On Ordering Problems in Message Passing Software

Yuheng Long Mehdi Bagherzadeh Eric Lin Ganesha Upadhyaya Hridesh Rajan
Iowa State University, USA
{csgzlong,mbagherz,eylin,ganeshau,hridesh}@iastate.edu

Abstract

The need for concurrency in modern software is increasingly fulfilled by utilizing the message passing paradigm because of its modularity and scalability. In the message passing paradigm, concurrently running processes communicate by sending and receiving messages. Asynchronous messaging introduces the possibility of message ordering problems: two messages with a specific order in the program text could take effect in the opposite order in the program execution and lead to bugs that are hard to find and debug. We believe that the engineering of message passing software could be easier if more is known about the characteristics of message ordering problems in practice. In this work, we present an analysis to study and quantify the relation between ordering problems and semantics variations of their underlying message passing paradigm in over 30 applications. Some of our findings are as follows: (1) semantic variations of the message passing paradigm can cause ordering problems exhibited by applications in different programming patterns to vary greatly; (2) some semantic features such as in-order messaging are critical for reducing ordering problems; (3) modular enforcement of aliasing in terms of data isolation allows small test configurations to trigger the majority of ordering problems.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming — Distributed programming; D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent programming structures

Keywords Message passing, quantification of message ordering problems, asynchronous messages

1. Introduction

The concurrency revolution [45] has renewed interest in the message passing paradigm [2] because of its modular [3, 6, 18, 26, 27] and scalable [48] concurrent programming model. Use of Erlang [5] in the development of Amazon Web Services and Akka [4] in the Guardian's web site are just a few examples of such interest.

The message passing paradigm allows for modular development [3, 26] and reasoning [6, 18, 27] of its programs. In the message passing paradigm, a module (process) communicates with other concurrently running processes by sending and receiving messages. For enhanced throughput, these messages are usually sent asynchronously [3, 4]. However, asynchronous sending of messages introduces the possibility of ordering problems that we call the *message ordering problem* and makes development of message passing software difficult and error-prone. In an ordering problem, two messages with a specific sequential order in the program text could take effect in the opposite order in the program execution. Previous work [12, 24, 30, 47] shows that ordering problems are prevalent and could lead to bugs that are hard to find and debug. A bug usually happens *when a programmer, based on the program text, assumes a message order which may not actually exist during the program execution.*

```
1 class Client extends Actor {
2   ActorName server; ..
3   @message void start() {
4     send(server, "set", 1);        // async message send
5     int v = send(server, "get");   // async message send
6     assert(v == 1);                // assertion Φ
7   } ..
8 }
```

Figure 1: A message ordering problem happens when using the program text, on lines 4–5, a programmer mistakenly assumes that the asynchronous message `set` is sent and processed before `get`, causing a real world bug [24].

To illustrate, Figure 1 – written in the message passing programming language ActorFoundry [1] – shows a simplified version of a real world bug [24] caused by a message ordering problem. A `Client` process (actor), sends an asynchronous message `set` to a `server` process, on line 4, to set the value of a variable in the server to 1. Later, the client sends another asynchronous message `get`, on line 5, to read the value of the variable in the server. Because sending of the `set` message appears before sending of the `get` message in the program text, it is very likely [12, 24, 30, 47] that a sequentially-trained programmer assumes that the `set` message is sent and processed in the server before the `get` message and therefore `get` reads the value 1 set by `set` and the assertion Φ holds; and this is exactly what the programmer assumes in this example [24, 47]. However, the execution of the program could surprise the programmer by causing a bug and violation of Φ. This is because, depending on the messaging semantics of ActorFoundry, `set` and `get` messages could during program execution be received and processed by the server in an order which is the opposite of their appearance order in the program text in the client. This in turn causes the `get` message to return a value that may not be equal to 1 and therefore cause a bug and violate Φ. This bug happens because, based on the program text, the programmer mistakenly assumes an ordering (in-order delivery and processing) between `set` and `get` messages that may not exist during the program execution, depending on the semantics of the message ***send*** operation.

1.1 Root Causes of Message Ordering Problems

We believe that the semantic variations of the following 3 criteria in the message passing paradigm are the root causes of message ordering problems:

- $\mathfrak{C}1$: Message synchronization semantics, categorized into asynchronous and synchronous messaging [22].
- $\mathfrak{C}2$: Message processing semantics, categorized into nondeterministic and in-order delivery and processing.
- $\mathfrak{C}3$: Sharing semantics, categorized into data sharing and data isolation of memory effects between processes.

In addition to semantic variations of $\mathfrak{C}1$–$\mathfrak{C}3$, two other factors could make it harder for a programmer to understand and avoid message ordering problems: (1) semantic tuning of the underlying message passing paradigm of a single program and (2) diversity of semantic variations that are available for the message passing paradigm.

1.1.1 Semantic Tuning in a Single Paradigm

There are message passing languages and frameworks, such as Akka [4], that allow the programmer to configure various semantics for $\mathfrak{C}1$–$\mathfrak{C}3$ in the underlying message passing paradigm of a single program. Such semantic tuning could make message ordering problems harder to understand and avoid: a program that is free from message ordering problems in one configuration may suffer from ordering problems in another configuration.

For example, suppose the code in Figure 1 was written in Akka instead of ActorFoundry. Akka allows the programmer to configure the message synchronization semantics of ***send*** and to change it from the default asynchronous messaging to synchronous. Such configuration of message synchronization semantics takes original program with a message ordering problem and changes it to a program without any ordering problems. A similar scenario in the opposite direction can happen as well: a configuration change from synchronous to asynchronous messaging could change a program without message ordering problems into one with message ordering problems.

To further illustrate semantic tuning, Akka configurations allow a programmer to configure and run a program with either asynchronous or synchronous messaging for $\mathfrak{C}1$. For $\mathfrak{C}2$, ActorFoundry and HAL constraints [19] allow configurations of both nondeterministic and in-order message delivery and processing. For $\mathfrak{C}3$, Kilim annotations [43] allow configurations for both data sharing and isolation.

1.1.2 Semantic Diversity in Multiple Paradigms

There are several message passing programming languages and frameworks available with different semantics for $\mathfrak{C}1$–$\mathfrak{C}3$. Such semantic diversity could make it difficult to understand and avoid message ordering problems, especially when a program with no ordering problems in one paradigm may suffer from ordering problems in another paradigm [39] and vice versa. For example, deprecation of Scala Actor [17] message passing paradigm and its evolution into Akka may require a programmer to migrate their program from Scala Actor to Akka with cautionary warnings like:

"Due to differences between the two actor implementations it is possible that errors appear. It is recommended to thoroughly test the code after each step of the migration process." — The Scala Actors Migration Guide [39]

There is no guarantee that a program in Scala Actor with no ordering problems will have no ordering problems when migrated to Akka.

To illustrate the semantic diversity, Scala [16] and Panini [37, 38] support both asynchronous and synchronous messaging for $\mathfrak{C}1$ whereas Erlang [5] and standard actor model only support asynchronous messaging. For $\mathfrak{C}2$, ActorFoundry messages are delivered and could be processed nondeterministically whereas Akka and JCoBox [40] support in-order delivery and processing. For $\mathfrak{C}3$, the standard actor model supports data isolation between processes, Scala allows sharing and ActorFoundry supports both.

1.2 Understanding Message Ordering Problems

On the one hand, the concurrency revolution requires sequentially-trained programmers that mostly think sequentially [30] to write concurrent programs in the message passing paradigm, which is a modular concurrent programming model with increasing popularity. On the other hand, the message ordering problems make programming in this paradigm difficult and error-prone [24, 47] for these programmers [30, 45]. We believe that the engineering of message passing software could be easier if we know more about the characteristics of message ordering problems in practice. This makes it critical to study and understand the relation between the semantics of a message passing paradigm and the ordering problems of its programs.

1.3 Contributions

This paper makes the following contributions:

- Illustration of the relation between message ordering problems and synchronization semantics, message delivery and processing semantics, and sharing semantics; and

- Quantify the relation between the semantics of *five* message passing models and their ordering problems for various concurrent programming patterns in about 130,000 lines of code adapted from previous work; and
- Study the minimum number of processes and messages that are required to trigger these ordering problems; and
- Quantify the overlapping of message passing models in preventing the same ordering problems; and
- Discuss implications of our findings for application developers, framework designer and application verifiers in the engineering of concurrent message passing software.

Outline §2 illustrates the relation of ordering problems and various semantics of the message passing models. §3 defines causal happens-before relations [23] for our five message passing models, defines unsafe interleavings of operations that lead to ordering problems and discusses our static analysis for detection of ordering problems. §4 presents our study setup. §5 discusses our observations and their implications. §6 discusses related work. §7 concludes the paper.

2. Problems

This section illustrates the relation between ordering problems and semantic variations of ℭ1–ℭ3 for message synchronization, message delivery and processing, and data sharing semantics of the underlying message passing model.

```
1  class Client extends Actor{
2    ActorName server;
3    Client(ActorName s){ server = s; }
4    @message void start(){
5      send(server, "set", 1);
6      int v = send(server, "get");
7      assert(v == 1);       // assertion Φ
8    }
9  }
10 class Server extends Actor{
11   int val = 0;
12   @message void set(int v){ val = v; }
13   @message int get(){ return val; }
14 }
15 class Driver{
16   static void main(String[] args){
17     ActorName server = createActor(Server.class);
18     ActorName client = createActor(Client.class);
19     send(client, "start", server);
20   }
21 }
```

Figure 2: (1) With synchronous messaging for ℭ1 or in-order message delivery and processing for ℭ2, `Client` has no ordering problems. (2) With alternative semantics for ℭ1 or ℭ2, `Client` suffers from message ordering problems.

To illustrate, Figure 2 shows the `Client` from Figure 1 with its `Server`. The code is implemented in ActorFoundry [1] where a process is declared by a class that extends the `Actor` class and is instantiated using the `createActor` constructor; a message handler is a method marked with the `@message` annotation. For example, lines 1–9 declare the process `Client`, line 18 instantiates a process instance `client` and line 12 declares a message handler `set`. The `Server` keeps track of a variable `val` and provides access to it using its message handlers `set` and `get`. The `Client` sets the value of `val` to 1 by sending a `set` message to the server, on line 5, reads its value by sending a `get` message, on line 6 and finally checks if `val` is actually set to 1 using the assertion Φ, on line 7. `Driver`, the entry point to the program, creates `client` and `server` process instances, on lines 17–18, and initiates the execution of `client` by sending it a `start` message, on line 19. The name of the receiving process, a message name and message parameters are required when sending a message using the **`send`** operation. In the message send on line 5, `server` is the name of the receiving process, `set` is the name of the message and 1 is the parameter of the message in the server. The assertion Φ is the representative for message ordering problems. That is, the assertion holds when there is no ordering problem and is otherwise violated.

Semantics of ActorFoundry ActorFoundry supports: (1) asynchronous **`send`** and synchronous blocking **`call`** messaging for ℭ1; (2) nondeterministic delivery and processing and programmer-specified processing of messages [22] for ℭ2; and (3) data isolation for ℭ3 by its call by value messaging (or by relying on the programmer to guarantee data isolation for call by reference messaging).

In the following, we illustrate the relation of message ordering problems and ℭ1–ℭ3.

2.1 ℭ1: Message Synchronization Semantics

To illustrate the relation of ordering problems and message synchronization, we consider two alternative semantics for **`send`**: asynchronous and synchronous messaging[1].

2.1.1 Synchronous

With synchronous semantics for message sends, `Client` has no ordering problems and the assertion Φ holds. This is because the blocking semantics of message sends ensures that the `set` message that appears before the `get` message in the program text is in fact sent and processed before `get`.

2.1.2 Asynchronous

On the other hand, with asynchronous semantics for message sends, `Client` has message ordering problems. This is because, the `set` message may be processed by the server after the `get` message, which is the opposite of their order in the program text, especially if the messages are processed in a nondeterministic order. Since the message handlers for messages `set` and `get` write and read the same variable `val` in the server, their execution in the opposite order causes a message ordering problem. Despite the programmer's assumption the `get` message may read a value of `val` which is not equal to 1 and the assertion Φ is violated.

[1] This is for illustration purposes only, otherwise we treat **`call`** messages as synchronous and **`send`** as asynchronous as specified in the semantics.

2.2 ℭ2: Message Delivery and Processing Semantics

To illustrate the relation of ordering problems and message delivery and processing, we consider two alternative semantics for message delivery and processing: nondeterministic and in-order. In nondeterministic messaging semantics, there is no order for either the delivery or processing of messages. However, in in-order delivery, two messages sent directly (with no intermediate processes) from one process to another are guaranteed to be delivered in the same order that they are sent. Similarly, in in-order processing, messages are processed in the order by which they are delivered.

2.2.1 In-order

With in-order delivery and processing, `Client` has no ordering problems, because the `set` and `get` messages are delivered and processed in the order in which they appear in the program text. That is, the `set` message is delivered and processed before the `get` message in the server. This is true even for asynchronous message synchronization semantics.

2.2.2 Nondeterministic

Unlike in-order delivery and processing, with nondeterministic delivery and processing, `Client` has ordering problems. This is because `set` and `get` messages could be delivered to the server and processed in any arbitrary order. Such arbitrary order could include an order in which `get` is processed before `set`.

2.3 ℭ3: Sharing Semantics

To illustrate the relation of message ordering problems and sharing, we consider two alternative semantics for sharing among processes: data sharing and data isolation.

In ActorFoundry, a process cannot directly access the internal state of another process because call by value messaging guarantees data isolation. Therefore, data sharing can only happen through call by reference messaging [32].

To illustrate, Figure 3 shows a variation of Figure 2 in which the server keeps track of the value of an object `val`, line 12, instead of its primitive integer counterpart in Figure 2. The omitted code remains the same.

```
1  class Client extends Actor { ..
2    @message void start() {
3      Value val = new Value();
4      send(server, "init", val);
5      send(server, "set", 1);
6      val.num = 2;
7      assert(val.num == 2);    // assertion Φ
8    }
9  }
10 class Value{ int num; }
11 class Server extends Actor{
12   Value val;
13   @message void set(int v){ val.num = v; }
14   @message void init(Value v){ val = v; }
15 }
```

Figure 3: (1) With isolation ℭ3 `Client` has no ordering problems. (2) With data sharing `Client` suffers from message ordering problems.

2.3.1 Data Sharing

With data sharing among processes, `Client` has message ordering problems especially if message sends are asynchronous and message delivery and processing is nondeterministic. This is because the call by reference semantics of the `init` message, on line 4, shares the object `val` between the client and the server and the asynchronous semantics of message sends allows the assignment in the client, on line 6, to execute before the processing of the `set` message in the server, sent on line 5. This could result in values for `val.num` that may not be equal to 2 and violate Φ.

2.3.2 Data Isolation

On the other hand, with data isolation, the `Client` has no message ordering problems. This is because the call by value semantics for sending of the `init` message transfers a deep copy of `val` to the server instead sharing it. Therefore, the client and the server work on two separate unrelated copies of the object.

2.4 Summary

As illustrated, the presence of message ordering problems can vary greatly as the semantics of message synchronization, message delivery and processing, and sharing vary in the underlying message passing paradigm. Figure 4 summarizes the relation between message ordering problems in `Client` and the semantic variations for ℭ1–ℭ3.

ℭ1 *Message Synchronization*		ℭ2 *Message delivery & processing*		ℭ3 *Data sharing*	
synchronous	*asynchronous*	*nondeterministic*	*inorder*	*isolation*	*sharing*
No	Yes	Yes	No	No	Yes

Figure 4: Relation of message ordering problems in `Client` with semantics variations of ℭ1–ℭ3. Yes and No mark the presence and absence of ordering problems.

3. Detection Analysis for Ordering Problems

This section defines 5 message passing models based on semantic variations of ℭ1–ℭ3, defines happens-before [23] relations of their operations and describes our static analysis to detect ordering problems in these models.

3.1 Five Message Passing Models

Based on the semantic variations of ℭ1–ℭ3 for message synchronization, message delivery and processing, and sharing we define five message passing models: (1) *base* is a model with asynchronous and no built-in synchronous messaging for ℭ1, non-deterministic message delivery and processing for ℭ2 and data sharing among processes for ℭ3; (2) +*sync* adds built-in synchronous messaging to *base*; (3) +*inorder* adds in-order delivery and processing of messages to *base*; (4)+*trans* adds transitive in-order delivery and processing to *base*; and finally (5) +*isol* adds data isolation to *base*. In transitive in-order delivery and processing, two messages sent from one process to another are delivered and processed in the same order they are sent, if the first message is sent directly (with no intermediate processes) while the second message could be sent directly or indirectly.

3.2 Happens-Before Relations

A happens-before relation defines a causal order between executions of operations. Definition 1 defines the happens-before relation $\prec$ for our models. To accommodate the encoding of both synchronous and asynchronous messages, we divide sending and processing of a message into $\mathit{MsgSend}$ and $\mathit{MsgReturn}$ operations and divide the processing of the message into $\mathit{MsgStart}$ and MsgEnd.

DEFINITION 1. ***(Happens-before relation*** $\prec$*)*
Let $Op(o, i, A)$ be the operation o at the position i in the program text of a process A. Let $\mathit{MsgSend}(m, A, A')$ be sending of a message m from process A to A', $\mathit{MsgStart}(m, A, A')$ and $\mathit{MsgEnd}(m, A, A')$ be the start and end of the processing of m in A' and $\mathit{MsgReturn}(m, A, A')$ be the returning to A after the end of the processing of m in A'; let $\mathit{Handler}(m, A')$ be the message handler of m sent in A'. Let $\prec$ denote the happens before relation. Then the following happens-before relations holds in our models.
(i) *For* base*:*

1. *if $i < j$, then $Op(o, i, A) \prec Op(o', j, A)$*
2. $\mathit{MsgSend}(m, A, A') \prec \mathit{MsgStart}(m, A, A')$
3. $\mathit{MsgSend}(m, A, A') \prec \mathit{MsgReturn}(m, A, A')$
4. $\mathit{MsgStart}(m, A, A') \prec \mathit{MsgEnd}(m, A, A')$
5. *if $Op(o,i,A') \in \mathit{Handler}(m,A')$ then $\mathit{MsgStart}(m,A,A') \prec Op(o, i, A')$ and $Op(o, i, A') \prec \mathit{MsgEnd}(m, A, A')$*

(ii) *For* +sync *and its synchronous messaging:*

6. $\mathit{MsgEnd}(m, A, A') \prec \mathit{MsgReturn}(m, A, A')$

(iii) *For* +inorder *and its in-order delivery and processing:*

7. *if $\mathit{MsgSend}(m, A, A') \prec \mathit{MsgSend}(m', A, A')$ then $\mathit{MsgEnd}(m, A, A') \prec \mathit{MsgStart}(m', A, A')$*

(iv) *And finally for* +trans *and its transitive in-order:*

8. *if $\mathit{MsgSend}(m, A, A') \prec \mathit{MsgSend}(m_1, A, B_1)$ and $\mathit{MsgSend}(m_1, A, B_1) \prec \mathit{MsgSend}(m_2, B_1, B_2) \prec \ldots \prec \mathit{MsgSend}(m_n, B_{n-1}, B_n) \prec \mathit{MsgSend}(m', B_n, A')$ then $\mathit{MsgEnd}(m, A, A') \prec \mathit{MsgStart}(m', B_n, A')$.*

Data isolation in *+isol* does not add any extra happens-before relations between operations of the model. The happens-before relation is transitively closed [13].

Definition 1 says that in:
base model (1) An operation happens-before another operation if the former appears before the latter in the program text; (2) Sending of a message in the sender happens before the start of its processing in the receiver; (3) Sending of the message happens before its returning; (4) Start of the processing of a message happens before the end of its processing, and (5) Start of the processing of the message happens before any operation in its message handler.

+sync (6) For a synchronous message the end of the processing of a message happens before the returning of the message. This is not true for asynchronous messages where the message returns right after it is sent, independent of the start and end of its processing. All happens-before relations of *base* are included in the happens-before relations for *+sync*, as *+sync* is built on top of *base*. The same applies to any of our models built on top of another.

+inorder (7) For two messages sent to another process *directly* (with no intermediate process) the first message is delivered and processed first and therefore the end of its processing happens before the start of the second message.

+trans (8) For two messages sent to another process in which the first message is sent directly and the second message is sent directly or *indirectly* (through one or more intermediate processes), the first message is delivered and processed before the second one.

Figure 5 illustrates the happens-before relations for operations of the `Client` process in Figure 2 for two models *base* and *+sync*.

3.3 Unsafe Interleavings

Definition 2 defines an unsafe interleaving of operations of a process that leads to a message ordering problem.

DEFINITION 2. ***(Unsafe interleavings of operations)***
Let op_1 and op_2 be operations of a process A such that op_1 appears before op_2 in the program text; let op_1 and op_2 send messages to other processes, directly or indirectly, and cause the execution of two other operations op'_1 and op'_2. Then op_1 and op_2 from an unsafe interleaving if and only if:

- *There is no happens-before relation $\prec$ between op'_1 and op'_2, as defined in Definition 1; and*
- *Operations op'_1 and op'_2 have conflicting memory effects.*

Definition 2 is based on an observation from previous work that most sequentially-trained programmers think sequentially and they often wrongly assume that during the execution of a program messages are sent and processed in the same order that they appear in the program text [12, 24, 30, 47]. Two memory effects conflict if they access the same memory location and at least one of them is a write.

3.4 Detection Analysis

Our static analysis to detect ordering problems has two phases: (1) convert a program to its corresponding system graph and (2) analyze the system graph for the unsafe interleaving of operations that lead to ordering problems.

System graph The system graph of a program is an alternative representation of the program that encodes its processes, their message exchanges, happens-before relations and memory effects. One benefit of the system graph is that it

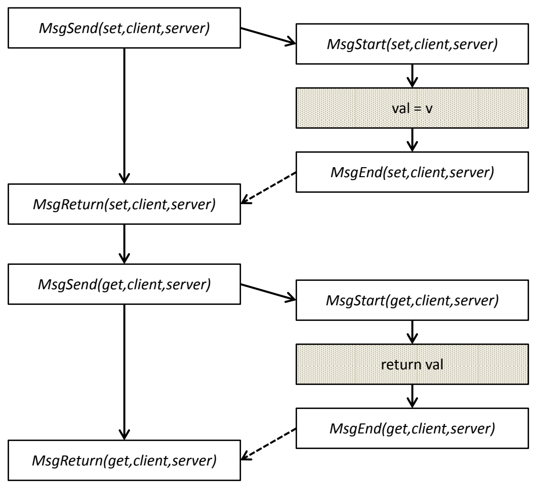

MsgSend(set,client,server)
MsgStart(set,client,server)
val = v
MsgEnd(set,client,server)
MsgReturn(set,client,server)
MsgSend(get,client,server)
MsgStart(get,client,server)
return val
MsgEnd(get,client,server)
MsgReturn(get,client,server)

Figure 5: Happens-before relations for `client` and `server` processes in Figure 2 for the *base* model (solid arrows) and +*sync* model (solid and dashed arrows both). No happens-before relation in *base* between server operations for reading and writing `val` (in grey) causes a message ordering problem and violation of Φ.

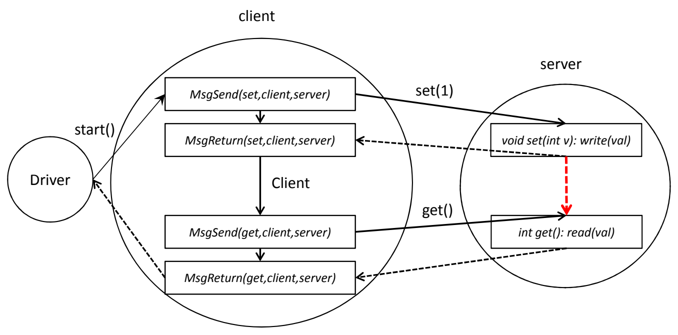

client
server
Driver
start()
MsgSend(set,client,server)
MsgReturn(set,client,server)
Client
MsgSend(get,client,server)
MsgReturn(get,client,server)
set(1)
get()
void set(int v): write(val)
int get(): read(val)

Figure 6: System graph for the program in Figure 2 for the *base* model (solid happens-before arrows) and +*sync* model (solid and dashed arrows both).

allows our analysis to be independent of the implementation language of the program. Figure 6 shows the system graph for the program in Figure 2 for *base* and +*sync* models.

Finding ordering problems To find ordering problems of a program in a message passing model, our analysis: (1) first uses the happens-before relations of the model, defined in §3.2, to statically infer happens-before relations among as many operations of processes as possible. Output of this phase is a set of process operations with no happens-before relations between them. (2) For each pair of these operations the analysis uses the sharing semantics of the model to decide if their memory effects conflict. Finally (3) pairs of operations that do not have any happens-before relation and their memory effects conflict are flagged, as defined in Definition 2, as unsafe interleavings.

To illustrate, consider the `Client` process in Figure 2 and its system graph in Figure 6. In `Client`, sending of the `set` message appears before the sending of the `get` message in the program text. And sending of `set` and `get` messages in the `Client` causes the execution of the `set` and `get` message handlers in the `Server`, respectively. Memory effects of message handlers `set` and `get` are writing and reading of the value of the same location `val`. When analyzing the system graph in the *base* model (with solid happens-before relation arrows), there is no happens-before relation between the execution of `set` and `get` message handlers in the server which means that their memory effects conflict. Therefore, the pair of `set` and `get` messages in the client form an unsafe interleaving that the analysis flags as a message ordering problem. However, when analyzing the system graph in the +*sync* model (with both solid and dashed happens-before arrows) the analysis using transitivity of the happens-before relation infers that there is a happens-before relation (red arrow) between the execution of `set` and `get` message handlers. Therefore, there does not exist any unsafe interleavings or message ordering problems in the client.

4. Case Studies

This section discusses our benchmark applications and their refactoring process.

4.1 Benchmarks

We study 34 small to large sized benchmark applications totalling about 130,000 lines of code adapted from the following previous work: (1) Basset [24], (2) Habanero [20], (3) Jetlang [21], (4) well-known benchmarks, including NAS Parallel Benchmarks [15] and parallel JavaGrande [42] and (5) examples shipped with the Panini compiler [34]. These benchmark applications are implemented in a variety of concurrent programming patterns [31] including Master Worker (MW), Loop Parallelism (FL), Pipeline (PL) and Event-based Coordination (EC). Figure 8 shows our benchmarks.

For our analysis, we first refactored the benchmark applications to their corresponding message passing programs in the message passing language Panini [6, 37, 38]. Then we converted these programs into their corresponding system graphs and later analyzed these system graphs.

4.1.1 Refactoring to Panini

Panini is a message passing concurrent programming language that promotes capsule-oriented programming [37, 38]. In Panini, a capsule, similar to a process, encapsulates its data and thread of control and communicates with other capsules by sending and receiving messages. A procedure of a capsule is similar to a message handler of a process. To illustrate, Figure 7 shows a refactoring of the ActorFoundry program in Figure 2 into Panini where lines 1–5 declare the `Client` capsule; lines 14–18 define a system declaration in which lines 16 and 15 declare `client` and `server` capsule instances and line 17 connects them together.

To refactor our benchmarks, we follow a very strict and non-intrusive set of mostly syntactic refactoring steps. Our refactoring steps for a multi-threaded benchmark application are: (1) refactor a thread object to a capsule; (2) refactor a synchronized method or block to a capsule procedure; and (3) create a capsule field for a top-level class instance. Our refactoring steps for a message passing benchmark application are: (1) refactor a process to a capsule (2) refactor a message handler to a capsule procedure and a message send to an invocation of the capsule procedure; (3) create processes in its system declaration.

```
capsule Server {
  int val = 0;
  void set(int v) { val = v; }
  int get() { return val; }
}
capsule Client(Server server) {
  void start() {
    server.set(1);
    int v = server.get();
    assert(v == 1);
  }
}
capsule Driver {
  design {
    Server server;
    Client client;
    client(server);
  }
  void run() {
    client.work();
  }
}
```

Figure 7: Refactoring of Figure 2 in Panini [37, 38].

4.1.2 System Graph Construction

Conversion of a Panini program to its corresponding system graph and computation of its processes and their message exchanges is rather straightforward. In Panini, capsule instances are statically specified in the system declaration and cannot be stored or passed among capsules and there is no subtyping relation among capsule types [6]. To compute the memory effects of capsules, we use a sound alias analysis technique from previous work [9, 28]. Note that our analysis is also applicable in the presence of dynamic process creation as long as a system graph soundly abstract the processes of the program and their communications.

4.2 Soundness and Completeness

Soundness of our analysis guarantees the absence of false negatives. That is, if there is an ordering problem our analysis will report it. For completeness, and to avoid false positives, we manually verify that a reported ordering problem, is an actual problem.

It is noteworthy that the existence of a message ordering problem in a benchmark application does not necessarily mean that the application is buggy. This is because, similar to data races, not every message ordering problem leads to a bug and some may even be intentional for performance purposes. Also we analyze the application in five different message passing models while the application is originally implemented with only one of these models in mind. Therefore, a message passing problem that exists in one model may not exist in the original model.

5. Quantification, Observations and Implications

This section quantifies the relation between message ordering problems and our message passing models defined in §3.1 and studies how these ordering problems can be triggered in various concurrent programming patterns [31]. It also discusses observations of our study and their implications as complementary guidelines that could be useful for application developers, application verifiers and framework designer when engineering message passing software.

It is noteworthy that, although our findings are the result of studying a large amount of code in a variety of applications we do not intend to draw general conclusions about all message passing applications or paradigms. Also, all our findings and implications are only associated with the applications and message passing models studied.

5.1 Models and Patterns

We first discuss our findings for our five message passing models of *base*, +*sync*, +*inorder*, +*trans* and +*isol* per concurrent patterns of our benchmark applications. Figure 8 and its bar chart representation in Figure 9 show our findings. Our findings suggest the following trends.

5.1.1 Event-Based Coordination

- Together, in-order message delivery and processing of +*inorder* and message synchrony of +*sync*, written as (+*inorder* and +*sync*), prevent 97% of message ordering problems in Event-based Coordination applications.
- Data isolation in +*isol* does not prevent any ordering problems in this pattern.

In-order messaging in +*inorder* prevents most of the ordering problems because Event-based Coordination [31] applications usually involve multiple iterations where the same set of messages is exchanged among the same set of processes in each iteration; and the processing of messages of one iteration should be ordered before messages of its subsequent iterations. In-order messaging guarantees such ordering for each iteration. Synchronous messaging of +*sync* helps with ordering problems of the check-then-act idiom in Barbershop and Philosopher applications in this pattern, where it ensures that a check message blocks the execution of its act message until the check message is processed. Data isolation of +*isol* prevents the least number of ordering problems because programmers manually enforce the data isolation or exchanged messages are mostly primitive values. Transitivity of +*trans* is not very important, because system graphs of these applications, except Barbershop, do not include any *triangle pattern*. In a triangle pattern, a process A sends a message to process A' directly and sends another message to A' indirectly. Triangle patterns are main beneficiaries of transitivity of message delivery and processing.

	Applications	LOC	*base*	*+sync*	*+inorder*	*+trans*	*+isol*
EC	Bank	42	6	6	4	6	6
	Barbershop	82	15	11	9	14	15
	Factorial	28	0	0	0	0	0
	Philosophers	60	8	5	3	8	8
	Pi	47	0	0	0	0	0
	SC	39	2	2	1	2	2
	Signature	20	0	0	0	0	0
	PingPong	46	0	0	0	0	0
	ThreadRing	34	0	0	0	0	0
	Server	39	1	1	0	1	1
	Total		**32**	**25 (↓22%)**	**17 (↓47%)**	**31 (↓3%)**	**32 (↓0%)**
	Unresolved		16 (50%)				
FL	BT	34,804	55	5	25	55	30
	CG	3,434	4	0	2	4	2
	FT	4,831	11	2	5	11	4
	IS	913	4	0	2	4	2
	LU	36,736	101	7	45	101	56
	MG	7,818	22	0	18	22	4
	SP	28,098	72	6	30	72	42
	LUFact	1,737	4	1	2	4	2
	MolDyn	2,417	21	5	3	21	18
	Series	873	1	1	1	1	0
	SOR	771	4	4	4	4	2
	Matmult	818	1	1	1	1	0
	Crypt	1,567	3	1	1	3	1
	RayTracer	2,303	0	0	0	0	0
	MonteCarlo	2,252	2	1	2	2	0
	Pi	51	1	1	0	1	1
	Total		**306**	**35 (↓89%)**	**141 (↓54%)**	**306 (↓0%)**	**164 (↓46%)**
	Unresolved		0 (0%)				
PL	Histogram	44	3	3	0	3	3
	Pipeline	70	5	5	0	5	5
	Download	68	3	3	0	3	3
	Pipesort	50	3	3	0	3	3
	Prime	57	7	7	0	7	7
	Total		**21**	**21 (↓0%)**	**0 (↓100%)**	**21 (↓0%)**	**21 (↓0%)**
	Unresolved		0 (0%)				
MW	Fibonacci	55	1	1	1	1	1
	PiPrec	143	11	11	9	11	11
	Sudoku	349	24	24	19	19	23
	Total		**36**	**36 (↓0%)**	**29 (↓19%)**	**31 (↓14%)**	**35 (↓3%)**
	Unresolved		23 (64%)				
	Total	**130,696**	**395**	**117 (↓70%)**	**187 (↓53%)**	**389 (↓2%)**	**252 (↓36%)**
	Unresolved		39 (10%)				

Figure 8: Quantification of ordering problems over message passing models for various concurrent programming patterns [31]: Master Worker (MW), Pipeline (PL), Loop Parallelism (FL) and Event-based Coordination (EC).

5.1.2 Loop Parallelism

- (*+sync* and *+inorder* and *+isol*) together prevent all ordering problems in Loop Parallelism applications.
- Transitivity of in-order message delivery and processing in *+trans* does not prevent any ordering problems.

For this pattern, *+sync* prevents the most ordering problems (89%) followed by *+inorder* (54%) and *+isol* (46%) and *+trans* prevents none (0%). There is overlapping between models in prevention of ordering problems. That is, there are ordering problems that can be prevented by more than one model, as discussed in §5.2.

Loop Parallelism [31] usually involves an implicit barrier point [35] to synchronize all iterations of a loop before proceeding. *+sync* enables easy enforcement of such synchronization points. *+inorder* helps with ordering messages of different loops according to the appearance order of the loops in the program text. Extensive use of call by reference messages, to avoid copying of data, makes *+isol* important in the prevention of ordering problems. Absence of triangle patterns makes *+trans* the least beneficial model.

5.1.3 Pipeline

+inorder prevents all ordering problems in Pipeline applications. *+sync*, *+trans* and *+isol* prevent none.

In Pipeline [31] applications, each stage of the pipeline should process messages in the same order they are delivered which in turn makes *+inorder* the most important. The shared data between the pipeline stages is restricted to be a sequence (stream) and stage processes do not reuse the data after processing it. This in turn makes *+isol less important for Pipeline applications, which verifies the findings of previous work* [35]. Since a stage process does not synchronize with its subsequent stage processes, *+sync* is not important. Lack of triangle patterns renders *+trans* less important.

5.1.4 Master Worker

(*+inorder* and *+trans* and *+isol*) only prevent 36% of ordering problems in Master Worker. *+sync* prevents none.

In Master Worker [31], the master process usually uses different messages to initialize worker processes, assign work to them and shut them down. These messages should be processed by workers in the order in which they are sent which in turn makes *+inorder* more important. Unlike other patterns, the system graphs for these applications usually contain more triangle patterns which in turn make *+trans* important. *+isol* is less important because master and workers usually do not share data with each other.

5.1.5 Unresolved Ordering Problems

78% of ordering problems in Master Worker can be prevented by commutativity guarantees.

Figure 8 shows that there are ordering problems that cannot be prevented by any model and therefore remain unresolved. However, stronger guarantees such as commutativity of operations can prevent some of these problems. Two operations are commutative if they can be executed in any order without changing the outcome.

5.1.6 All Together

For all of the applications across all patterns, *+sync* prevents most of the message ordering problems (70%), followed by *+inorder* (53%) and *+isol* (36%) where *+trans* only prevents 2% of the problems.

+inorder prevents most of the ordering problems for Event-based Coordination, Pipeline and Master Worker applica-

tions whereas +*sync* prevents the most for Loop Parallelism. +*trans* prevents no ordering problems in Loop Parallelism and Pipeline and +*isol* prevents none in Event-based Coordination and Pipeline.

5.2 Overlapping of Models

There are ordering problems that could be prevented by *only* one model while there are others that could be prevented by multiple models. Figure 10 quantifies the relation between ordering problems and the overlapping of our models. The overlapping area of two models shows the number of ordering problems prevented by *either* model. Figure 10 suggests the following trends in overlapping.

5.2.1 Event-Based Coordination

- (+*inorder* and +*trans*) can prevent all ordering problems that +*sync* prevents in Event-based Coordination.
- More than half of the ordering problems in this pattern can *only* be prevented by +*inorder*.

This makes +*inorder* critical in preventing ordering problems in Event-based Coordination applications. Definition 3 defines a critical model for an application.

DEFINITION 3. ***(A critical model)***
A message passing model is critical to a message passing application, if there are ordering problems in the application that can be prevented only by that model.

5.2.2 Loop Parallelism

- (+*inorder* and +*isol*) can prevent all ordering problems that +*sync* can prevent in Loop Parallelism.
- Both +*inorder* and +*isol* are critical in preventing message ordering problems in this pattern.

More than 9% of message ordering problems in Loop Parallelism can be prevented only by +*inorder*. Similarly, less than 2% can be prevented only by +*isol*.

5.2.3 Pipeline

For Pipeline applications, all ordering problems are prevented by +*inorder* and there is no overlapping.

5.2.4 Master Worker

+*inorder*, +*trans* and +*isol* are critical to Master Worker.

+*inorder* is the most critical model because it prevents 54% of ordering problems that cannot be prevented by other models. +*trans* is more critical than +*isol* because it prevents 39% where +*isol* prevents 8% of ordering problems that cannot be prevented by other models.

5.2.5 All Together

- +*inorder* is critical to applications in all Event-based, Loop Parallelism, Pipeline and Master Worker patterns. +*isol* is critical to Loop Parallelism and Master Worker. +*trans* is critical to Master Worker only.
- +*sync* is not a critical model for any pattern.

5.3 Triggering of Ordering Problems

A message ordering problem that leads to a bug should be reproducible by a test case for debugging purposes. The complexity of such a test case is exponential in the number of processes it should control [30] and messages among them [25, 47]. For efficiency, it is necessary to be able to trigger a bug with minimum number of processes and messages. Figure 11 shows the minimum number of processes and messages, whose interleavings should be controlled to trigger ordering problems in each model. Figure 11 suggests the following trends.

5.3.1 Minimum Number of Processes

More than half of the ordering problems can be triggered by controlling the interleavings of only 2 processes.

5.3.2 Minimum Number of Messages

- More than three quarters of problems in +*trans*, *base* and +*isol* are triggered by only 2 messages.
- There are message ordering problems that can be triggered with only 1 message.

One message can trigger an ordering problem in a process A when A sends a message and the effects of processing that message in other processes conflicts with the rest of A.

5.3.3 All Together

Only 2 processes and 2 messages trigger 75% of ordering problems in +*isol* and 49% in +*trans* and *base*.

However, such a testing configuration only triggers 2% of ordering problems in +*inorder*. Triggering about half of problems in +*inorder* requires up to 3 processes and 3 messages and the other half needs even more.

5.4 Implications

Our observations could provide key insights as complementary guidelines to previous work [24, 30, 46, 47] for application developers, application verifiers and framework designers who engineer message passing software.

5.4.1 Application Developers

An application developer may use our findings in deciding: (1) how to tune the semantics of a message passing paradigm or (2) choose the proper paradigm with suitable semantics for their applications. For example:

(+inorder and +isol and +sync) for all Based on the observations in §5.1.6 and §5.2.5, the application developer may decide to implement their applications in a model whose semantics, by default or through semantic tuning, supports synchronous messaging, in-order delivery and processing, and data isolation. This is because such a model, (+*inorder* and +*isol* and +*sync*), prevents the majority (100%) of ordering problems and +*inorder*, +*isol* and +*sync* are all critical to one programming pattern or another.

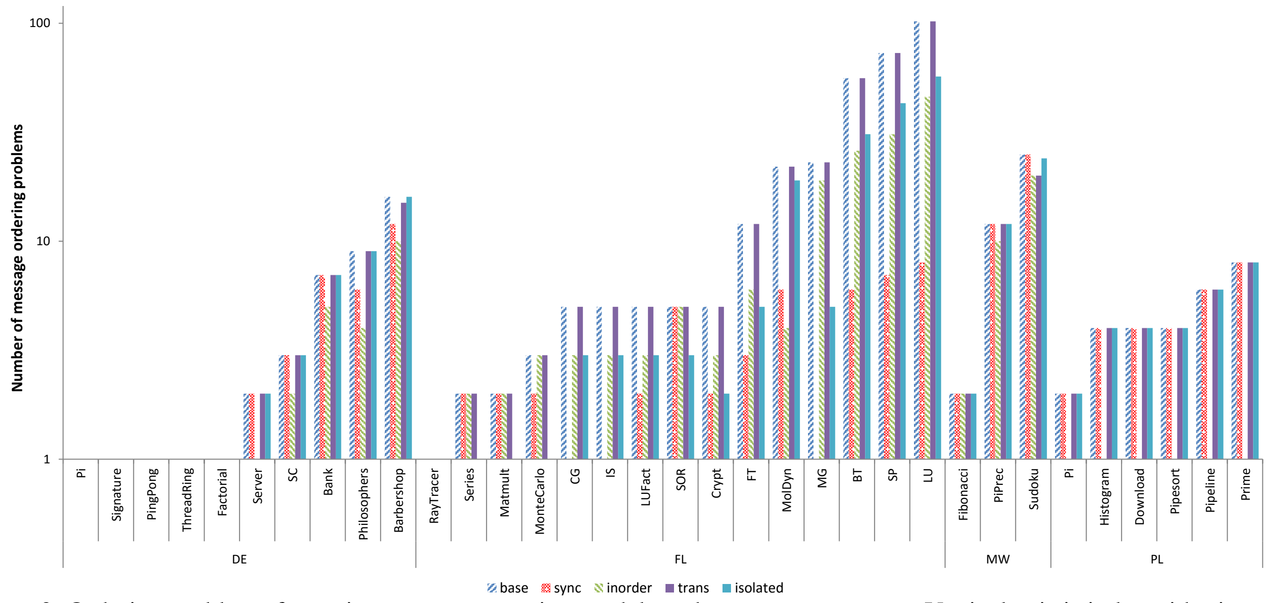
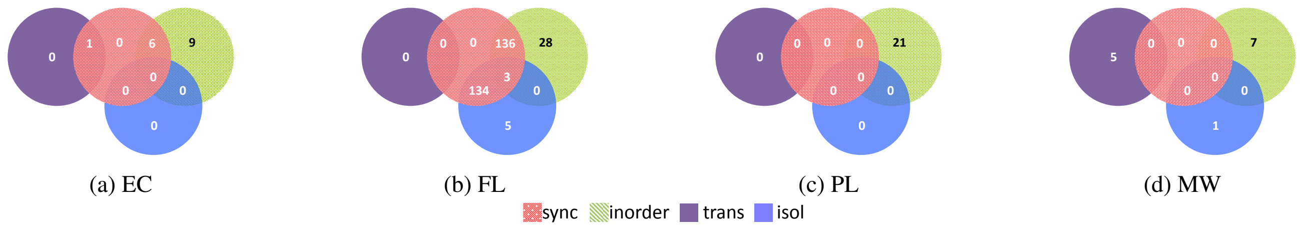


Figure 9: Ordering problems for various message passing models and concurrent patterns. Vertical axis is in logarithmic scale.



(a) EC (b) FL (c) PL (d) MW

Figure 10: Overlapping of various message passing models in preventing ordering problems.

Process / Messages	*base*				*+sync*				*+inorder*				*+trans*				*+isol*			
	2	3	4+	Total	2	3	4+	Total	2	3	4+	Total	2	3	4+	Total	2	3	4+	Total
1	8%	0%	0%	**8%**	3%	0%	0%	**3%**	17%	0%	0%	**17%**	8%	0%	0%	**8%**	0%	0%	0%	**0%**
2	49%	27%	0%	**76%**	43%	2%	0%	**44%**	2%	19%	0%	**20%**	49%	28%	0%	**77%**	75%	0%	0%	**75%**
3	0%	1%	0%	**1%**	0%	2%	0%	**2%**	0%	11%	0%	**11%**	0%	0%	0%	**0%**	0%	1%	0%	**1%**
4+	2%	6%	7%	**15%**	7%	21%	23%	**51%**	5%	37%	9%	**51%**	2%	6%	7%	**15%**	3%	10%	11%	**24%**
Total	**59%**	**34%**	**7%**		**52%**	**25%**	**23%**		**24%**	**67%**	**9%**		**60%**	**33%**	**7%**		**78%**	**11%**	**11%**	

Figure 11: Quantification of minimum number of processes and messages required to trigger ordering problems.

Panini [6, 37, 38] and JCoBox [40] by default support such a model and Akka [4] can be configured to support it.

+inorder for Pipeline Based on the observation in §5.1.3, the application developer may decide to tune the semantics of their model to only support in-order delivery and processing for their Pipeline applications and not the whole (+*inorder* and +*isol* and +*sync*). This is because +*inorder* prevents the majority (100%) of ordering problems while +*isol* and +*sync* prevent none for Pipeline.

5.5 Application Verifiers

An application verifier may use our findings in: (1) designing test cases with minimal complexity and (2) budgeting and focusing of test efforts. For example:

Minimal test cases Based on the observation in §5.3.3, the application verifier can design test cases with minimal complexity by controlling only 2 processes and 2 messages between them and test for three quarters of ordering problems, if the underlying model supports isolation. The complexity of a test case is exponential in the number of processes and messages it controls [30].

Budgeted test efforts Based on the observation in §5.1.2, for Loop Parallelism, the application verifier may decide to allocate twice the effort to test for ordering problems related to synchrony compared to isolation and almost no effort for transitivity. This is because +*sync* prevents 89% of ordering problems in this pattern compared to +*isol* that prevents 46%. However, based on the observation in §5.2.2, the verifier may refine his decision and decide to actually not test for synchrony and focus mostly on in-order delivery and processing and isolation. This is because (+*inorder* and +*isol*) can prevent all ordering problems that +*sync* prevents.

5.6 Framework Designers

A framework designer may use our findings in deciding which semantic features are more important than others in the design of their framework. For example:

***Critical* +inorder *and* +isol** Based on the observation in §5.2.5, the framework designer may decide to include both in-order message delivery and processing, and data isolation in their framework. This is because there are message ordering problems in Event-based Coordination, Loop Parallelism, Pipeline and Master Worker patterns that can be prevented *only* by +*inorder* and there are ordering problems in Loop Parallelism and Master Worker that can only be prevented by +*isol*.

+inorder *is important for session-based programming framework* The observation in §5.2.5 verifies and supports the decision of the framework designer in previous work [18] for supporting in-order message delivery and processing for the prevention of ordering problems in session-based message passing programming. A session structures a set of messages between a set of participating processes.

+sync *is important in Habanero framework* The observation in §5.1.2, verifies and supports the decision of the framework designer in the Habanero framework [20] to support synchronous messaging in their framework to enhance productivity of programmers for Loop Parallelism programs. Based on the observation in §5.2.5, the framework designer could alternatively decide to support the combination of in-order and isolation instead of message synchrony. This is because, (+*inorder* and +*isol*) can prevent all the ordering problems +*sync* can.

5.7 Modular Enforcement of Aliasing

Data isolation among processes is a simple form of modular enforcement of aliasing [33, 40] in which each process encapsulates and owns its data and does not share its data with other processes. With data isolation, data of a process can be accessed only through its message handlers. The observation in §5.3.3 shows that modular enforcement of aliasing can decrease the complexity of testing and test cases. This is because in a model with data isolation, three quarters of message ordering problems can be prevented by minimal test cases that control only 2 messages and 2 processes.

5.8 Threats to Validity

External validity The external validity of our study is limited by our choice of benchmark applications that are from either previous work or well-known concurrent benchmarks. Due to such a choice, we cannot claim that they form an exhaustive set of all typical message passing programs. Another threat to external validity of our study is that the same application is not implemented in various concurrent patterns and therefore there is an uneven distribution of ordering problems among applications of different patterns.

Internal validity The internal validity of our study is limited by our refactoring process to adapt message passing programs of previous work and multi-threaded benchmarks to Panini [37, 38] though our well-defined refactoring is designed to be mostly syntactic and as minimally intrusive as possible. Another threat to internal validity is that our analysis cannot detect manual implementations of synchronous messages using asynchrony [22], in-order delivery and processing and transitive in-order delivery.

6. Related Work

Sequential consistency Grace [8] proposes a transactional memory technique to enforce sequential semantics and avoid concurrency bugs for multi-threaded programs. Safe futures [49] and Asynchronous, Typed Events [29] provide a semblance of sequential semantics for multi-threaded programs. Lamport [23] proposes requirements for shared memory multiprocessor programs to guarantee sequential consistency. Qadeer [36] and Cain *et al.* [11] propose model checking techniques for sequential consistency. However, these works do not study the relation between ordering problem and semantic variations of message passing models.

Testing and model checking Lauterburg *et al.* [24] proposes Basset to explore possible interleaving of message passing programs. Sen and Agha [41] propose jCute to explore behaviors of message passing programs using concolic execution for test generation. Fredlund and Svensson propose McErlang [14] to model check distributed and fault tolerant Erlang programs. Tasharofi *et al.* [46] proposes Setac for testing Scala programs using user specified constraints on nondeterministic schedule of message exchanges; and proposes Bita [47] for testing using higher coverage scheduling. Bordini *et al.* [10] propose a translation from the message passing language AgentSpeak into Java to enable its model checking by Java PathFinder (JPF). Other previous work [7, 44] propose model checking techniques for distributed and networked systems. However, these works are mostly concerned about testing and model checking of message passing programs and not about the relation of ordering problem and semantic variations of message passing models.

7. Conclusion and Future Work

In this work we studied and quantified the relation between message ordering problems and semantic variations for the semantics of three criteria $\mathfrak{C}1$–$\mathfrak{C}3$ of message synchronization, message delivery and processing and sharing. We discussed implications of our findings for application developers, application verifiers and framework designers that engineer message passing paradigms. We also verified some of the findings of previous work. One avenue for future work is to study other problems of message passing programming in addition to ordering problems.

Acknowledgments

Authors are supported by NSF grants CCF-11-17937, CCF-14-23370 and CCF-08-46059.

References

[1] ActorFoundry: http://osl.cs.uiuc.edu/af/

[2] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)

[3] Agha, G., Frølund, S., Kim, W., Panwar, R., Patterson, A., Sturman, D.: Abstraction and modularity mechanisms for concurrent computing. Parallel & Distributed Technology: Systems & Applications, IEEE 1(2)

[4] Akka: http://akka.io/

[5] Armstrong, J.: Erlang. CACM'10 53(9)

[6] Bagherzadeh, M., Rajan, H.: Panini: A concurrent programming model for solving pervasive and oblivious interference. In: MODULARITY'15

[7] Barlas, E., Bultan, T.: Netstub: a framework for verification of distributed Java applications. In: ASE'07

[8] Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for C/C++. In: OOPSLA'09

[9] Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling: techniques for efficiently managing shared state. In: PLDI'11

[10] Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. AAMAS'06

[11] Cain, H.W., Lipasti, M.H.: Verifying sequential consistency using vector clocks. In: SPAA'02

[12] Csallner, C., Fegaras, L., Li, C.: New ideas track: Testing mapreduce-style programs. In: ESEC/FSE'11

[13] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL'05

[14] Fredlund, L.A., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: ICFP'07

[15] Frumkin, M., Schultz, M., Jin, H., Yan, J.: Implementation of the NAS Parallel Benchmarks in Java (2002)

[16] Haller, P., Odersky, M.: Event-based programming without inversion of control. In: JMLC'06

[17] Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. TCS'09 410(2-3)

[18] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL'08

[19] Houck, C.R., Agha, G.: HAL: A high-level actor language and its distributed implementation. In: ICPP '92

[20] Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. In: OOPSLA'12

[21] Jetlang: code.google.com/p/jetlang/

[22] Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: A comparative analysis. In: PPPJ '09

[23] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. CACM'78 21(7)

[24] Lauterburg, S., Dotta, M., Marinov, D., Agha, G.: A framework for state-space exploration of Java-based actor programs. In: ASE'09

[25] Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: FASE'10

[26] Lee, E.A.: The problem with threads. Computer'06 39(5)

[27] Lei, J., Qiu, Z.: Modular reasoning for message-passing programs. In: ICTAC'14

[28] Long, Y., Liu, Y.D., Rajan, H.: Intensional effect polymorphism. In: ECOOP '15

[29] Long, Y., Rajan, H.: A type-and-effect system for asynchronous, typed events. In: MODULARITY '16

[30] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: ASPLOS'08

[31] Mattson, T., Sanders, B., Berna, M.: Patterns for Parallelel Programming. Addison-Wesley Professional (2004)

[32] Negara, S., Karmani, R.K., Agha, G.: Inferring ownership transfer for efficient message passing. In: PPoPP'11

[33] Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: ECOOP'98

[34] Panini Web Site: http://www.paninij.org/

[35] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X.: The tao of parallelism in algorithms. In: PLDI'11

[36] Qadeer, S.: Verifying sequential consistency on shared-memory multiprocessors by model checking. IEEE Trans. Parallel Distrib. Syst.'03 14(8)

[37] Rajan, H.: Capsule-oriented programming. In: ICSE'15

[38] Rajan, H., Kautz, S.M., Lin, E., Kabala, S., Upadhyaya, G., Long, Y., Fernando, R., Szakács, L.: Capsule-oriented programming. Tech. Rep. 13-01, Iowa State U.

[39] Scala Actors Migration Guide: http://docs.scala-lang.org/overviews/core/actors-migration-guide.html

[40] Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: ECOOP'10

[41] Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: FASE'06

[42] Smith, L., Bull, J., Obdrizalek, J.: A parallel Java Grande benchmark suite. In: SC'01

[43] Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP'08

[44] Stoller, S.: Model-checking multi-threaded distributed Java programs. In: SPIN '00

[45] Sutter, H., Larus, J.: Software and the concurrency revolution. Queue'05 3(7)

[46] Tasharofi, S., Gligoric, M., Marinov, D., Ralph, J.: Setac: A framework for phased deterministic testing of Scala actor programs. In: The Scale Workshop'11

[47] Tasharofi, S., Pradel, M., Lin, Y., Johnson, R.E.: Bita: Coverage-guided, automatic testing of actor programs. In: ASE'13

[48] Upadhyaya, G., Rajan, H.: Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. In: OOPSLA'15

[49] Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: OOPSLA'05