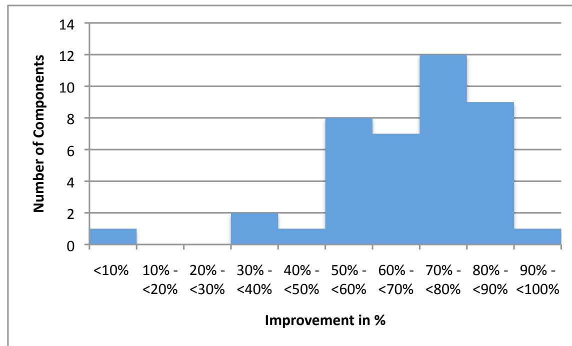
In this paper we presented Helios, an analysis technique that specifically targets DEB systems. Helios exploits the

Component	Msg Sinks	Msg Sources	LSME	Helios		Improvement	
				CF	CF+ State	CF	CF+ State
KLAX							
ChuteArtist	3	4	12	7	8	41.67%	33.33%
ChuteADT	3	4	12	6	12	50.00%	0.00%
Clock	6	4	24	5	15	79.17%	37.50%
MatchLogic	2	5	10	5	10	50.00%	0.00%
NextTileLogic	2	3	6	4	4	33.33%	33.33%
PaletteArtist	5	7	35	8	23	77.14%	34.29%
PaletteADT	6	6	36	7	26	80.56%	27.78%
RelPosLogic	3	3	9	3	6	66.67%	33.33%
StatusArtist	5	7	35	9	18	74.29%	48.57%
StatusADT	3	2	6	3	5	50.00%	16.67%
StatusLogic	11	5	55	11	35	80.00%	36.36%
TileArtist	3	3	9	3	3	66.67%	66.67%
WellArtist	2	3	6	4	4	33.33%	33.33%
WellADT	8	7	56	14	56	75.00%	0.00%
DRADEL							
ArchADT	19	74	1406	92	1043	93.46%	25.82%
ConstrCheck	2	2	4	2	4	50.00%	0.00%
CodeGen	4	2	8	2	8	75.00%	0.00%
Parser	3	27	81	28	58	65.43%	28.40%
Repository	7	4	28	8	28	71.43%	0.00%
TypeChecker	4	3	12	3	12	75.00%	0.00%
TypeMismatch	3	7	21	7	15	66.67%	28.57%
UserPalette	9	13	117	24	37	79.49%	68.38%
Stoxx-Subsystem							
DBAbsLimit	4	5	20	5	8	75.00%	60.00%
DBPortfolioItm	4	3	12	5	5	58.33%	58.33%
DBRelLimit	3	3	9	3	4	66.67%	55.56%
QuoteMinMax	5	4	20	4	4	80.00%	80.00%
jms2009-PS							
DistribCenter	7	8	56	8	8	85.71%	85.71%
Headquarter	5	3	15	3	3	80.00%	80.00%
Supermarket	7	4	28	4	4	85.71%	85.71%
Supplier	3	4	12	4	4	66.67%	66.67%
ERS							
Clock	1	1	1	1	1	0.00%	0.00%
DeployAdvisor	2	2	4	2	3	50.00%	25.00%
Map	12	8	96	10	52	89.58%	45.83%
Repository	6	4	24	5	13	79.17%	45.83%
ResrcManager	8	5	40	9	36	77.50%	10.00%
ResrcMonitor	5	7	35	7	7	80.00%	80.00%
SimulatAgent	4	5	20	6	18	70.00%	10.00%
StrategyKB	9	4	36	4	21	88.89%	41.67%
StrategAnalyzer	2	3	6	3	6	50.00%	0.00%
Weather	3	3	9	3	7	66.67%	22.22%
WeathAnalyzer	2	2	4	2	2	50.00%	50.00%

Table 2: Results of extracting intra-component dependencies



(a) Including State-Based Dependencies



(b) Only Control Flow-Based Dependencies

Figure 4: Helios's Improvement over LSME

characteristics of such systems to uncover and combine both intra- and inter-component dependencies. While Helios requires DEB 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 a DEB 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 strongly indicates that, unlike traditional software systems and even message-oriented middleware platforms, of which many are freely available, *DEB applications* tend to be closely guarded by their owners. Other avenues of future work include assessing Helios's applicability to DEB 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 DEB 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. ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under award number 1117593.

7. REFERENCES

[1] J. Aldrich. Using types to enforce architectural structure. In *IEEE/IFIP WICSA*, 2008.

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

[3] K. Bierhoff, N. Beckman, and J. Aldrich. Practical api protocol checking with access permissions. In *Proc of ECOOP*, 2009.

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

[5] F. Biscotti, T. Jones, and A. Raina. Market Share: AIM and Portal Software, Worldwide, 2009. *Gartner market research report*, April 2010.

[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*, June 2006.

[9] P. Eugster and K. Jayaram. Eventjava: An extension of java for event correlation. *ECOOP 2009–Object-Oriented Programming*, pages 570–594, 2009.

[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*, 2009.

[11] D. Giffhorn and C. Hammer. An evaluation of slicing algorithms for concurrent programs. In *Proc. SCAM*, 2007.

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

[13] K. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, pages 113–124. ACM, 2011.

[14] J. Krinke. Evaluating context-sensitive slicing and chopping. In *Proceedings. International Conference on Software Maintenance, 2002.*, pages 22–31, 2002.

[15] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ACM SIGSOFT ESEC-FSE*. ACM, 2007.

[16] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proc of ICSE*, May 2010.

[17] 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.

[18] D. Liang and M. Harrold. Slicing objects using system dependence graphs. In *Proc. ICSM*, 1998.

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

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

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

[22] L. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *Int. J. on Software Tools for Technology Transfer*, 2(4):343–349, 2000.

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

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

[25] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM TOSEM*, 5(3):262–292, 1996.

[26] D. Popescu. Impact analysis for event-based components and systems. In *Proc. of ICSE*, volume 2, 2010.

[27] V. Rajlich. A model for change propagation based on graph rewriting. In *ICSM*, page 84. Published by the IEEE Computer Society, 1997.

[28] T. Reps and G. Rosay. Precise interprocedural chopping. In *ACM SIGSOFT FSE*, 1995.

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

[30] K. Sachs et al. Benchmarking publish/subscribe-based messaging systems. In *Proc. BenchmarX*, 2010.

[31] M. Sherriff and L. Williams. Empirical software change impact analysis using singular value decomposition. In *ICST '08*, 2008.

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

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

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

[35] J. Zhao et al. Change impact analysis to support architectural evolution. *J. Software Maintenance and Evolution Research and Practice*, 14(5), 2002.