Identifying Message Flow in Distributed Event-Based Systems

Joshua Garcia, Daniel Popescu*, Gholamreza Safi,
William G.J. Halfond, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA, USA
{joshuaga,dpopescu,gsafi,halfond,neno}@usc.edu

ABSTRACT

Distributed event-based (DEB) systems contain highly-decoupled components that interact by exchanging messages. This enables flexible system composition and adaptation, but also makes DEB systems difficult to maintain. Most existing program analysis techniques to support maintenance are not well suited to DEB systems, while those that are tend to suffer from inaccuracy or make assumptions that limit their applicability. This paper presents *Eos*, a static analysis technique that identifies message information useful for maintaining a DEB system, namely, message types and message flow within a system. Eos has been evaluated on six off-the-shelf DEB systems spanning five different middleware platforms, and has exhibited excellent accuracy and efficiency. Furthermore, a case study involving a range of maintenance activities undertaken on three existing DEB systems shows that, on average, Eos enables an engineer to identify the scope and impact of required changes more accurately than existing alternatives.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.11 [**Software Architectures**]: Patterns

General Terms

Design, Experimentation

Keywords

distributed event-based systems, message flow, maintenance

1. INTRODUCTION

Distributed event-based (DEB) systems, developed using message-oriented middleware (MOM) platforms, have become widespread. In 2005, the market size for MOM licenses was about $1 billion [15]; by the end of the decade, the market for all middleware licenses was nearly $20 billion, with MOM among the fastest growing middleware platform types [11, 12]. One of the reasons for DEB systems' popularity is that they are highly decoupled, which facilitates the development of scalable, concurrent, and heterogeneous distributed applications [17, 39, 22, 16]. To achieve this low coupling, the components of a DEB system *implicitly* invoke other components by publishing messages that a connector [40] routes to the correct recipients; in turn, the recipient components may consume these messages and perform some functionality in response.

Although implicit invocation provides many benefits, it renders the *maintenance* of DEB systems labor-intensive and error-prone [24]. A major reason for this is that implicit invocation makes it difficult to determine the types of messages passed within a system and where those messages will flow at runtime. In particular, two types of programming practices lead to this difficulty: high-branching invocation statements and ambiguous interfaces. A *high-branching invocation statement* is a type of statement that may have multiple targets at runtime. Such statements often stem from mechanisms used to implement implicit invocations, such as callback functions and reflection [24]. Prior work has demonstrated that identifying the targets of high-branching invocation statements through manual inspection is challenging [24], while existing automated program analysis techniques are unable to compute that information accurately [31]. This makes it difficult for maintainers to determine the impact of a change, identify dependencies between components, and localize faults.

For DEB system maintenance, it is often necessary to know which messages can be consumed by a component. However, components in MOM-based DEB systems often rely on *ambiguous interfaces*. An ambiguous interface accepts a single, abstract message type and requires that a component internally filter and dispatch each message based on the message's attributes [19]. This complicates maintenance because examining the entry point of a message does not reveal explicit type information about the message — only that the generic message type is consumed. Instead, the maintainer must infer this information by other means, such as examining the operations performed on a consumed message.

Researchers have recognized the need for automated support of DEB system maintenance. However, existing techniques have limitations that affect their accuracy and/or applicability to MOM systems. For example, a suite of analysis techniques has focused on improving the performance of DEB systems [23]. However, these techniques rely on a set of added, specialized programming language features [18] and do not handle the ambiguous interfaces prevalent in MOM-based systems. Another technique, LSME [30], uses developer-provided regular expressions to identify message types. However, this technique neither identifies the specific attributes that

*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.
ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08 ...$15.00.

define a message type nor computes how messages may flow within a component. Our prior work, Helios [32], is a semi-automated technique that improves on LSME by using developer-provided annotations to identify message flow within a component. Like LSME, however, Helios is unable to identify the attributes that uniquely identify each message type.

To address the limitations of existing approaches, we have developed a new technique, *Eos*, that automatically identifies a DEB system's message properties, specifically, the system's message types and message flow. The key insight underlying our work is that these message properties can be identified by analyzing a DEB system's implementation and aggregating "message-revealing" information defined by statements along different possible execution paths. Our approach leverages this insight to fully automate the identification of message information via a static program analysis. Our evaluation of Eos on six existing DEB systems shows that our technique is both accurate and fast. Eos averaged over 90% precision and recall of the subject systems' message types and dependencies, and could analyze each system in about a minute. Furthermore, a case study involving a range of maintenance activities undertaken on three existing DEB systems shows that, on average, Eos enables an engineer to identify the scope and impact of required changes more accurately than existing alternatives.

The remainder of the paper is organized as follows. Section 2 discusses how modern DEB systems are typically implemented and existing challenges for identifying message flow in DEB systems. Section 3 describes how Eos addresses these challenges. Section 4 presents our evaluation results. Section 5 overviews the relevant related work. Section 6 presents our conclusions.

2. BACKGROUND AND MOTIVATION

A distributed event based (DEB) system consists of components that send and receive messages in response to events that occur in the system [29]. Components in DEB systems communicate via two types of message interfaces: the *message source* is the interface that publishes a message, while the *message sink* is the interface that consumes a message. To publish a message, a DEB component passes an instance of the MOM's message type to a message source; and to consume a message, the component provides a method whose input parameter is the MOM's message type. The specific API a component must implement or use for the sinks and sources is typically defined by the underlying MOM platform's abstract base component class. Similarly, the message types either implement or extend the MOM platform's base message type.

Figure 1 shows the partial implementation in Java of two simple DEB components, an adder (lines 1-26) and a printer (lines 28-31). Both extend the MOM's abstract base component class `Component` and provide an implementation of the message sink interface, `consume`. The parameter to `consume` is of type `Message`, which is the MOM's base message type. Component `DEBAdder` receives a message at line 4, checks whether its name attribute is "add", and, if so, calls the `add` method to perform the addition. The result of the addition is published by calling the `pub` method, which wraps a call to the MOM's message source interface, `publish`. `DEBAdder` can also receive messages whose name attribute is "resultRequest", for which it creates and then publishes an instance of the MOM's base message type that contains the previously calculated sum value ("lastResult"). The `DEBPrinter` component consumes messages whose name attribute is "lastResult".

There are three different mechanisms for defining message types in modern DEB systems: attribute-based, nominal, and subject-based [29]. Attribute-based message types, which are defined as sets of named attributes, are used in Figure 1. By convention, a message

```
1   public class DEBAdder extends Component{
2    private BigInteger sum = null;
3    private String sumStr = "sum";
4      public void consume(Message m) {
5        String nameAttr = (String)m.getAttr("name");
6        if ("add".equals(nameAttr))
7        { Message sumMsg = this.add(m);
8           this.pub(sumMsg); }
9        else if ("resultRequest".equals(nameAttr))
10       { if (sum != null)
11         { Message m2 = new Message();
12           m2.setAttribute("name", "lastResult");
13           m2.setAttribute(sumStr, sum);
14           this.pub(m2); }}}
15
16    private Message add(Message m3){
17      Integer n1 = m3.getAttr("num1");
18      Integer n2 = m3.getAttr("num2");
19      sum = n1 + n2;
20      Message m4 = new Message();
21      m4.setAttribute("name", "currentSum");
22      m4.setAttribute(sumStr, sum);
23      return m4;}
24
25    private pub(Message m){
26       publish(m);}
27
28  public class DEBPrinter extends Component{
29    public void consume(Message m){
30      String nameAttr = m.getAttr("name");
31      if ("lastResult".equals(nameAttr){...}}}
```

Figure 1: The `DEBAdder` and `DEBPrinter` components

type has a special attribute that names the message, which we refer to as the "name" attribute. Nominal message types are explicitly declared in the underlying programming language's type system. For example, a class called `Add` that inherits from the class `Message` would be a nominal message type. In subject-based message typing, each message type is defined via a pre-determined string subject field. An example is a message with a "subject" field whose value is "/Message/Add". Both nominal and subject-based message types can be represented as attribute-based types: for subject-based messages, a corresponding attribute-based type has an attribute to represent the subject field; for nominal types, a corresponding attribute-based message type has an attribute to represent the programming language class of the nominal type. Therefore, for simplicity of the explanation, we restrict our discussion in the rest of this paper to attribute-based messages. However, it is important to note that our technique is not limited to attribute-based messages. In fact, two of the MOM platforms used in our evaluation (Section 4) support nominal message typing.

The message types in a given DEB system can be divided into *published message types* (PMTs), which are the types sent via a message source interface, and *consumed message types* (CMTs), which are the types received via a message sink interface. An example PMT published at line 8 is $t_{sum} \leftarrow \{(\text{“}name\text{”},\text{“}currentSum\text{”}), (\text{“}sum\text{”},\varepsilon)\}$; an example CMT of DEBAdder is $t_{add} \leftarrow \{(\text{“}name\text{”}, \text{“}add\text{”}), (\text{“}num1\text{”},\varepsilon), (\text{“}num2\text{”},\varepsilon)\}$. Note that ε denotes an unknown value. A message type, such as $t_{last} = \{(\text{“}name\text{”},\text{“}lastResult\text{”}), (\text{“}sum\text{”},\varepsilon)\}$, can be both a PMT and a CMT since it is published by DEBAdder at line 14 and consumed by DEBPrinter at line 31.

There are two types of relationships between PMTs and CMTs: an *intra-flow dependency* represents a relationship where a PMT may be published by a given component as the result of the component's consumption of a CMT; an *inter-flow dependency* represents a relationship where a PMT may be published by one component and then consumed by another component. In DEBAdder, an intra-

flow dependency exists between t_{add} and t_{sum} since the consumption of t_{add} leads to the publication of t_{sum} at line 8. An inter-flow dependency exists between DEBAdder and DEBPrinter since t_{last}, published at line 14, can be consumed at line 31 of Figure 1.

As mentioned in Section 1, identifying the types and flow of messages is difficult due to the use of ambiguous interfaces and implicit invocation [24]. Ambiguous interfaces in DEB systems make it difficult to identify the message types that can be published or consumed at a given source or sink. For example, consider the two message interfaces shown in Figure 1, `consume(Message m)` and `publish(Message m)`. These interfaces only indicate that the consumed and published types are of the base type `Message`. However, from our inspection of the code, we know that the types are actually t_{add} and t_{last}. Similarly, implicit invocation makes it difficult to identify message flow. For example, without knowing the message types present at lines 26 and 29 of Figure 1, it would be challenging for a software maintainer to determine whether there exists an inter-flow dependency among these statements, as there is no explicit definition of the relationship in the code.

Simple and straightforward solutions are unlikely to remedy this problem or would result in crude estimates of the correct relationships. For example, a naive solution for identifying message types would be to convert them all to nominal types (e.g., `consume(AddMsg m)` and `publish(LastResult m)`). This would allow one to subsequently leverage explicit subtyping relationships among the resulting nominal types. However, even if a MOM platform were to provide such interfaces, the underlying programming languages would still be unable to support dynamic dispatch of appropriate messages to these more specific interfaces [14]. Similarly, one can simply assume that all message sources could be connected to all message sinks, but this would result in significant over-approximation of the message flows.

A key insight underlying our approach is that message types and their relationships can be identified through systematic analysis of a DEB system's components. By focusing on certain kinds of statements, which we call *message revealing* statements, we are able to identify useful information about the message attributes that make up a message type. By themselves, these statements do not give us a complete picture; but by combining and aggregating the information along control-flow paths that originate from message sources or terminate at message sinks, we can identify message types. For example, consider the CMT t_{add} at line 6. Here we can see that along the path where the condition at line 6 is true, a message object originating from the message sink is assumed to have an attribute "name" with the value "add", and then also the attributes "num1" and "num2," which are accessed at lines 17-18. Similarly, consider the PMT t_{last}, created at line 11. Along a path containing lines 11-14, the message object published at line 14 is created at line 11 and has two attributes, "name" and "sum", added on lines 12-13 before it is published. Identifying these message types also allows us to identify flows and relationships among the statements that define a message type. In Section 3, we describe how we formalize this intuition to create an automated analysis that is able to detect message types and flows in a DEB system.

3. DEB SYSTEM ANALYSIS IN EOS

Eos is a static analysis technique for identifying (1) attribute-based message types and (2) message flow in DEB systems. The input to Eos is the implementation of the DEB system and a specification of the underlying MOM's API. This specification comprises the generic message class; methods that access and modify a message or its attributes; and, for MOMs that support attribute-based message types, the name of the special attribute that denotes a message name. From our experience, all major MOM frameworks provide APIs for this functionality and the specification needs to only be performed once per MOM API. The output of Eos is the DEB system's Message Flow Graph (MFG), which shows the system's message types and the relationships between those types. The MFG is a directed graph (C,P,E), where C is the set of consumed message types (CMTs), P is the set of published message types (PMTs), and E is the set of directed edges representing intra-flow $(C \times P)$ dependencies and inter-flow $(P \times C)$ dependencies.

Eos consists of two core analyses, *findCMT*, which computes CMTs and *findPMT*, which computes PMTs. Both analyses are defined as summary-based iterative data-flow analyses that propagate message-flow information extracted from different types of message revealing statements. The analyses assume that aliases can be resolved precisely. Each analysis propagates the message-flow information using four data-flow sets [5] for each statement s in a system's implementation: $in[s]$ contains information that flows along a program path to s; $gen[s]$ contains information that is generated at s; $kill[s]$ stores information that is no longer valid because of information generated at s; and $out[s]$ stores information that flows to s's successors. All four data-flow sets are initially empty. As detailed in Sections 3.1 and 3.2, the analyses update the data-flow sets for each statement in a method until a fixed point is reached.

Both *findCMT* and *findPMT* compute a summary of each method in a DEB component. The *method summary* describes the message-flow information that can be inferred from a method. Method summaries make Eos's analyses inter-procedural, thus allowing an entire DEB application to be analyzed. Methods are analyzed in reverse topological order with respect to the DEB component's call graph so that a given method's summary is computed before any methods that call it are analyzed. Cycles in the call graph (e.g., from recursion) are handled in the standard manner, by treating the involved methods as one "super method."

In both analyses, Eos must calculate the names of attributes and the value of the special message-name attribute. To do this, Eos uses string constant propagation, which provides a precise solution since, by convention, DEB systems use constant strings to define these values. Eos does not calculate the value of other attributes, and we use ε to denote these unknown values. Eos stores attribute information in two sets, *Attr* and *TypeHier*. *Attr* is a set of pairs $(t,(name,value))$, where t is a message type and $(name,value)$ is the name and value of one of t's attributes. *TypeHier* is a set of message type pairs, (t,t'), where t' extends t by including all of t's attributes and a new attribute name-value pair.

Next, we define the analysis for identifying CMTs (Section 3.1) and describe the key steps of the analogous analysis for identifying PMTs (Section 3.2). We then discuss how the CMTs and PMTs are used to identify message-flow dependencies (Section 3.3).

3.1 Identifying Consumed Message Types

The *findCMT* analysis is shown as Algorithm 1 and computes the information needed to identify a DEB component's CMTs. The input to *findCMT* is a single method of a DEB component and its outputs are the *Attr*, *TypeHier*, and *in* sets. To compute CMTs, *findCMT* identifies and tracks three types of *Consumed Message Revealing (CMR) statements* — CMR-entry, CMR-attr, and CMR-invoke. Together, these statements identify the message types entering methods and the attributes that could be contained in those message types. Eos uses the information extracted from these statements to identify the attributes of each CMT.

CMR-entry statements are points at which messages enter a method. *findCMT* creates a reference-type pair to track each message that originates from a CMR-entry. A *reference-type pair* is an

Algorithm 1: findCMT

Input: $meth \in Methods$
Output: $Attr, TypeHier, in$

```
1  gen[entry] = {(p, t_entry) | p ∈ P ∧ typeOf(p) = Message}
2  workList ← {entry of method meth}
3  repeat
4      s ← first statement of workList
5      match s do
6          case "if(r_v.equals(r_attrVal)){...}"
7              foreach (r, attrName) ∈ getAttrInfo(r_attrVal) do
8                  handleCMRAttr(s, r, attrName, r_v)
9          case "r_attrVal=r.getAttribute(attrName);"
10             handleCMRAttr(s, r, attrName, ε)
11         case "if(r.hasAttribute(attrName)){...}"
12             handleCMRAttr(s, r, attrName, ε)
13         case "r = f(A);"
14             updateSets(gen[s], Attr, TypeHier,
               A, summary(f))
15             kill[s] ← {(r, t_i) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
16         case "r_i = r_j;"
17             gen[s] ← {(r_i, t_k) | (r_j, t_k) ∈ in[s] ∧ k ∈ Stmts}
18             kill[s] ← {(r_i, t_k) | (r_i, t_k) ∈ in[s] ∧ k ∈ Stmts}
19         case "return r_i;"
20             kill[s] ← {(r, t_j) | (r, t_j) ∈ in[s] ∧ r ≠ r_i ∧ j ∈ Stmts}
21     out[s] ← (in[s] \ kill[s]) ∪ gen[s]
22     foreach s' ∈ succ(s) do in[s'] ← in[s'] ∪ out[s]
23     foreach s' ∈ succ^f(s) do in[s'] ← in[s'] ∪ in[s]
24     workList ← workList ∪ {s' | (s' ∈
       succ(s) ∪ succ^f(s)) ∧ changed(in[s'])}
25 until workList = ∅
```

Procedure 1: *handleCMRAttr(s, r, attrName, val)*

Input: $s \in Stmts, r, attrName, val$

```
1  TypeHier ← TypeHier ∪ {(t_i, t_s) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
2  Attr ← Attr ∪ {(t_s, (attrName, val)) | (t_i, t_s) ∈ TypeHier ∧ i ∈ Stmts}
3  gen[s] ← {(r, t_s) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
4  kill[s] ← {(r, t_i) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
```

ordered pair (r, t_i) where r is a reference to a message object and t_i represents the message type assigned to that object. We use the subscript i to denote the line number at which the message type t originated. All four data-flow sets (*in*, *out*, *gen*, and *kill*) propagate reference-type pairs. At line 1 of Algorithm 1, *findCMT* creates a reference-type pair (p, t_{entry}) for each parameter p of type `Message` in method *meth*'s parameters P. These reference-type pairs are added to $gen[entry]$ so that they will be propagated to other CMR statements. For example, for the method entry at line 4 of Figure 1, $gen[entry] = \{(\mathtt{m}, t_4)\}$.

CMR-attr statements perform operations on an attribute of a message object. *findCMT* can identify attribute information by correlating these operations with the statement's *reaching reference-type pairs,* i.e., reference-type pairs that flow into the statement via its *in* set. There are three variants of CMR-attr statements: (1) `if` statements that check whether an attribute has a particular value, (2) statements that retrieve an attribute from a message object, and (3) `if` statements that check whether a message object has an attribute.

The first CMR-attr variant, handled at line 6 of Algorithm 1, allows *findCMT* to infer the value of an attribute, the name of that attribute, and the message type to which that attribute belongs. The first CMR-attr variant checks if an attribute, referenced by $r_{attrVal}$, has a predetermined value, referenced by r_v. An example of this variant appears at line 6 of Figure 1, where the predetermined value "add" is compared to the value of the attribute reference `nameAttr`. From this CMR-attr variant, *findCMT* can infer that the value of the attribute referenced by `nameAttr` is equal to "add" along one path originating from the CMR-attr statement and is not equal to "add" along the other path.

Handling the first variant of CMR-attr statements involves three steps: (1) identify the name and value of the attribute corresponding to $r_{attrVal}$ as well as the reference to the attribute's containing message object, (2) create a new message type from the identified attribute information, and (3) propagate different information along the two branches originating from the CMR-attr statement. In the first step, *findCMT* looks for definitions of $r_{attrVal}$ by traversing all of its definition-use chains. Specifically, *getAttrInfo* traverses each definition-use chain of $r_{attrVal}$ until it reaches a call of the form *r*.`getAttribute`(*attrName*), at which point it adds a pair (*r*, *attrName*) to its result set. This internal logic of *getAttrInfo* is straightforward and is thus elided from Algorithm 1. For the example at line 6 of Figure 1, the call to *getAttrInfo*(`nameAttr`) returns (`m`, "*name*").

The second step begins once *getAttrInfo* has identified all definitions of $r_{attrVal}$. *findCMT* iterates over *getAttrInfo*'s result set and calls *handleCMRAttr* (Procedure 1), which creates new message types from the identified attribute information. For the example at line 6 of Figure 1, *findCMT* calls *handleCMRAttr* with (6, `m`, "*name*", "*add*"). The $in[6]$ set contains $(\mathtt{m}, t_5)$ as the sole reaching reference-type pair. *handleCMRAttr* creates a new message type t_6 from t_5 and records this by adding (t_5, t_6) to *TypeHier*. *handleCMRAttr* also updates *Attr* with (t_6, ("*name*", "*add*") to match the newly created type with the identified attribute information. Finally, *handleCMRAttr* creates a new reference-type pair $(\mathtt{m}, t_6)$, adds $(\mathtt{m}, t_6)$ to $gen[6]$, and adds $(\mathtt{m}, t_5)$ to $kill[6]$.

In the third step, *findCMT* uses branch-sensitive transfer functions [8, 9] to propagate the new reference-type pairs along one branch of the CMR-attr statement but not along the other branch. At line 22 of Algorithm 1, *findCMT* propagates the *newly generated* reference-type pairs to the *in* set of the CMR-attr's successor statement on the true branch. So at line 6 of Figure 1, *findCMT* propagates $(\mathtt{m}, t_6)$ along the true branch to $in[7]$. At line 23 of Algorithm 1, *findCMT* propagates the CMR-attr statement's *unmodified* reaching reference-type pairs to the *in* set of its successor on the false branch. So at line 6 of Figure 1, *findCMT* propagates $(\mathtt{m}, t_5)$ along the false branch to $in[9]$.

The second variant of CMR-attr, identified at line 9 of Algorithm 1, allows *findCMT* to infer the name of an attribute and the message type to which that attribute belongs. Specifically, this CMR-attr variant retrieves the attribute's name, *attrName*, from the message object referred to by *r*. *findCMT* can infer that all message types paired with *r* in the CMR-attr statement's reaching reference-type pairs are expected to contain the retrieved attribute, *attrName*. As in the previous variant, *findCMT* calls *handleCMRAttr*; however, no attribute value information is provided in this case and the unknown value, ε, is passed to *handleCMRAttr* instead. Since this variant of CMR-attr is not contained within an `if` statement, all of the new reference-type pairs are propagated to the *in* set of *s*'s successor by line 22 of Algorithm 1 ($succ^f(s)$ is undefined for non-branching statements).

To illustrate, consider the CMR-attr statement at line 5 of Figure 1. To analyze the statement, *findCMT* calls *handleCMRAttr* with the arguments (5, `m`, "*name*", ε). The sole reaching reference-type pair for line 5 is $(\mathtt{m}, t_4)$, so *handleCMRAttr* adds (t_4, t_5) to *TypeHier*. The *Attr* set is updated with (t_5, ("*name*", ε)). Finally, $gen[5]$ is set to $\{(\mathtt{m}, t_5)\}$ and $kill[5]$ is set to $\{(\mathtt{m}, t_4)\}$. The transfer function at line 22 of Algorithm 1 propagates $(\mathtt{m}, t_5)$ to $in[6]$.

The third variant of CMR-attr, handled at line 11 of Algorithm 1, accounts for `if` statements that check whether a message object referred to by *r* has an attribute named *attrName*. From this CMR-attr

statement, *findCMT* can infer that, along one path originating from the statement, all message types associated with r in the reaching reference-type pairs contain this attribute and, along the other path, they do not. *findCMT* handles this CMR-attr statement by (1) creating, for each reaching reference-type pair that contains r, a new message type that includes the attribute (as is done with the second CMR-attr variant), and (2) propagating branch-sensitive information (as is done with the first variant).

CMR-invoke statements, handled at line 13 of Algorithm 1, are invocations of DEB component methods. At the point of each CMR-invoke, *findCMT* incorporates information from the summary of the invocation's target method. A method's summary comprises the contents of the *Attr*, *TypeHier*, and *out* sets of the method's exit point. If any of these sets include message types defined by the method's formal parameters, then that means the CMR statements in the method operate on the arguments provided by the CMR-invoke. To account for the actions within the target method, *updateSets* substitutes the message types of the CMR-invokes's arguments for the message types defined by the corresponding formal parameters in the summary. The target method's summary is then used to update the *Attr*, *TypeHier*, and *gen* sets of the statement containing the CMR-invoke.

To illustrate, consider the CMR-invoke statement at line 7 of Figure 1. The summary of the invoked method `add`, whose implementation starts on line 16 of Figure 1, is: $Attr \leftarrow \{(t_{17}, (\text{"}num1\text{"}, \varepsilon)), (t_{18}, (\text{"}num2\text{"}, \varepsilon))\}$; $TypeHier \leftarrow \{(\mathbf{t_{16}}, t_{17}), (t_{17}, t_{18})\}$; and $out[add_{exit}] \leftarrow \emptyset$. t_{16} is highlighted because it is the message type referred to by `add`'s formal parameter `m3`, and will be substituted with a message type from an argument at a call site. In this case, the argument provided by the CMR-invoke, `m`, has a message type of t_6, which will be substituted for t_{16}. *updateSets* performs this substitution, sets $gen[7]$ to $out[add_{exit}]$, and adds the summary's *Attr* and *TypeHier* sets to the corresponding sets in *findCMT*.

The last two case blocks in Algorithm 1 handle assignment and return statements. Line 16 handles assignment statements by updating the statement's *gen* and *kill* sets so that message reference r_i points to all the message types pointed to by r_j and the reaching reference-type pairs involving r_i are no longer propagated. Line 19 of Algorithm 1 handles return statements by updating the *kill* set of the statement so that only the reaching reference-type pairs referenced by r_i are allowed to propagate beyond the return statement.

findCMT analyzes each method in the DEB component once and the analysis of each method terminates when the worklist is empty. Since *findCMT* only adds items to the worklist when an *in* set changes, termination occurs when all *in* sets have reached a fixed point. The *in* sets will reach a fixed point because there is a finite upper bound on each *in* set—the set of all reference-type pairs defined in the method—and each iteration of the algorithm causes the *in* set to monotonically grow with new reference-type pairs. In general, iterative data flow analysis is $O(n^2)$, but with the optimal statement traversal it can be $O(cn)$ where c is the maximum loop nesting depth in the method's control flow graph. The DU chains used by *getAttrInfo* can be pre-computed for each method in $O(n^2)$ using standard reaching definition [5].

After *findCMT* analyzes all methods of a DEB component, Eos identifies the attributes of each CMT by using the information contained in the *Attr* and *TypeHier* sets. By definition, a CMT originates from a message sink interface. Therefore, Eos identifies the message type t_{sink} defined at the message sink, extracts each sequence of pairs in *TypeHier* that extend t_{sink}, and finds their corresponding attributes in *Attr*. For example, consider the type t_{18}. The set of pairs in *TypeHier* that extends t_{18} is $\{(t_4, t_5), (t_5, t_6), (t_6, t_{17}), (t_{17}, t_{18})\}$, where $t_4 = t_{sink}$. For these message types, the relevant set of pairs in *Attr* is $\{(t_5, (\text{"}name\text{"}, \varepsilon)), (t_6, (\text{"}name\text{"}, \text{"}add\text{"})), (t_{17}, (\text{"}num1\text{"}, \varepsilon)), (t_{18}, (\text{"}num2\text{"}, \varepsilon))\}$. t_4 has no attributes. Thus, the attributes of t_{18} extracted by Eos are $\{(\text{"}name\text{"}, \text{"}add\text{"}), (\text{"}num1\text{"}, \varepsilon), (\text{"}num2\text{"}, \varepsilon)\}$. Note that, for a given type t' created from t, the value of each attribute of t' (e.g., ("*name*", "*add*")) takes precedence over prior values of the same attribute (in this case, ("*name*", ε)).

Algorithm 2: findPMT

```
   Input: meth ∈ Methods, Attr, TypeHier, in_c
   Output: Attr, PubTypes, TypeHier
 1 workList ← {entry of method meth}
 2 gen[entry] = {(p, t_entry) | p ∈ P ∧ typeOf(p) = Message}
 3 repeat
 4   s ← first statement of workList
 5   in[s] = ⋃_{p∈pred(s)} out[p]
 6   match s do
 7     case "r = createMessage();"
 8       gen[s] ← {(r, t_s)}
 9       kill[s] ← {(r, t_i) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
10     case "r.setAttribute(attrName, val);"
11       TypeHier ← TypeHier ∪ {(t_s, t_i) | i ∈ Stmts ∧ (r,
         t_i) ∈ in_c[s] ∪ in[s]}
12       Attr ← Attr ∪ {(t_s, (attrName, val)) | (t_s, t_i) ∈ TypeHier}
13       gen[s] ← {(r, t_s) | (r, t_i) ∈ in[s] ∪ in_c[s] ∧ i ∈ Stmts}
14       kill[s] ← {(r, t_i) | (r, t_i) ∈ in[s] ∪ in_c[s] ∧ i ∈ Stmts}
15     case "publish(r);"
16       PubTypes ← PubTypes ∪ {(t_i, s) | i ∈ Stmts ∧ (r,
         t_i) ∈ in[s] ∪ in_c[s]}
17     case "r = f(A);"
18       updateSets(gen[s], Attr, TypeHier, PubTypes, A,
         summary(f))
19       kill[s] ← {(r, t_i) | (r, t_i) ∈ in[s] ∧ i ∈ Stmts}
20     case "r_i = r_j;"
21       gen[s] ← {(r_i, t_k) | (r_j, t_k) ∈ in[s] ∧ k ∈ Stmts}
22       kill[s] ← {(r_i, t_k) | (r_i, t_k) ∈ in[s] ∧ k ∈ Stmts}
23     case "return r_i;"
24       kill[s] ← {(r_j, t_i) | (r_j, t_i) ∈ in[s] ∧ r_i ≠ r_j ∧ i ∈ Stmts}
25
26   out[s] ← (in[s] \ kill[s]) ∪ gen[s]
27   workList ← workList ∪ {s' | s' ∈ succ(s) ∧ changed(out[s])}
28 until workList = ∅
```

3.2 Identifying Published Message Types

The *findPMT* analysis is shown in Algorithm 2 and computes the information needed to identify a DEB component's PMTs. The algorithm takes as its input a single method of a DEB component and the *Attr*, *TypeHier*, and *in* sets computed by *findCMT*; for clarity, in this section we denote the latter as in_c. *findPMT* utilizes the in_c set in addition to its own *in* set since a CMT can be published after it is consumed. The outputs of *findPMT* are updated versions of the *Attr* and *TypeHier* sets and a new set called *PubTypes*. *PubTypes* is a set of pairs (t_i, l), where t_i is a PMT that originated on line i of the program, while l denotes the line where the type is published.

To compute PMTs, *findPMT* identifies and tracks four types of *Published Message Revealing (PMR) statements*: PMR-create, PMR-attr, PMR-publish, and PMR-invoke. Together, these statements identify the message types created in methods, the attributes that could be contained in those message types, and which of these types are published. In the remainder of this section, we elaborate on the four types of PMR statements.

PMR-create statements initialize a reference to an object of type `Message`. There are two variants of *PMR-create*: statements that explicitly create a new instance of `Message` via a method (e.g., `new` instructions or factory methods), and statements that implicitly create such instances via the list of formal parameters to a method.

An example of the first PMR-create variant is shown on line 20 of Figure 1. The `add` method's declaration on line 16 of Figure 1 is an example of the second PMR-create variant. Algorithm 2 tracks references to message objects identified at PMR-create statements to determine when new attributes are added to the message objects via PMR-attr statements or when they are published via PMR-publish statements. The two variants are handled at lines 7 and 2 of *findPMT*, respectively.

A *PMR-attr* statement adds an attribute to a message object and is handled at lines 10–14 of Algorithm 2. These statements are analogous to CMR-attr statements, and allow *findPMT* to infer the set of attributes that are added to a message type by extending each of the PMR-attr's reaching reference-type pairs with the newly identified attribute. To illustrate, consider the *PMR-attr* statement at line 21 of Figure 1. At that line, *findPMT* determines that any messages type referred to by message object `m4` has the attribute $(\text{“}\textit{name}\text{”}, \text{“}\textit{currentSum}\text{”})$.

PMR-publish statements allow *findPMT* to infer which message types are actually published. *PMR-publish* statements are handled at line 15 of *findPMT*. All message types in the PMR-publish statement's reaching reference-type pairs that correspond to the published reference (r) are PMTs. These PMTs are added to the the *PubTypes* set. For example, line 26 of Figure 1 is a *PMR-publish* statement. *findPMT* determines that any message types referenced by `m` are PMTs and are added to the *PubTypes* set.

PMR-invoke statements, handled at line 17 of *findPMT*, are invocations of DEB component methods. These are handled in a similar way to CMR-invoke statements. The only difference is that the method summaries also include the *PubTypes* set, which is handled in the same way as the *Attr*, *TypeHier*, and *out* sets.

To illustrate, consider the *PMR-invoke* statement at line 7 of Figure 1. The summary of method `add` allows *findPMT* to infer that the message type referred to by `sumMsg` at line 7 of Figure 1 has the attributes $(\text{“}\textit{name}\text{”}, \text{“}\textit{currentSum}\text{”})$ and $(\text{“}\textit{sum}\text{”}, \varepsilon)$. Line 8 of Figure 1 shows a PMR-invoke statement where the message types in the summary depend on the arguments at the call site of an invoked method. This statement calls the method `pub`, whose summary indicates that message types referred to by `m` are PMTs. *updateSets* identifies `sumMsg` as the argument corresponding to the formal parameter `m` of `pub` and adds any message types referred to by `sumMsg` to the *PubTypes* set, which indicates that those types are PMTs.

The last two case blocks in Algorithm 2 handle assignment and return statements in the same way as *findCMT*. Line 20 ensures that reference-type pairs are propagated properly when references are copied. Line 23 ensures that only the reaching reference-type pairs relevant to the returned reference flow out of it.

The runtime and termination conditions of *findPMT* are analogous to *findCMT*. After *findPMT* completes the processing of all methods of a DEB component, the *PubTypes* set contains reference-type pairs that can be used to compute the PMTs. This is done using the attribute information from the *Attr* and *TypeHier* sets in the same manner as described in Section 3.1.

3.3 Identifying Message Dependencies

Eos identifies intra-flow dependencies by combining the consumed and published message type information for a DEB component. An intra-flow dependency exists when a message type may be published by a given component as a result of having consumed another type. More precisely, when a set of CMTs, T_C, flows into a PMR-publish statement that publishes the set of message types T_P, then the intra-flow dependencies for that PMR-publish statement comprise all edges in the set $T_C \times T_P$. The identification process for T_C and T_P is as follows. For each PMR-publish statement at line l of the DEB system's implementation, Eos identifies all of the pairs (t_i, l) in *PubTypes*. The set of all message types t_i published at l is T_P. Eos then examines the $in_c[l]$ set for the statement at l to identify all message types T_C that flow into the corresponding statement as part of its reaching reference-type pairs.

To illustrate, consider the PMR-publish at line 8 of Figure 1, which publishes message type t_{22}. The $in_c[22]$ set has the reference-type pair $(\texttt{m3}, t_{18})$, which indicates that t_{18} flows to the statement where t_{22} is published. Therefore, Eos creates the intra-flow dependency (t_{18}, t_{22}). Other intra-flow dependencies for the DEBAdder component in Figure 1 are computed in an analogous manner. The resulting dependency chains allow Eos to transitively establish the dependency between $(\texttt{m}, t_4)$ and $(\texttt{m}, t_{25})$, i.e., to relate the message referenced by `m` that is consumed at DEBAdder's sink on line 4 with the message referenced by `m` that is published by DEBAdder's source on line 26.

Eos creates inter-flow dependencies by matching PMTs of a component with CMTs of another component. For a given PMT p and CMT c, an inter-flow dependency from p to c is created iff $attrib(p) \supseteq attrib(c)$, where *attrib* is a function that returns the set of attribute names of a message type as well as the value of the special message-name attribute. This relationship ensures that a component will not attempt to access attributes that are not present in a message it receives [39]. Eos computes the inter-flow dependencies by comparing all published and consumed messages to determine whether this relationship holds. For the example in Figure 1, the only inter-flow dependency is (t_{13}, t_{31}), where $t_{13} = \{(\text{“}\textit{name}\text{”}, \text{“}\textit{lastResult}\text{”}), (\text{“}\textit{sum}\text{”}, \varepsilon)\}$ and $t_{31} = \{(\text{“}\textit{name}\text{”}, \text{“}\textit{lastResult}\text{”})\}$.

Note that Eos considers only the names, and not the types, of the published (p) and consumed (c) messages' attributes. An obvious alternative is to consider the attributes' types as well: an attribute that has been set in a component that publishes p can only be properly processed by a component that consumes c if the attribute's type in p (e.g,. integer) is the same as or a subtype of the attribute's type in c (e.g., float). This alternative is more conservative and avoids recording potentially spurious dependencies in the case of attributes that have identical names but unrelated types. Eos can be extended to provide this additional attribute-type analysis. However, we opted for the name-only alternative, because it can aid developers in system verification: if Eos does not record dependencies between components that are intended to interact but whose corresponding attributes have unintentionally been assigned different types, developers will not be alerted that such dependencies, in fact, exist and will have to localize any resulting runtime errors using some other means.

4. EVALUATION

In this section we present the results of an empirical evaluation of Eos. In the evaluation we measure the accuracy of Eos in determining *message properties*, i.e., CMTs, PMTs, and message dependencies; its usefulness for *change impact analysis*, a common software maintenance task; and its *execution time*. Specifically, we investigate the following three research questions:

- **RQ1**: How accurate is Eos in identifying the message properties of MOM-based systems?
- **RQ2**: Does Eos improve the effectiveness of change impact analysis as compared to other approaches?
- **RQ3**: What is Eos's execution time?

4.1 Subject Systems and Implementation

Table 1 provides an overview of our six subject systems, five of which have been described or used in prior publications [26, 25, 28,

Table 1: DEB Systems Used in the Evaluation

App Name	App Type	SLOC	Comps	Msg Type	MOM
KLAX	Arcade Game	4.5K	14	attr-based	c2.fw [26]
DRADEL	Architectural Analysis	10.8K	8	attr-based	c2.fw [26]
ERS	Emergency Response	7.1K	11	attr-based	Prism-MW [25]
Stoxx	Stock Ticker Notification	6.2K	4	nominal	REBECA [28, 29]
jms2009-PS	JMS Bench-mark	18.6K	4	attr-based	JMS [37, 36]
Spark [4]	Chat Client	85K	59	nominal	Smack [3]

29, 37, 36, 40]. Column *App Type* notes the application domain; *SLOC* shows the source-lines-of-code; *Comps* shows the number of DEB components in each system; *Msg Type* indicates whether a system relies on nominal, subject-based, or attribute-based message types; and *MOM* specifies each system's underlying MOM platform. All of the subject systems are written in Java and together make use of five different MOM platforms. The application domains span gaming, distributed systems, financial information systems, supply management, chat clients, and enterprise systems.

The Eos algorithms are implemented in Java and Scala. The implementations leverage the Soot [42] program analysis library to generate call graphs and control-flow graphs. We use Soot's built-in class-hierarchy analysis (CHA) to resolve aliases. The Eos implementation is available online [1]. The evaluation was performed on a system running Windows 7 Professional with a quad-core i7 2.80GHz processor and 8GB of memory.

4.2 RQ1: Accuracy of Eos

To address the first research question, we determined the accuracy of the set of message properties identified by Eos. We omit inter-flow dependencies since they are directly dependent on the accuracy of the other identified message properties. The results of Eos's analysis were compared against the "ground truth" results to calculate Eos's precision and recall. We did not compare against LSME because it is unable to report message types of attribute-based systems or identify intra-flow dependencies for any MOM-based systems.

To determine the "ground truth" for the subject systems, three graduate students manually analyzed the source code of each subject system and identified all message types and intra-flow dependencies. The results of this manual inspection are available in [1]. In order to perform the requisite comparisons, we defined two notions of equality, one for message types and another for intra-flow dependencies.

For the purpose of our evaluation, a message type extracted by Eos, t_{Eos}, can be classified as matching or spurious. t_{Eos} is matching if and only if there exists a message type in the ground truth, t_{gt}, where $attrib(t_{Eos}) = attrib(t_{gt})$. t_{Eos} is spurious if it does not match any t_{gt}. All t_{gt} for which there is no matching t_{Eos} are classified as missing. Similarly, an intra-flow dependency extracted by Eos, (t_{EosSrc}, t_{EosTgt}), matches a ground truth intra-flow dependency, (t_{gtSrc}, t_{gtTgt}), if and only if $attrib(t_{EosSrc}) = attrib(t_{gtSrc}) \wedge attrib(t_{EosTgt}) = attrib(t_{gtTgt})$. Any intra-flow dependency extracted by Eos that does not match a ground truth intra-flow dependency is considered spurious. Any ground truth intra-flow dependency that is not matched by a dependency extracted by Eos is considered missing.

Note that these definitions cause certain errors in the analysis to count twice against Eos's accuracy. For example, if a message type in the ground truth is $t_{gt1} = \{(\text{"name"}, \text{"n"}), (\text{"a1"}, \varepsilon), (\text{"a2"}, \varepsilon)\}$, but Eos only extracts the message type $t_{Eos1} = \{(\text{"name"}, \text{"n"}), (\text{"a1"}, \varepsilon)\}$, we count t_{Eos1} as a spurious message type; furthermore, if no other message type extracted by Eos matches t_{gt1}, t_{gt1} counts as missing.

Table 2 shows the precision (*PR*) and recall (*RE*) results exhibited by Eos for each subject system's CMTs, PMTs, and intra-flow dependencies. The table also depicts how many message types and intra-flow dependencies extracted by Eos were considered matching (true-positive, *TP*), spurious (false-positive, *FP*), and missing (false-negative, *FN*). The results reported in Table 2 were manually verified by the authors.

Overall, the results indicate that Eos is highly accurate. For two of the subject systems—Stoxx and Spark—the precision and recall are 100% across all message properties. For two subjects—DRADEL and ERS—the results are almost perfect, varying between 95% and 100% across all message properties. For the remaining two systems—KLAX and jms2009-PS—the precision and recall are somewhat lower: they vary from 73% to 100% across the different message properties. Next, we discuss the reasons why Eos did not perform as well on these systems.

There are three reasons for the decrease in Eos's precision on several of the subjects. First, Eos's string constant propagation is defined as a conservative analysis that propagates constants across all paths in the component regardless of whether they are feasible. This means that the identification of a string variable's values is safe, but could include "extra" values that flow to it over infeasible paths. A path-sensitive string propagation could resolve this issue, but would require an expensive per-path analysis, which could affect the scalability of our approach.

Second, some KLAX components contain a `Map` object that stores all attributes of a message in a single variable. Whenever one of these components publishes a message, it does not add individual attributes (e.g., via `m.setAttribute(attrName)`); instead, it adds all attributes to the message at once (e.g., via `m.setAllAttributes(Map)`). This feature appears to have been introduced as a programming shortcut targeted specifically at KLAX because some of its components must share copies of large data structures. DRADEL, the other application implemented on the same MOM platform (c2.fw [26]), does not use this feature and its results were not affected by the feature.

Third, some attributes are conditionally added or extracted from a message. A small number of instances of these "conditional attributes" appeared in the two c2.fw applications, DRADEL and KLAX. Eos can be extended to handle such conditional statements. However, this could impact Eos's scalability because, as the number of variables in a condition increases linearly, the combination of values those variables can take increases exponentially.

There are two reasons for the decrease in Eos's recall for several of the subjects. First, similar to the case discussed above, Eos misses some CMTs in those KLAX components that return all message attributes in a single `Map` object. Second, two KLAX components also extract attributes from a `Map` object's clone. Since Eos does not track flow within container classes, such as `Map` objects, it does not extract CMTs for these two components. This situation also occurred in jms2009-PS. These resulting missing message types led to missing intra-flow dependencies.

4.3 RQ2: Effectiveness for Maintenance

In the second research question, we address the effectiveness of Eos for improving software maintenance. To measure this, we incorporated Eos into a common software maintenance technique, change impact analysis, and compared the accuracy of the impact analysis against two baselines, one representing a "naïve" approach and the second built around LSME. A *change impact analysis* iden-

Table 2: Results for Message Types, Intra-flow Dependencies, and Execution Time

	Consumed Message Types					Published Message Types					Intra-flow Dependencies					Time (ms)	
Systems	TP	FP	FN	PR	RE	TP	FP	FN	PR	RE	TP	FP	FN	PR	RE	C	S
KLAX	68	8	7	89.47%	90.67%	71	17	15	80.68%	82.56%	95	35	18	73.08%	84.07%	12	168
DRADEL	54	0	0	100.00%	100.00%	71	0	1	100.00%	98.61%	116	2	0	98.31%	100.00%	87	696
ERS	71	4	0	94.67%	100.00%	57	0	0	100.00%	100.00%	86	4	0	95.56%	100.00%	33	363
Stoxx	36	0	0	100.00%	100.00%	33	0	0	100.00%	100.00%	42	0	0	100.00%	100.00%	36	144
jms2009-PS	20	2	0	90.91%	100.00%	31	0	2	100.00%	93.94%	31	2	2	93.94%	93.94%	44	176
Spark	22	0	0	100.00%	100.00%	19	0	0	100.00%	100.00%	29	0	0	100.00%	100.00%	30	1770

tifies the set of entities in a system's implementation (typically statements) that can be affected by a specific change made to the implementation [6, 13]. Change impact analysis is widely used in tasks such as regression testing and bug fixing. In our case study, we focused on impact analysis because the use of implicit invocation and ambiguous interfaces in DEB systems makes it difficult to determine the scope and impact of changes made during maintenance [19]. All three techniques—based on Eos, LSME, and the naïve approach—were used to identify the change impact sets for a group of common DEB system maintenance tasks. The impact sets were compared against each task's "ground truth," to compute each approach's precision and recall.

4.3.1 Case Study Setup

For the case study, we created a representative set of maintenance tasks for three systems: KLAX, jms2009-PS, and Stoxx. We selected these systems because they provide coverage for both message types encountered in our subject systems (attribute-based and nominal), and span three different underlying MOM platforms. Furthermore, Eos's analysis of the three systems exhibited different accuracy, from Stoxx's perfect precision and recall to varying inaccuracies in the cases of KLAX and jms2009-PS (recall Section 4.2). The tasks represent different kinds of maintenance operations, such as removing or changing (1) program statements, (2) message dependencies, and (3) message types. To compute the "ground truth" impact sets, two graduate students analyzed the systems' source code and documentation to determine the exact set of message dependencies and statements that would be affected by each of the maintenance tasks.

As mentioned above, we implemented three versions of the impact analysis, the first based on Eos, the second on LSME, and the third on a naïve approach. For all three, we tried to emulate the typical operations a developer might perform in using the message properties identified by the approaches. The Eos and LSME approaches each identify two impact sets, the first comprising message types and dependencies, and the second comprising program statements. The naïve approach only computes an impact set of program statements.

The general approach of the Eos-based impact analysis is to first generate the MFG by running Eos on a subject system. Then keywords and text strings from the maintenance task are used to identify an initial impact set of message types in the MFG. For example, a maintenance task for jms2009-PS includes a description involving the removal of "golden tickets" from the system. Thus, we searched for the keyword "golden" over the MFG produced by Eos and identified a message type, t_{gold}, with an attribute "goldenTicket". Using this message type and its containing component as a starting point, we then performed a reachability analysis on the MFG to identify all dependent message types and intra-flow dependencies, and also computed the program statements that corresponded to them.

The LSME approach is similar, but we make one addition to the technique to compensate for the fact that LSME does not identify intra-flow dependencies: we assume that there is an intra-flow dependency from each CMT to each PMT that is identified by LSME. This addition allows the approach to build an MFG that can be used for impact analysis. Without this addition, it would not have been possible to use LSME for impact analysis.

The naïve approach uses keywords from the description of a maintenance task to identify program statements that may be modified to complete the task. For example, if a maintenance task description includes the clause "increase the amount by which the score increments," then any program statements that contain text matching the keywords "increase," "amount," "score," or "increment" may be included as part of the impact set. Since it is not possible to use this information to build an MFG, define message types, or identify dependencies, we could only use statement-based impact sets generated by the naïve approach.

4.3.2 Discussion of Results

The results of the case study are shown in Figures 2 (message-level) and 3 (statement-level). Each of the maintenance tasks is identified along the x axis (AVG depicts the average), while the y axis shows the precision and recall achieved by the three approaches in comparison to the ground truth. As noted above, the naïve approach does not compute impact sets based on message dependencies, so Figure 2 omits this approach. Also, for maintenance tasks involving KLAX and jms2009-PS, the LSME-based approach could not calculate a meaningful impact set. The reason for this is that LSME is unable to identify the attributes of a message in an attribute-based system. Therefore, we could only apply LSME on Stoxx, which uses nominal message types.

For precision, the figures show that the Eos-based approach significantly outperforms the LSME-based approach and the naïve approach. At the message-level (Figure 2), the Eos-based approach achieved an average precision of 34% as opposed to 11% for LSME. As mentioned above, for six of the tasks involving attribute-based messages, LSME's precision was zero. At the statement-level (Figure 3), Eos achieved an average precision of 52% as opposed to 19% for LSME and 16% for the naïve approach. Eos's precision is superior to the naïve approach for all tasks, except for klax1, which we explain below.

For recall, the figures show that, on average, Eos again significantly outperforms LSME and and the naïve approach. For message-level impact sets, Eos achieved an average recall of 78% as opposed to LSME's 31%. For statement-level impact sets, Eos achieved an average recall of 88% as opposed to 26% for LSME, and 32% for the naïve approach. Note that, for the tasks for which LSME was able to provide an impact set, its recall was identical to the Eos-based approach. Although this seems positive for LSME, the primary reason behind this is that we assumed an intra-flow dependency between each pair of CMTs and PMTs identified by LSME

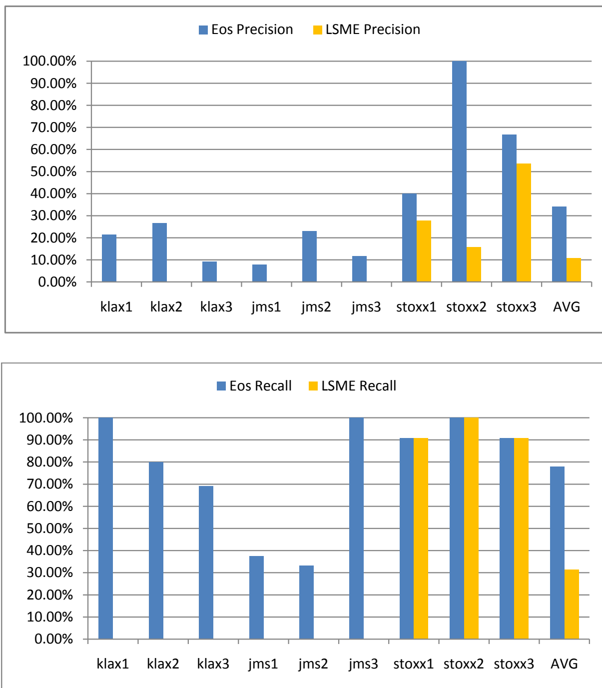

Figure 2: Impact Analysis Results for Messages

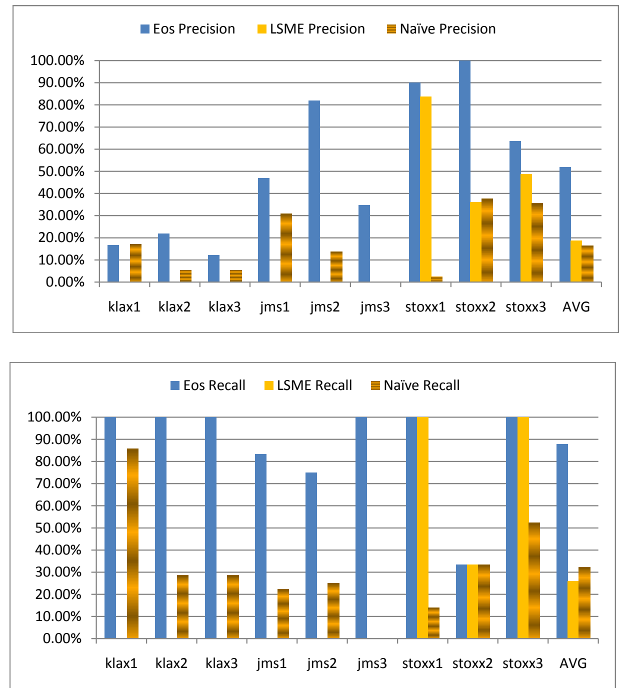

Figure 3: Impact Analysis Results for Program Statements

(see Section 4.3.1); LSME is unable to identify these intra-flow dependencies on its own.

Our results show that Eos obtains higher precision and recall than the other approaches. For the software maintainer, this is a notable benefit. By utilizing Eos for impact analysis, it is almost always possible to significantly reduce the number of spurious statements and message properties that must be inspected after a change. Additionally, the Eos-based approach ensures that a maintainer will be able to find at least those impacted statements that the other approaches can find. Overall, these results provide a strong indication that Eos can be used to increase the effectiveness of a common software maintenance technique.

The lone exception in our case study to the improved precision results is klax1, where Eos has slightly lower precision (16.67%) than the naïve approach (17.14%). This occurs because our Eos-based impact analysis finds all types transitively dependent on the initially identified impact set. In the case of this task, the additional dependencies do not need to be modified. Specifically, klax1 only requires modifications to dependencies within two components: StatusComponent and StatusArtist. However, traversing the relevant dependencies involving StatusComponent and StatusArtist in the MFG returns dependencies on the TileArtist, TileRectArtist, and LetterTileArtist components. By chance, the keywords used in the naïve approach do not match any text in the code of those three components, so the naïve approach does not return any results from them.

This is an instance of a larger issue. Namely, the precision of both the Eos-based and LSME-based approaches is lower overall than would be achieved with more sophisticated impact analysis techniques. Our impact analysis approach is somewhat simplistic and uses reachability analysis to return all possible dependencies in the MFG. Similarly, in some cases recall could be improved by a more rigorous way of identifying the initial impact set, as a keyword search may not identify all the components needed to obtain an accurate impact set. For example, in the case of klax2, the keyword "pause" in the description of the maintenance task indicated a modification to the game's pausing functionality. However, searching for this keyword did not return any relevant message types because the relevant functionality is in a component called Clock and involves the words "suspend" and "clock" rather than "pause." A more sophisticated impact analysis could mitigate this issue, but the development of such a technique is beyond the scope of this paper and will be addressed in future work.

4.4 RQ3 - Execution Time

To address the third research question, we measured the execution time of Eos. In our study we recorded the time needed to analyze each component of each system, and also the time needed to analyze the entire system. The results of this study are shown in the two right-most columns of Table 2. For each subject system, the table shows the average analysis time per component (C) and the total analysis for the entire system (S). All time measurements are shown in milliseconds. We excluded from both of these measurements the time used by Soot to generate the call graphs and control flow graphs. On average, the Soot analysis took about 65 seconds per system. As Table 2 shows, Eos took from 144ms to 1,770ms to analyze the systems, with an average of 550ms. If we include the Soot analysis time, then, on average, Eos took less than 66 seconds to analyze each system. Given this average running time, we conclude that the execution time of Eos is reasonable and the analysis is sufficiently fast to be used in practice.

5. RELATED WORK

Eos and the Message Flow Graphs (MFGs) it produces are similar to certain code-based dependency analysis techniques and the dependence graphs they produce. In particular, they both allow the navigation of dependencies between software entities. Program slicing [43] techniques compute sets of program statements that are related through control or data dependencies [10, 41, 44]. These techniques tend not to scale because they produce output that is too large to be used for other analyses, such as impact analysis. Furthermore, such techniques are unable to recover precise message

dependencies from DEB systems. In particular, they do not provide a mechanism to capture the notions of message types, sources, and sinks.

On the other end of the abstraction spectrum is the research on analyzing event-based *modeling* approaches. Stafford and Wolf [38] developed a dependence analysis technique for Rapide, a modeling language that allows one to specify and simulate the behavior of an event-based system. This approach accounts for inter- and intra-flow dependencies. Baresi et al. [7] embed publish-subscribe constructs into Bogor [35], an extensible model checker, to allow for automatic verification of pub-sub architectures while coping with the problem of state-space explosion. Neither of these two approaches provides a mapping of the model to an implementation. Zhao introduced slicing for the software architecture modeling language Wright [45]. This technique is of limited use for DEB systems since Wright explicitly captures several relationships between event-based components that a programming language such as Java does not. Millett and Teitelbaum introduced slicing of Promela models [27]. However, it is unclear how Promela's channels could be mapped to DEB systems that do not use channels, such as those of most existing MOM systems [29].

Halfond and Orso presented an approach that statically recovers implicit input interfaces of web applications and then groups the identified inputs that could be part of the same interface by analyzing data-flow paths [20]. [21] extends this work by recovering indirect interface invocations. While instructive, this approach cannot be used for recovering the invoked interfaces of a DEB system because DEB components neither encode invocations into one data object nor return their invoked interfaces to a specific client.

Purandare et al. [33] developed a program analysis technique that determines the conditions under which a component publishes a message. The technique is targeted at the Robotics Operating System (ROS) [2], which provides MOM-based publish-subscribe functionality. The technique reports the conditions under which a message source interface is called, which methods may transitively invoke that interface, and the specific topic to which a component may publish a message. However, unlike Eos, this technique does not determine the attributes that constitute a message, distinguish between CMTs and PMTs, or determine which messages may be published due to the consumption of another message.

Jayaram and Eugster developed three static analysis techniques targeted at improving the performance of DEB systems [23] implemented in EventJava [18]. EventJava is an extension to Java that provides support for event-based interaction. A system written in EventJava does not deal with the problem of ambiguous interfaces: the event (i.e., message) types in EventJava are explicitly defined via special "event methods", where the attributes of an event are specified as method arguments. While useful for constructing DEB applications, such capabilities are not typically found in existing MOMs. The proposed event causality analysis in [23] is a combination of static and dynamic analysis that relies on the event methods and EventJava's runtime framework to obtain dependencies between events. While a potentially useful complement to the analysis provided by Eos, adopting the static portion of EventJava's analysis in our approach would require that we overcome two difficulties. First, programmers would have to indicate which event types need to be causally ordered to ensure safety. Second, programmers need to explicitly indicate causally-independent events to avoid a high-overhead pessimistic analysis.

Lexical source model extraction (*LSME*) is an approach for extracting information from source code [30]. Of particular relevance is a case study in which LSME was used to extract inter-flow dependencies from the message-oriented Field programming environment [34]. However, LSME was unable to ensure that its results included all message sources and sinks. Additionally, LSME did not analyze intra-flow dependencies, forcing an engineer to assume that, within a component, each published message depends on each consumed message. Our analysis of LSME in the context of our previous work [32] quantified these shortcomings in terms of LSME's decreased precision and recall when applied to five DEB systems used in our evaluation in Section 4.

Our previous work, *Helios* [32], directly inspired the work in this paper. Helios also computes message-flow dependencies of DEB components. Our quantitative evaluation demonstrated that such an approach can exhibit very good precision and recall. However, Helios supports only DEB applications that use nominal message types, i.e., message types that are explicitly defined in the underlying programming language. Furthermore, Helios mandates that dispatching of messages be localized within the method that implements the message sink (e.g., the `consume` method in Figure 1). While these two constraints simplified the analysis Helios had to perform as compared to Eos, they also restricted Helios's applicability to a small class of existing DEB systems, and in some cases required partial re-implementation of such systems [32] to fit Helios's assumptions.

6. CONCLUSION

The rich body of work targeted at analyzing the dependencies in traditional software systems provides little benefit when applied to DEB systems. The rare techniques that are directly applicable to DEB systems are either (1) generic tools whose accuracy is inadequate or (2) specialized approaches that make significant limiting assumptions. Eos has been developed to eliminate both of these shortcomings. Its analysis algorithms directly capture the characteristics of modern DEB systems and their underlying MOM platforms. This is reflected in the empirical data: in our evaluations, Eos exhibited an average precision and recall higher than 90%, while in a large number of cases it achieved perfect scores. Additionally, Eos's runtime performance gives us confidence that it is likely to scale to very large DEB systems. Finally, a preliminary case study suggests that Eos can be used effectively in aiding change impact analysis, a common software maintenance activity.

There are a number of avenues of future work. We have already begun pursuing some (e.g., assessing the suitability of Eos for system maintenance). Several additional issues were highlighted by our evaluation results. Some of them (e.g., conditional attributes) were due to less common ways of message storage, access, retrieval, and propagation, which we had not considered. Others (e.g., path-insensitive string propagation) were a result of a deliberate design choice. In either case, enhancements to Eos to address these issues will require a careful analysis of trade-offs between the cost in analytic complexity and runtime performance vs. the benefit in added precision and recall.

7. ACKNOWLEDGMENTS

The authors wish to thank Kihoon Jeoung for his assistance with the analysis of the Spark system. This work has been supported by the National Science Foundation under award numbers 1117593, 1218115, and 1321141.

8. REFERENCES

[1] mfa:start [USC Softarch Wiki]. http://softarch.usc.edu/wiki/doku.php?id=mfa:start, 2012.

[2] Documentation - ROS Wiki, 2013.

[3] Ignite Realtime: Smack API, 2013.

[4] Ignite Realtime: Spark IM Client, 2013.

[5] A. Aho et al. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] R. Arnold and S. Bohner. Impact Analysis - Towards a Framework for Comparison. In *the International Conference on Software Maintenance*, 1993.

[7] L. Baresi et al. On Accurate Automatic Verification of Publish-Subscribe Architectures. In *ICSE*, 2007.

[8] K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2009.

[9] K. Bierhoff and J. Aldrich. PLURAL: Checking Protocol Compliance Under Aliasing. In *ICSE Companion*, 2008.

[10] D. Binkley and M. Harman. A Survey of Empirical Results on Program Slicing. *Advances in Computers: Advances in Software Engineering*, 2004.

[11] F. Biscotti et al. Market Share: AIM and Portal Software, Worldwide, 2009. *Gartner Market Research Report*, April 2010.

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

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

[14] C. Clifton et al. MultiJava: Design Rationale, Compiler Implementation, and Applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.

[15] J. Correira and F. Biscotti. Market Share: AIM and Portal Software, Worldwide, 2005. *Gartner Market Research Report*, 2006.

[16] G. Cugola et al. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE TSE*, 2001.

[17] P. Eugster et al. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 2003.

[18] P. Eugster and K. Jayaram. EventJava: An Extension of Java for Event Correlation. In *European Conference on Object-Oriented Programming*. Springer, 2009.

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

[20] W. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *ESEC/FSE*, 2007.

[21] W. Halfond and A. Orso. Automated Identification of Parameter Mismatches in Web Applications. In *FSE*, 2008.

[22] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. In *ESEC/FSE*. Springer, 1999.

[23] K. Jayaram and P. Eugster. Program Analysis for Event-Based Distributed Systems. In *International Conference on Distributed Event-based Systems*, 2011.

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

[25] S. Malek et al. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE TSE*, 2005.

[26] N. Medvidovic et al. The Role of Middleware in Architecture-Based Software Development. *Int. J. of Softw. Eng. and Knowl. Eng.*, 2003.

[27] L. Millett and T. Teitelbaum. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *International Journal on Software Tools for Technology Transfer*, 2000.

[28] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

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

[30] G. C. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM TOSEM*, 1996.

[31] D. Popescu. *Dependence Analysis for Distributed Event-Based Systems*. PhD thesis, University of Southern California, 2012.

[32] D. Popescu et al. Impact Analysis for Distributed Event-Based Systems. In *International Conference on Distributed Event-Based Systems*, 2012.

[33] R. Purandare et al. Extracting Conditional Component Dependence for Distributed Robotic Systems. In *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[34] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 1990.

[35] Robby et al. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *ESEC/FSE*, 2003.

[36] K. Sachs et al. Performance Evaluation of Message-Oriented Middleware Using the SPECjms2007 Benchmark. *Performance Evaluation*, 2009.

[37] K. Sachs et al. Benchmarking Publish/Subscribe-Based Messaging Systems. In *Proc. BenchmarX*, 2010.

[38] J. Stafford and A. Wolf. Architecture-Level Dependence Analysis for Software Systems. *Int. J. of Softw. Eng. and Knowl. Eng.*, 2001.

[39] R. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, 1996.

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

[41] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 1995.

[42] R. Vallée-Rai et al. Soot - a Java Bytecode Optimization Framework. In *Conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[43] M. Weiser. Program slicing. In *ICSE*, 1981.

[44] B. Xu et al. A Brief Survey of Program Slicing. *ACM SIGSOFT Software Engineering Notes*, 2005.

[45] J. Zhao et al. Change Impact Analysis to Support Architectural Evolution. *J. Software Maintenance and Evolution Research and Practice*, 2002.