IOWA STATE UNIVERSITY
Digital Repository

Computer Science Technical Reports | Computer Science

4-25-2014

Quantification of Sequential Consistency in Actor-like Systems: An Exploratory Study

Yuheng Long
Iowa State University, csgzlong@iastate.edu

Mehdi Bagherzadeh
Iowa State University, mbagherz@iastate.edu

Eric Lin
Iowa State University, eylin@iastate.edu

Ganesha Upadhyaya
Iowa State University, ganeshau@iastate.edu

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Software Engineering Commons

Recommended Citation
Long, Yuheng; Bagherzadeh, Mehdi; Lin, Eric; Upadhyaya, Ganesha; and Rajan, Hridesh, "Quantification of Sequential Consistency in Actor-like Systems: An Exploratory Study" (2014). *Computer Science Technical Reports*. 333.
http://lib.dr.iastate.edu/cs_techreports/333

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Quantification of Sequential Consistency in Actor-like Systems: An Exploratory Study

Yuheng Long Mehdi Bagherzadeh Eric Lin Ganesha Upadhyaya Hridesh Rajan
TR #14-03a
Initial Submission: March 31, 2014
Revised: April 25, 2014

Keywords: sequential consistency, actor-oriented programming

CR Categories:
D.1.3 [*Concurrent Programming*] Parallel programming
D.2.11 [*Software Architectures*] Languages, Patterns
D.3.3 [*Programming Languages*] Concurrent programming structures, Language Constructs and Features - Control structures

Copyright (c) 2014, Yuheng Long and Mehdi Bagherzadeh and Eric Lin and Ganesha Upadhyaya and Hridesh Rajan.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Quantification of Sequential Consistency in Actor-like Systems: An Exploratory Study

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

ABSTRACT

For sequentially-trained programmers, sequential consistency, i.e. program operations run in the same order as they appear in code, is the most intuitive consistency model to understand their programs. Recently variations of the actor model have been added to programming languages and libraries as a concurrency mechanism. Actor models, in general, do not guarantee sequential consistency. A surprising observation, studied in detail here using a large (130 KLOC) set of benchmarks, is that: the variation of the actor model supported by a language or library causes sequential inconsistencies exhibited by programs to vary greatly. Knowing the impact of these variations on sequential inconsistencies is important for focusing testing and verification efforts: for instance, if the variation supports in-order messaging then programs have 53% less sequential inconsistencies; or support for data isolation allows triggering of 75% of sequential inconsistencies by only controlling interleavings of 2 actors and 2 messages.

1. INTRODUCTION

Sequential consistency The advent of multicore processors requires sequentially-trained programmers to write concurrent programs [1]. For these programmers, sequential consistency is the most intuitive consistency model of memory when understanding their concurrent programs [2, 3]. A multiprocess program is sequentially consistent if its results could be produced on a single processor with a total ordering among its operations such that:

(i) the order of operations in each process is preserved as dictated by the program text; and

(ii) each read gets the last previously written value [3,4] according to the total order.

Sequential consistency is concerned about *(i)* the order of operations and *(ii)* their conflicting memory effects. Two memory operations conflict if they both access the same memory location, one of them is a write operation, and there is no order among them [5].

Since sequential consistency assumes that the operations of a program and their memory effects happen in the same order as in the program code, it helps decrease the conceptual gap between the static structure and the dynamic structure of a program for sequentially-trained programmers [6].

Actor-oriented programming The concurrency revolution has renewed interest in actor-oriented programming [7, 8] mainly because actors offer a flexible and scalable concurrent programming model. For example, SimpleDB for Amazon's Web Services and MochiWeb for Facebook's chat servers are programmed in the actor language of Erlang [9, 10]; or Guardian's web site uses the actor language and runtime of Akka [11, 12]. An actor has its own thread of control, manages its internal state and communicates with other actors via messages.

Concurrent actor programs are more overwhelming and more difficult to understand than sequential ones [1]. For these programs, sequential consistency creates the illusion of a sequential execution that sequentially-trained programmers are most familiar with and makes their understanding easier. Sequential consistency, in general, is not guaranteed by actors. Thus it is critical for sequentially-trained programmers to understand sequential consistency in the context of actors, if they choose actors for writing and testing of their concurrent programs.

To understand sequential consistency of actors, the previous definition of sequential consistency for multiprocess systems could be reused by mapping actors to processes [13]. Similar to multiprocess programs, sequential consistency of actor programs is concerned about the order of operations in actors as well as their conflicting memory effects.

The order of operations in an actor program is dependent on the semantics of its underlying actor model and is determined by:

- *Criterion (1)*: Message synchronization, as in asynchronous and synchronous messaging [14]; and
- *Criterion (2)*: Message delivery and processing semantics, as in non-deterministic or in-order delivery and processing;

Similarly, conflicts of memory effects of actors are determined by:

- *Criterion (3)*: Sharing semantics, as in data sharing or data isolation of memory effects of actors.

Diverse actor semantics Recently variations of the actor programming model [7, 8] have been added to programming languages and libraries as a concurrency mechanism. These variations have a diverse set of semantics for criteria *(1)*–*(3)* and consequently affect sequential consistency of their actor programs differently. For example for criterion *(1)*, Scala Actors [15], ActorFoundry [16], JCoBox [17], Kilim [13] and Panini [18] support both asynchronous and built-in synchronous request-reply messag-

ing whereas Erlang [9] and Standard actor model do not[1]. For *criterion (2)*, ActorFoundry messages are delivered and could be processed non-deterministically whereas Akka [11], Panini and JCoBox support in-order delivery and processing of messages. Finally, for criterion *(3)*, Standard actor model supports data isolation among actors whereas Scala Actors allow shared state among actors; ActorFoundry supports both.

PtolemyII [19], SALSA [20], Jetlang [21], JavAct [22], Ruby Stage [23] and Python Parley [24] are other examples of actor languages and frameworks with various semantics for these criteria.

Sequential consistency and actor models On one hand, the concurrency revolution requires sequentially-trained programmers to write concurrent programs with a choice of using actors as a scalable and popular concurrent programming model. Understanding concurrent programs is overwhelming for these programmers. On the other hand, sequential consistency makes understanding of concurrent programs easier for these programmers but is not generally supported by actors and their diverse set of semantics. This makes it critical to study and understand the relation between the semantics of an actor model and sequential consistency of its programs.

Testing Understanding such a relation can help focus testing and verification efforts. For instance, for a Pipeline actor program in an actor model with in-order messaging and synchronous messaging both, a programmer can decide where to focus testing by answering questions like: between in-order and synchronous, which one prevents more sequential inconsistencies? and to trigger inconsistencies, interleavings of how many actor instances and messages should be controlled by test cases?. Complexity of a test case is exponential in number of actor instances [25] and messages [26].

1.1 Contributions

In this paper, we quantify the relation of semantics of actor models and sequential (in)consistency of their programs, for 5 individual actor models and their combinations in 2 semantic spectrums. For each individual model, we study the minimum number of actor instances, with controlled interleavings, and the number of messages among them needed to trigger sequential inconsistencies. Finally we quantify the overlapping of individual actor models in preventing sequential inconsistencies.

Our benchmarks are adapted from benchmarks in previous work on actors, e.g. Basset [27], Habanero [28], Jetlang [21], and from well-known multi-threaded benchmarks, such as NAS Parallel Benchmarks [29] and parallel JavaGrande [30]. The are 34 benchmark applications with different sizes and various concurrent programming patterns [31] including: Master Worker, Loop Parallelism, Pipeline and Event Based Coordination.

Individual actor models We quantify the number of sequential inconsistencies for five individual actor models: *base*, +*sync*, +*inorder*, +*trans* and +*isol*, each of which focus on an individual criterion *(1)–(3)*. *base* is a model with asynchronous but no built-in synchronous messaging for criterion *(1)*, non-deterministic delivery and processing of messages for criterion *(2)*, and data sharing among actors for criterion *(3)*. +*sync* adds built-in synchronous messaging to *base*; +*inorder* adds in-order delivery and processing of messages to *base*; +*trans* adds transitive in-order delivery and processing to *base* and finally +*isol* adds data isolation.

Semantic spectrums We quantify sequential inconsistency for combinations of individual actor models using two semantic spectrums. The semantic spectrum (*base* $\leadsto$ *base*+*sync* $\leadsto$ *base*+*sync*+*inorder* $\leadsto$ *base*+*sync*+*inorder*+*trans*) combines various models in each point on the spectrum. It starts with *base*, adds synchronous messaging of +*sync* in the next point, in-order delivery and processing of +*inorder*, and finally transitive in-order delivery and processing of +*trans*. Two variations of this spectrum alternate in their support for data isolation and data sharing.

Actor instances and messages For testing and verification of actor programs, we need to know what conditions are needed to reliably trigger a sequential inconsistency. One condition that previous work studies in the context of multi-threaded programs is the number of threads [25]. For actor programs, these conditions are: number of actor instances, with controlled interleavings, and the number of messages these instances exchange. Controlling interleavings of a limited number of actor instances of a program, instead of all of its instances, can greatly reduce the testing complexity [25]. We quantify the minimum number of actor instances and messages needed to trigger sequential inconsistencies, for each individual actor model and their combinations.

1.2 Observations

Our study suggests that: semantic variations of actor models cause sequential inconsistencies of actor programs to vary significantly, per concurrent pattern of programs. The minimum numbers of actor instances and messages needed to trigger sequential inconsistencies also vary significantly per actor model.

To illustrate, we list a few of our findings below. Section 4 discusses all of our findings and their *implications* in detail.

Individual actor models

1. In-order messaging of +*inorder* model prevents 53% more sequential inconsistencies than a model without it.
2. Synchronous messaging of +*sync* prevents 89% of inconsistencies of Loop Parallelism programs and none for Pipeline;
3. For Pipeline, in-order messaging of +*inorder* prevents 100% of inconsistencies whereas synchronous messaging of +*sync* prevents none.

Semantic spectrums

4. A semantic spectrum prevents more sequential inconsistencies than any individual actor model and their combinations;
5. Data isolation prevents up to 15% more inconsistencies in various points of spectrum.

Overlapping of actor models

6. Synchronous messaging of +*sync* prevents all the inconsistencies that a combination of in-order messaging of +*inorder*, transitivity of +*trans* and data isolation of +*isol* does.

Number of actor instances & messages

7. Controlling only 2 actor instances and 2 messages trigger 75% of sequential inconsistencies in +*isol* whereas they trigger only 2% in +*inorder*;
8. Triggering 22% of inconsistencies in +*sync* require controlling more than 4 actor instances and 4 messages.

Implications Such observations are useful in focusing and reducing the complexity of testing and verification of actor programs. For example, for a Pipeline actor program in an actor model with support for in-order and synchronous messaging, based on observation *(3)*, one would devise more test cases to exercise various ordering of delivery and processing of messages and less for their synchrony; and based on observation *(7)* most test cases for in-order require controlling interleavings of more than 2 actors and 2 messages to trigger sequential inconsistencies.

[1] A synchronous message could manually be implemented using asynchronous messages [14] however, it may not be easy to detect statically. A built-in synchronous message is *syntactically* different from an asynchronous message and easy to detect.

1.3 Outline

Section 2 illustrates the relation of sequential (in)consistency and various semantics of the underlying actor models. Section 3 defines a happens-before [32] relation for our 5 actor models and unsafe interleavings of actor operations leading to sequential inconsistencies and discusses how sharing semantics causes different patterns of sequential inconsistencies. Section 4 presents our study setup and Section 5 discusses our observations and their implications in detail. Section 6 discusses related work and Section 7 concludes.

2. PROBLEMS

This section illustrates sequential consistency and its relation to the three criteria of message synchronization, message delivery and processing, and sharing semantics of the underlying actor model.

```
1  class Server extends Actor{
2   int val = 0;
3   @message void set(int v){ val = v; }
4   @message int get(){ return val; }
5  }
6  class Client extends Actor{
7   ActorName server;
8   Client(ActorName s){ server = s; }
9   @message void start(){
10   call(server, "set", 1);
11   int v = call(server, "get");
12   assert(v == 1);  // Φ
13  }
14 }
15 class Driver{
16  static void main(String[] args){
17   ActorName server, client;
18   server = createActor(Server.class);
19   client = createActor(Client.class, server);
20   call(client, "start");
21  }
22 }
```

Figure 1: The `Client` is sequentially consistent regarding the assertion Φ, on line 12, if the call messages on lines 10 and 11 are treated as synchronous messages; and is sequentially inconsistent otherwise, when they are asynchronous. Also, it is sequentially consistent with in-order delivery and processing of messages and inconsistent with non-deterministic delivery and processing.

Figure 1 shows a simplified client-server example, adapted from previous work [27] written in ActorFoundry [16], which is a JVM-based framework for actor-oriented programming. For message synchronization, i.e. *criterion (1)*, ActorFoundry supports both asynchronous `send` and built-in synchronous request-reply `call` messaging, which is blocking; for message delivery and processing semantics, i.e. *criterion (2)*, it supports non-deterministic delivery and processing as well as programmer-specified processing of the messages [14]; and for sharing semantics, i.e. *criterion (3)*, it guarantees data isolation among actors by its call-by-value messaging or relies on the programmer when using call-by-reference.

In ActorFoundry, actors are declared by classes that extend the `Actor` class, e.g. `Client` actor on lines 6–14, and are instantiated using `createActor`, which returns actor instances of type `ActorName`, e.g. actor instance `client` on line 19. *Message handlers* are methods of actor classes marked with `@message` annotations, e.g. `set` on lines 3 is a message handler for set messages sent to `Server`.

In the client-server example of Figure 1, the `Server` actor keeps track of a variable `val` and provides access to it using two message handlers `set` and `get`, which respectively set and return the value of the variable. The `Client` actor, sets the value of `val` to 1 by sending the server a set message, on line 10, reads its value by sending a get message, on line 11, and finally checks if `val` is actually set to 1 using the assertion Φ, on line 12. The `Driver`, the program's entry point, creates client and server actor instances, on lines 18–19, and initiates the execution of the client by sending it a `start` message, on line 20. To send a message, using `call` or `send`, the name of the receiving actor, message name and the parameters for the message handler are necessary, e.g. on line 10, `server` is the name of the receiving actor, `"set"` is the name of the message and 1 is the parameter required by the handler of the set message in the server actor.

Criterion (1): Message synchronization To illustrate the relation of sequential consistency and message synchronization, we assume two alternative semantics for `call`: asynchronous and synchronous messaging[2]. Unlike synchronous messages which are blocking, asynchronous messages are non-blocking.

With synchronous blocking semantics for call messages, the example of Figure 1 is sequentially consistent and the assertion Φ holds. This is true because the blocking semantics of call messages ensure that the set message, on line 10, is actually processed by the server before the get message, on line 11, as suggested by the program text. The set and get messages are both concerned about the variable `val` in the server and their processing in the server could cause sequential inconsistencies, if there is no order among them or the order is not consistent with their appearance order in the program text (see the definition of sequential consistency in Section 1).

However, the example of Figure 1 will not be sequentially consistent if the call messages are not semantically synchronous. This is because the set message, that appears before the get message in the program text, may actually be processed by the server after the get message, especially if the messages are processed in a non-deterministic order. This in turn means that the value of the variable `v` may not necessarily be equal to 1 and thus the assertion Φ could be violated. The assertion Φ is a representative of sequential consistency in Figure 1, i.e. it holds when the client server example of Figure 1 is sequentially consistent and does not hold otherwise.

Criterion (2): Message delivery and processing semantics To illustrate the relation of sequential consistency and message delivery and processing semantics, we assume two alternative semantics for message delivery and processing: non-deterministic and in-order. In in-order delivery, two messages sent from one actor to another, with no intermediate actor in the middle, are guaranteed to be delivered in the same order that they are sent; and in in-order processing, the messages are processed in the same order they are delivered. In non-deterministic, there is no guarantee on the delivery or processing order of the messages.

With in-order delivery and processing of messages, the example of Figure 1 is sequentially consistent, because the set and get messages are delivered and processed in the same order they appear in the text, i.e. set before get. This is true even when call messages are treated as asynchronous messages. However, with non-deterministic delivery and processing and asynchronous call messages, these two messages could be delivered to the server and processed in any arbitrary order, including the order in which the get message is processed before the set message. This in turn makes the example sequentially inconsistent and violates Φ.

Criterion (3): Sharing semantics To illustrate the relation of sequential consistency and sharing, we assume two alternative semantics for sharing: data sharing among actors and data isolation.

In ActorFoundry, an actor cannot directly access the internal state of another actor. This in turn means that data sharing among actors can only happen through messaging [33]. Consequently, call-by-value messaging guarantees data isolation whereas call-by-reference messaging allows data sharing among actors. Figure 2 shows a modified version of the previous client server example in which the server keeps track of an object `val`, on line 3, instead of

[2] This is for illustration purposes only, otherwise we treat `call` messages as synchronous and `send` messages as asynchronous as specified by the semantics of ActorFoundry.

its primitive counterpart in Figure 1. The omitted client and driver code remains the same.

```
1  class Value{ int num; }
2  class Server extends Actor{
3   Value val;
4   @message void set(int v){ val.num = v; }
5   @message void init(Value v){ val = v; }
6  }
7  class Client extends Actor{
8   ActorName server;
9   Client(ActorName s){ server = s; }
10  @message void start(){
11   Value val = new Value();
12   call(server, "init", val);
13   call(server, "set", 1);
14   val.num = 2;
15   assert(val.num == 2); // Φ
16  }
17 }
```

Figure 2: The `Client` is sequentially consistent, regarding the assertion Φ, on line 12, if the client and server actors are isolated and is sequentially inconsistent otherwise, when they share data.

With data sharing among actors, the example of Figure 2 is sequentially inconsistent, especially if the call messages are asynchronous and messages are delivered and processed non-deterministically. This is because the call-by-reference semantics of the initialization call, on line 12, allows the client object `val` to be shared between the client and the server, and the asynchronous semantics of call messages allows the assignment, on line 14, to run before the processing of the set message on the server, on line 13. This in turn could result in values, e.g. 1, for the variable `val.num` and thus violate the assertion Φ, on line 15.

However, with data isolation among actors, the example of Figure 2 is sequentially consistent, with regard to the assertion Φ. This is because the call-by-value semantics, especially for the initialization message, on line 12, avoids sharing of `val` between the client and the server by sending a deep copy of `val` to the server.

Summary As illustrated, an actor program, such as the client-server examples of Figure 1 and Figure 2, can be sequentially consistent or inconsistent, or in more general terms have a varying number of sequential inconsistencies depending on the semantics of its underlying actor model, regarding criteria *(1)–(3)* of message synchronization, message delivery and processing and sharing.

3. SEQUENTIAL CONSISTENCY FOR ACTOR MODELS

This section discusses our static analysis of an actor program for sequential inconsistencies. It defines a happens-before [32] relation among actor operations for actor models *base*, *+sync*, *+inorder*, *+trans* and *+isol*; It also defines unsafe interleavings of actor operations that could lead to sequential inconsistencies. Finally this section discusses how sharing semantics among actors could lead to different patterns of sequential inconsistencies.

Our static analysis of an actor program for sequential inconsistencies has two phases:

- construction of a system graph, that encodes message exchanges among actors, and a conservative approximation of their read and write memory effects; and
- analyzing the system graph, for sequential inconsistencies, for various options of criteria *(1)–(3)*; their various combinations and number of actors and messages involved.

The system graph for an actor program is simply an alternative representation of the program that makes it easier to study its sequential consistency by encoding actor instances of the program, their message exchanges, types of messages exchanged among actors, as in asynchronous or synchronous, and their memory effects. After the construction of the system graph of a program, the analysis looks for unsafe interleaving of actor operations, which could lead to sequential inconsistencies, for each individual actor model.

Although the conversion of an actor program to its corresponding system graph is dependent on the syntax of its actor language, the system graph does not encode the semantics of the underlying actor model and thus is *independent* from the actor programming language. In this section we focus on the analysis of the system graph for sequential inconsistencies. Section 4 discusses conversion of actor programs to their representative system graphs, for the specific actor language of Panini [18].

3.1 Unsafe Interleavings

Definition 1 defines an unsafe interleaving of actor operations, which in turn could lead to a sequential inconsistency.

DEFINITION 1. *(Unsafe interleavings of actor operations) Let op_1 and op_2 be operations in the program text of an actor instance A such that op_1 appears before op_2 in the program text and these operations send messages to other actors, directly or indirectly, causing the execution of two other operations op'_1 and op'_2; Then op_1 and op_2 compose an unsafe interleaving if:*

- *op'_1 and op'_2 are concurrent, i.e. there does not exist a happens-before [32] ordering relation between them; and*
- *op'_1 and op'_2 have conflicting memory effects.*

Two memory effects conflict if they both are concerned about the same memory location and at least one of them is a write effect [5].

3.2 Happens-Before Relation

A happens-before relation defines an order among operations of an actor program. Definition 2 defines the happens-before relation for our 5 actor models, discussed in Section 1. In this definition sending of a message, is broken into $MsgSend$ and $MsgReturn$ operations and the processing of the message is broken into $MsgStart$ and $MsgEnd$.

DEFINITION 2. *(Happens-before relation $\prec$) Let $Op(o, i, A)$ be the operation o at the position i in the program text of an actor instance A. Let $MsgSend(m, A, A')$ be sending of the message m from the actor instance A to A', $MsgStart(m, A, A')$ and $MsgEnd(m, A, A')$ be the start and end of the processing of the message m in A' sent from A, and $MsgReturn(m, A, A')$ be the returning to A from the processing of the message m in A'; and $Handler(m, A')$ be the handler of message m sent to A'. Let $\prec$ denote the happens before relation.*

Then in the base *model:*

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

For synchronous messaging in +sync *model:*

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

For in-order delivery and processing of messages in +inorder *model:*

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

And finally for transitive in-order delivery and processing in the +trans *model:*

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

The happens-before relation $\prec$ is transitively closed [34].

Definition 2 says that in *base* model, *(1)* an operation happens-before another operation if the former appears before the latter in the program text of an actor; *(2)* sending of a message in the sender actor happens-before the start of its processing in the receiver actor and *(3)* sending of the message happens-before its returning; *(4)* start of the processing of a message also happens-before the end of its processing, and *(5)* start of the processing of the message happens-before any operation in the message handler of the message. Figure 1 illustrates the message handler `set` in the server actor, line 3, that handles set messages sent from the client actor.

The *+sync* model adds the synchronous messaging to *base*. Breaking down the message into its sending and processing is especially useful to model happens-before relation of synchronous messages. Definition 2 says that in *+sync (6)* for a synchronous message, unlike asynchronous messages, 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. It is worth to note that, the happens-before relations of *base* are all included in the happens-before relations of *+sync*, as well as other models, which are built on top of it.

The *+inorder* model adds in-order delivery and processing of the messages to the *base*. Definition 2 says that in *+inorder (7)* for two messages sent to another actor *directly*, with no other actor in the middle, the first message is delivered first and processed first and thus the end of its processing happens-before the start of the processing of the second message. The *+trans* model adds transitivity of the delivery to the *+inorder*. Definition 2 says that in *+trans* for two messages sent to another actor in which the first message is sent directly and the second message is sent *indirectly* through one or more actors in the middle, the first message is delivered and processed before the second message [35]. The indirect sending of one of the messages is critical to transitivity of in-order. Figure 3 shows the happens-before relation for the client server example of Figure 1, for two models *base* and *+sync*.

3.3 Analysis of Sequential Inconsistencies

Our analysis statically infers a happens-before relation among actor operations, as defined in Definition 2, and finds unsafe interleavings of actor operations based on their happens-before relations. For each individual actor model, the analysis uses its happens-before relations as defined in Definition 1. To decide if memory effects of operations conflict, the analysis takes into account the sharing semantics of actors in the underlying actor model.

To illustrate the analysis, consider the client-server example of Figure 1. Figure 3 shows the happens-before relation of this example for two models *base* and *+sync*. In *base*, there is no happens-before relation between the set and get server operations on `val` and thus they are concurrent. Since these two operations write and read the value of the same location `val` they are conflicting and thus form an unsafe interleaving that leads to a sequential inconsistency. However, in *+sync* there is a happens-before relation between the set and get operations on `val` which in turn means there

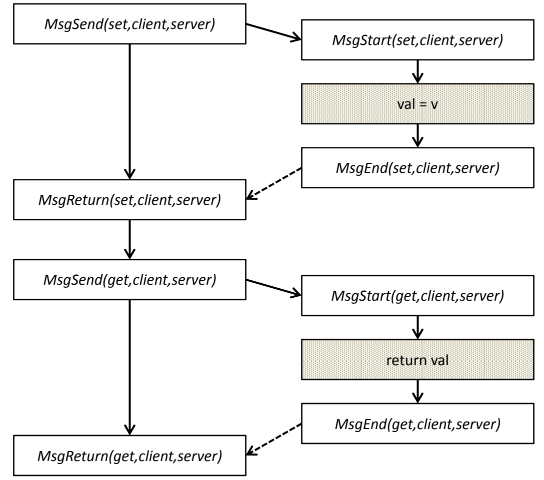
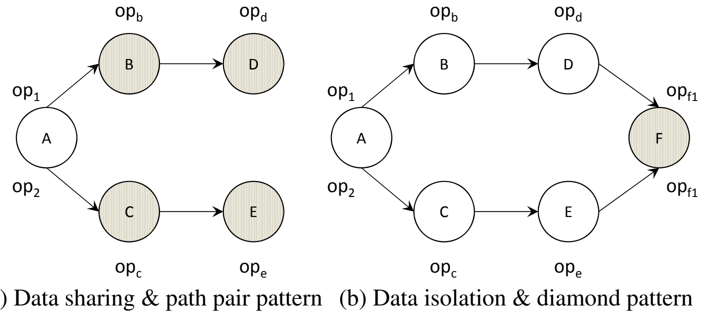


Figure 3: Happens-before relations $\prec$ for the example of Figure 1. Solid arrows show the happens-before relation for *base* model. Dashed arrows and solid arrows together show the happens-before relation for *+sync*. In *base*, there is no happens-before relation between the read and write server's operations on `val`, in grey boxes.

is no unsafe interleaving of set and get operations and consequently no sequential inconsistencies.

3.4 Patterns of Sequential Inconsistencies & Sharing Semantics

Sharing semantics of the underlying actor model causes different patterns for unsafe interleaving of actor operations, namely: *(i)* pair path pattern and *(ii)* diamond pattern. Our analysis looks for different patterns of unsafe interleavings of actor operations, depending on the sharing semantics of the underlying actor model. Figure 4 shows these patterns of unsafe interleavings.



(a) Data sharing & path pair pattern (b) Data isolation & diamond pattern

Figure 4: Patterns of unsafe interleavings and sharing semantics. Operations of shaded actors could cause sequential inconsistencies.

With data sharing among actors, the analysis looks for *pair path pattern*. In this pattern, if two operations op_1 and op_2 in an actor instance A start two message exchange paths among other actor instances, then *any* pair of operations op'_1 and op'_2 on these two paths, written as (op'_1, op'_2), could form an unsafe interleaving, provided there is no happens-before relation between them and their memory effects conflict. Figure 4(a) illustrates this situation in which op_1 and op_2 in the actor instance A send out messages to actor instances B and C, causing the operations op_b and op_c to execute, which in turn send messages to D and E, causing operations op_d and op_e to execute. The operations op_1 and op_2 start message exchange paths $op_b \rightarrow ob_d$ and $op_c \rightarrow ob_e$. With the underlying actor model support for data sharing among

actors, any pairs of operations on these two paths, i.e. any pairs in the set $\{(op_b, op_c), (op_b, op_e), (op_d, op_c), (op_d, op_e)\}$, could cause sequential inconsistencies if there is no happens-before relation among them and their effects conflict.

Unlike looking for path pair patterns in actor models with data sharing among actors, for actor models with data isolation, the analysis looks for *diamond pattern*. In this pattern, for two operations op_1 and op_2 in an actor instance that start two message exchange path, *at most one pair* of operations (op'_1, op'_2) on these two path could form an unsafe interleaving, provided that these two path lead to the same actor instance with operations op'_1 and op'_2 and there is no happens-before relation between these operations and their memory effects conflict. Figure 4 (b) illustrates this situation in which the data isolation semantics guarantees that the effects of any pairs of operations in $\{(op_b, op_c), (op_b, op_e), (op_d, op_c), (op_d, op_e)\}$ are isolated and thus do not cause any sequential inconsistency. The only pair of operations that could cause a sequential inconsistency is (op_{f1}, op_{f2}) which are in the same actor F. Interestingly, the path pair pattern does not cause any sequential inconsistency if the underlying actor model supports data isolation.

To check for sequential inconsistencies in Figure 4(a) with data sharing, all the pairs of operations on the two message exchange paths with no happens-before relation among their operations must be computed and checked for conflicts. In contrast to check for sequential inconsistencies in Figure 4(b) only the effects of the pair (op_{f1}, op_{f2}) should be computed if there is no happens-before relation among them. With data isolation among actors, to check for sequential inconsistencies in Figure 4(a) there is no computation of happens-before relations and effects of operations for conflict required, since it does not form a diamond.

4. EVALUATION

In this section, we quantify and study the relation of sequential consistency and the semantics of the underlying actor model regarding individual and combined effects of *criterion (1)* message synchronization, *criterion (2)* message delivery and processing semantics and *criterion (3)* sharing semantics among actors. We also quantify the minimum number of actor instances and messages that their interleaving should be controlled to reliably trigger sequential inconsistencies. Such quantifications can help in focusing testing and verification efforts of actor programs.

4.1 Study Setup

To look at the relation of sequential (in)consistency and the number of actor instance and messages involved with criteria *(1)–(3) individually*, we consider the following 5 variations of actor models:

1. *base*: the model has asynchronous messaging, no built-in synchronous messaging, non-deterministic delivery and processing of messages and data sharing among actors;
2. *+sync*: adds built-in blocking synchronous messaging to *base*, in addition to its asynchronous messaging;
3. *+inorder*: adds in-order delivery and processing of messages to *base*;
4. *+trans*: adds transitive in-order delivery and in-order processing of messages to *base*; and
5. *+isol*: adds data isolation among actors to the *base* model.

We also look at *combinations* of these models on various points of semantic spectrums. A semantics spectrum starts by *base*, adds synchronous messaging of *+sync* in the next point of the spectrum, followed by the addition of in-order message delivery and processing of *+inorder*, and finally adds up transitive in-order delivery of *+trans*. A semantic spectrum adds up happens-before relations, as defined in Definition 2, from one point to the next point in the spectrum. We consider two variations of such spectrum that alternate on their sharing semantics among actors:

6. (*base* $\rightsquigarrow$ *base+sync* $\rightsquigarrow$ *base+sync+inorder* $\rightsquigarrow$ *base+sync+inorder+trans*) with data sharing.
7. (*base* $\rightsquigarrow$ *base+sync* $\rightsquigarrow$ *base+sync+inorder* $\rightsquigarrow$ *base+sync+inorder+trans*) with data isolation.

We refactor our benchmark applications to actor programs in the actor language of Panini [18] and then convert these programs into their corresponding system graphs. System graphs allow analysis of these programs for sequential inconsistencies, using the analysis discussed in Section 3, independent of their implementation language of Panini. The initial set of sequential inconsistencies computed using our analysis is manually verified and its false positives are disposed of.

Panini actor language Panini proposes capsule-oriented programming [18] in which capsules, similar to actors, have their own thread of control, and communicate via asynchronous or built-in synchronous messages. Capsule methods are similar to message handlers in actors and sending of messages are modeled as invocations of capsule methods. A Panini program is a set of capsule declarations and a system declaration that declares capsule instances, and its execution starts with `run` methods of its capsule instances.

Figure 5 shows the client-server example of Figure 1 implemented in Panini. The figure shows the declaration of capsules `Server`, on lines 1–5, and `Client`, on lines 6–12. It also shows the system declaration, on lines 14–18 with capsule instance declarations for `client`, on line 16 and `server`, on line 15. Line 17 connects the client and servers instances.

System graph construction Conversion of a Panini program to its corresponding system graph and especially computation of its actor instances and their message exchanges, is rather straightforward. This is because 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. To compute the memory effects of actors, we use a sound alias analysis technique based on object graphs [36], in which two object graphs alias each other, if there exist objects in the graphs that alias each other.

```
1  capsule Server {
2   int val = 0;
3   void set(int v) { val = v; }
4   int get() { return val; }
5  }
6  capsule Client(Server server) {
7   void work() {
8    server.set(1);
9    int v = server.get();
10   assert(v == 1);
11  }
12 }
```

```
13 capsule Driver {
14  design {
15   Server server;
16   Client client;
17   client(server);
18  }
19  void run() {
20   client.work();
21  }
22 }
```

Figure 5: The client-server example of Figure 1 written in the actor language of Panini [18].

Benchmarks Figure 6 shows the list of 34 small to large size benchmarks application used in our study, that constitute more than 130,000 lines of code. The benchmark applications are adapted and refactored to Panini [18] from: previous work, including Basset [27], Habanero [28], Jetlang [21]; well-known benchmarks, including NAS Parallel Benchmarks [29] and parallel JavaGrande [30]; and examples shipped with the Panini compiler [37]. The benchmarks use a variety of parallel programming patterns [31] including

Master Worker (MW), Loop Parallelism (FL), Pipeline (PL) and Event Based Coordination (EC).

Refactoring to Panini Refactoring of benchmarks to Panini programs follows a very strict set of non-intrusive, mostly syntax related, steps. For multi-threaded programs these steps are: *(i)* refactor thread objects to capsule instances; *(ii)* refactor synchronized methods and blocks to capsule methods; *(iii)* create capsule fields for top-level class instances. For actor programs, not implemented in Panini, these steps are: *(i)* refactor actors to capsules *(ii)* refactor actor message handlers to capsule methods and message sends to invocation of capsule methods; *(iii)* create actor instances of the program in its system declaration.

5. OBSERVATIONS & IMPLICATIONS

In this section we discuss our observations for *(i)* the relation of sequential (in)consistency and *(ii)* the number of actor instance and messages involved with our 5 individual actor models and their combinations into 2 spectrums. We also discuss how these observations could be used in focusing testing and verification efforts.

5.1 Individual Actor Models

Figure 6 and its bar chart representation in Figure 7 show the number of sequential inconsistencies for 5 actor models of *base*, *+sync*, *+inorder*, *+trans* and *+isol* for benchmark applications per concurrent patterns. Figure 6 shows the following trends in actor programs of various patterns.

5.1.1 Event Based Coordination

For applications of this pattern, *+inorder* model prevents the most number of sequential inconsistencies (47%); and *+isol* (0%) and *+trans* (3%) prevent the least.

- This implies that for testing of Event Based Coordination actor programs, one would need to put more focus on exercising various ordering of delivery and processing of messages for in-order and less for data isolation, transitivity of messages and even their synchrony. Such testing efforts could be proportionally budgeted based on percentage share of each criteria in prevention of sequential inconsistencies, e.g. 47% for in-order versus 3% of transitivity.

In-order messaging of *+inorder* prevents the most number of sequential inconsistencies, because Event Based Coordination [31] applications, usually involve multiple iterations where in each iteration, the same set of messages is exchanged among the same set of actor instances; and the processing of messages of one iteration should be ordered before the messages of its subsequent iterations. In-order messaging guarantees such ordering for each iteration. Synchronous messaging of *+sync* helps with sequential consistency of check-then-act idiom [38], in Barbershop and Philosopher applications, where it ensures that a check message blocks the execution of its act message until the check message is processed and returns its result.

Data isolation of *+isol* prevents the least number of inconsistencies, because programmers manually enforce data isolation or messages exchanged among actors are composed of primitive values. Transitivity of *+trans* is not very important too, because system graphs of these applications, except Barbershop, do not include *triangle patterns*. In a triangle pattern, an actor instance A sends a message to actor instance A' directly and sends another message to A' indirectly through one or more intermediate actors. Triangle patterns are beneficiaries of transitivity of message delivery [35].

	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 6: Quantification of sequential inconsistencies over individual actor models and their prevention of sequential inconsistencies percentage-wise. Programs are of concurrent programming patterns [31] of (MW): Master Worker, (PL): Pipeline, (FL): Loop Parallelism and (EC): Event Based Coordination.

5.1.2 Loop Parallelism

For this pattern, *+sync* prevents the most number of sequential inconsistencies (89%) followed by *+inorder* (54%) and *+isol* (46%) and *+trans* prevents none (0%).

- The implication here is that for Loop Parallelism actor programs less testing effort is required for transitivity; and maybe twice the effort is needed for testing synchrony compared to isolation because there are almost twice the number of sequential inconsistencies causes by absence of the former compared to the latter.

Loop Parallelism [31] applications usually involve an implicit synchronization point (barrier synchronization [39]), that synchronizes all iterations of a loop before proceeding. The synchronous

messaging of *+sync* enables easy enforcement of such synchronization points. In-order messaging of *+inorder* helps with ordering messages of different loops according to the order of appearance of the loops in the program text. Extensive use of call-by-reference messages, to avoid copying of data, in turn makes *+isol* important in the prevention of sequential inconsistencies. System graphs for these applications do not contain triangle patterns and thus *+trans* is the least beneficial to these applications.

5.1.3 Pipeline

For this pattern, *+inorder* prevents all sequential inconsistencies (100%) and *+sync*, *+trans* and *+isol* prevent none (0%) with the implication being that the focus of testing should mostly be on in-order delivery and processing of the messages.

In Pipeline [31] applications, each stage of the pipeline should processes messages in the same order they are delivered which in turn makes *+inorder* the most important for these applications. The shared data between the pipeline stages is restricted to be a sequence (stream) and stage actors do not reuse the data after they processes it. This in turn makes *+isol* less important for pipeline applications which also corroborates findings of previous work [39]. Since a stage actor does not synchronize with its subsequent stage actors, then *+sync* is not very important too. Finally the lack of triangle patterns in linear system graphs of these applications renders *+trans* less important.

5.1.4 Master Worker

For this pattern, *+inorder* prevents the most number of sequential inconsistencies (19%), followed by *+trans* (14%); and *+sync* (0%) and *+isol* (3%) prevent the least. The implication being that testing of in-order messaging and its transitivity may need the same amount of effort.

In Master Worker [31] pattern, usually the master actor initializes worker actors, assigns work to them and shuts them down using different kinds of messages. These messages should be processed by the workers in the same order 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 makes *+trans* important. *+isol* is less important because master and worker actors do not share data with each other.

5.1.5 Unresolved sequential inconsistencies

Figure 6 shows that there are sequential inconsistencies that cannot be prevented by any individual models *base*, *+sync*, *+inorder*, *+trans* and *+isol*, i.e. they are unresolved. Interestingly, combination of these models into semantic spectrums, in Section 5.4, does not prevent these inconsistencies either. Stronger guarantees such as commutativity of operations can prevent some of these inconsistencies, e.g. in Master Worker applications, 78% of sequential inconsistencies could be prevented by commutativity guarantees. Two operations are commutative, if they can be executed in any order without affecting the outcome of their execution [40].

5.1.6 Summary

+inorder prevents the most number of sequential inconsistencies for Event Based Coordination, Pipeline and Master Worker applications whereas *+sync* prevents the most for Loop Parallelism. *+trans* prevents no sequential inconsistencies in Loop Parallelism and Pipeline and *+isol* prevents none in Event Based Coordination and Pipeline. These observations can be used to appropriately budget testing and verification efforts and resources and focus on one criteria more than the other when testing for inconsistencies.

5.2 Number of Actor Instances and Messages

Figure 8 shows the *minimum* number of actor instances and their messages, whose interleavings should be controlled to reliably trigger sequential inconsistencies.

The implication is that, for testing, one would use these numbers to control only a limited number of actor instances and their messages, instead of controlling all actor instance and messages in an actor program, and thus reduce the complexity of testing. The complexity of a test case is exponential in the number of actor instances it should control their interleavings [25] and the number of messages among them [26]. Figure 8 shows the number of actor instances and messages for each individual model. Similar quantifications for each individual model per concurrent programming patterns, as well as for semantic spectrums are excluded for space reasons and can be found in Section 8.

Figure 8 suggests following trends for individual actor models.

5.2.1 Number of Actor Instances

78% of sequential inconsistencies of actor programs in *+isol* can be triggered by controlling the interleavings of only 2 actor instances. More generally, controlling the interleavings of only 2 actor instances can trigger more than half of the inconsistencies in *+isol* (78%), *+trans* (60%), *base* (59%) and *+sync* (52%).

- The implication is that test cases for actor programs of these models can reliably trigger more than half of their sequential inconsistencies with *minimal* complexity, i.e. controlling the interleavings of only 2 actors.

5.2.2 Number of Messages

For the number of messages, test cases triggering sequential inconsistencies need only 2 messages to reveal more than three quarters of sequential inconsistencies in *+trans* (77%), *base* (76%) and *+isol* (75%). In contrast, using only one message, triggers 17% of inconsistencies in *+inorder* and only 8% in *base*, 3% in *+sync* and none in *+isol*. Inconsistencies triggered using one message are ones in which an actor A sends out a message and effects of processing that message by other actors conflicts with the rest of A.

- The implication here is that, using only one message to have minimal complexity, does not trigger the majority of sequential inconsistencies in any of the actor models.

5.2.3 Combined Actors and Messages

Using only 2 actor instances and 2 messages, 75% of sequential inconsistencies in *+isol* and 49% in *+trans* and *base*, are triggered. In contrast, such combination of actor instances and messages, only triggers 2% of inconsistencies in *+inorder*. Triggering about half of inconsistencies in *+inorder* requires up to 3 actor instances and 3 messages, the other half needs more actor instances and messages.

This implies that for the same amount of sequential inconsistencies, test cases in *+inorder* are more complex than test cases for *+isol*, *+trans* or *base*, mainly because they need more actor instances and messages to trigger a sequential inconsistency.

5.3 Overlapping of Actor Models

Another observation is that there are sequential inconsistencies that could be prevented using *only* one model, making the model critical for their prevention, while there are others that could be prevented in more than one model. A model is critical to an actor program, if there are inconsistencies in the program that can only be prevented using that model and not other models. Figure 6 contains both kinds of these sequential inconsistencies.

Figure 9 illustrates overlapping of the individual actor models in prevention of sequential inconsistencies quantified in Figure 6.

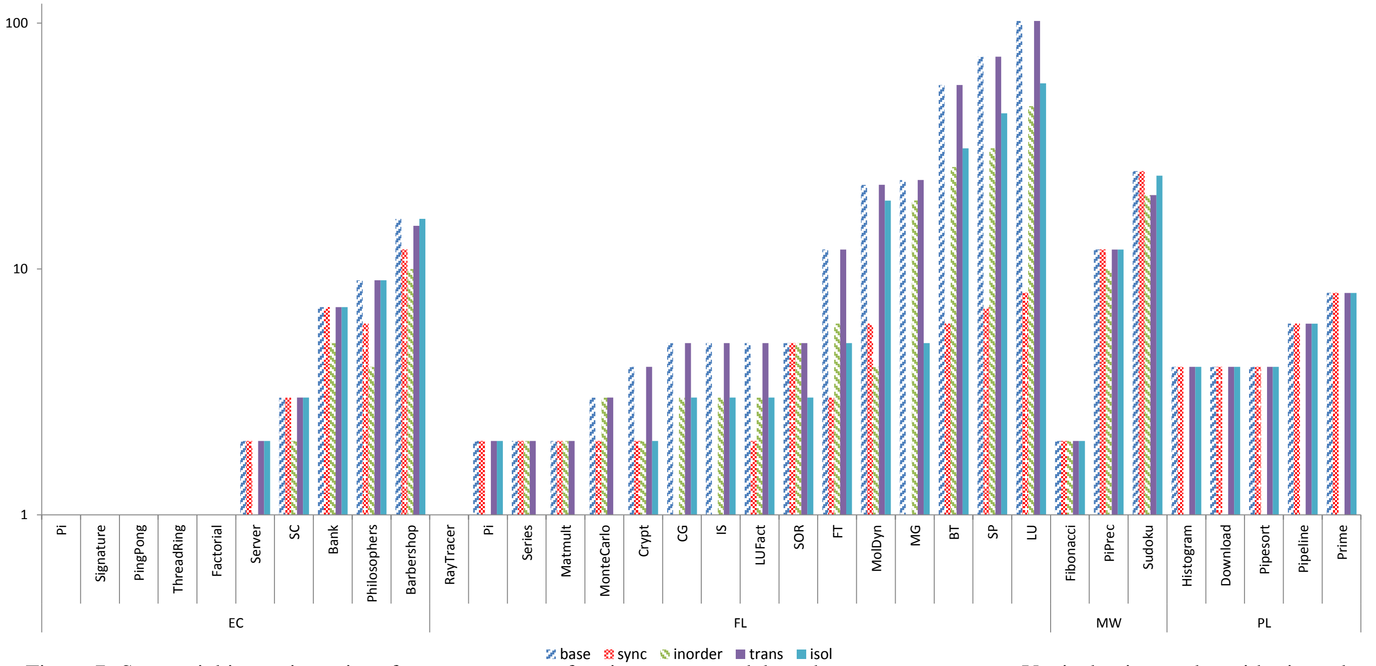


Figure 7: Sequential inconsistencies of actor programs of various actor models and concurrent patterns. Vertical axis uses logarithmic scale.

Actors / 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 8: Quantification of minimum number of actor instances messages to reliably trigger sequential inconsistencies.

The overlap area of two models shows the number of sequential inconsistencies prevented by *either* model.

Figure 9 suggests the following trends.

5.3.1 Event Based Coordination

For this pattern, all inconsistencies prevented by +*sync* can also be prevented using a combination of (+*inorder* and +*trans*). There are also inconsistencies (57%), in Figure 9, that can only be prevented using +*inorder* which makes it critical to this pattern.

- This implies that test cases can focus on the combination of (+*inorder* and +*trans*) rather than +*sync* and still test for all inconsistencies prevented by +*sync*.

5.3.2 Loop Parallelism

For this pattern, there is no overlapping and all inconsistencies prevented by +*sync* can also be prevented by a combination of (+*inorder* and +*isol*). There are inconsistencies that can only be prevented using +*inorder* or +*isol*, making them critical. This implies testing can focus on the combination (+*inorder* and +*isol*), ignore +*sync* and +*trans*, and still test for all sequential inconsistencies

5.3.3 Pipeline

For this pattern, all sequential inconsistencies are prevented using +*inorder*. The implication being that testing can focus only on +*inorder* and ignore the other criteria.

5.3.4 Master Worker

For applications of this pattern, +*inorder*, +*trans* and +*isol* are all critical, which in turn implies that testing should focus on the combination of (+*inorder* and +*trans* and +*isol*) and ignore +*sync*.

- Another implication is that, when focusing test efforts, one may want to consider not only the number of sequential inconsistencies prevented by a model but also if it is a critical model to a pattern. For instance, for Master Worker, +*isol* prevents only 3% of sequential inconsistencies but there are inconsistencies that only +*isol* can prevent and thus it makes sense to test for isolation along with in-order and synchrony.

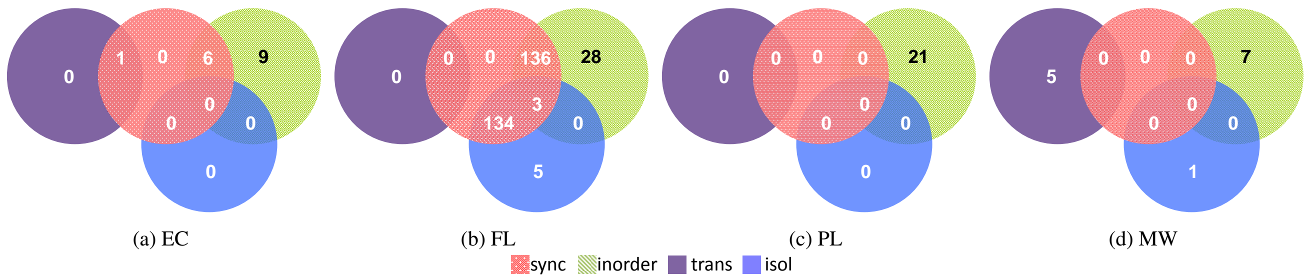
5.3.5 Summary

For Event Based Coordination and Loop Parallelism all sequential inconsistencies prevented by +*sync* are also prevented by combinations of other models and for Pipeline and Master Worker +*sync* does not prevent any inconsistencies. This implies that +*sync* does not need to be tested at all if other criteria are tested. In-order messaging of +*inorder* is critical to all patterns and data isolation of +*isol* is critical to Loop Parallelism and Master Worker. Test efforts should not only consider the number of inconsistencies prevented by a model, but also if the model is a critical one.

5.4 Semantic Spectrums of Actor Models

Figure 10 shows the number of sequential inconsistencies for various combinations of individual actor model, regarding *criteria (1)–(3)*, in two semantic spectrums of actor models: (*base* ⇝*base*+*sync* ⇝*base*+*sync*+*inorder* ⇝ *base*+*sync*+*inorder*+*trans*) with data sharing and data isolation.

Figure 10 suggests the following trends: *(i)* in all patterns of evaluation applications, the number of sequential inconsistencies drops as the happens-before relations add up through the spectrum points.



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

Figure 9: Overlapping of various actor models in preventing sequential inconsistencies (Note: circles not drawn to scale).

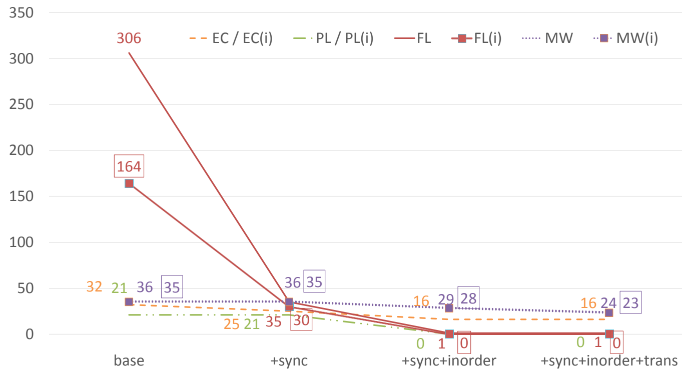


Figure 10: Relation of sequential inconsistency and semantics spectrums. Each spectrum point shows two numbers per pattern, one number with data sharing, and one with data isolation $_{(i)}$.

This is expected because addition of more happens-before relations decreases the chances of two operations running in concurrent, i.e. with no happens-before relation between them. This in turn decreases the chances of unsafe interleaving of operations leading to sequential inconsistencies; *(ii)* another observation is that data isolation prevents more sequential inconsistencies in Loop Parallelism in *base* than other patterns and models. *(iii)* and addition of data isolation to different points in a semantic spectrum prevents up to 15% more sequential inconsistencies for the point [41].

5.5 Threats to Validity

The external validity of our study is limited by our choice of actor programs. These actor programs are chosen from either previous work or well-known concurrent benchmarks, however, we cannot claim that they form an exhaustive set of all typical actor programs. Another threat to external validity is that the same actor program is not implemented in various concurrent patterns and there is an uneven distribution of sequential inconsistencies among actor programs of different patterns.

The internal validity of our study could be threatened by our refactoring process to adapt actor programs of previous work and multi-threaded benchmarks to Panini [18] 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 does not statically detect manual implementations of synchronous messages using asynchrony [14], in-order delivery and processing and transitive in-order delivery, by programmers.

6. RELATED WORK

Sequential consistency Grace [42] proposes a transactional memory technique to manage multi-threaded programs and enforce sequential semantics to avoid concurrency bugs such as race conditions or atomicity violations. Safe Futures [43] provide a semblance of sequential semantics for multi-threaded programs by using object versioning and task revocations. Lamport [44] proposes requirements for shared memory multiprocess programs to guarantee correctness regarding sequential consistency. Qadeer [45] and Cain *et al.* [46] propose model checking techniques for sequential consistency. However, these works are not focused on studying the relation of sequential consistency and semantics of actor models.

Testing and model checking of actor programs Lauterburg *et al.* [27] proposes Basset to explore possible interleaving of transitions of actor programs. Sen and Agha [47] propose jCute to explore behaviors of actor programs using concolic execution for test generation and deadlock detection. Fredlund and Svensson propose McErlang [48] to model check distributed and fault tolerant Erlang programs for safety and liveness properties. Tasharofi *et al.* [49] proposes Setac for testing Scala Actor programs using user specified constraints on non-deterministic schedule of message exchanges; and proposes Bita [50] for testing actor programs using higher coverage scheduling. Bordini *et al.* [51] propose a translation from the actor language AgentSpeak into Java to enable its model checking by Java PathFinder (JPF). Other previous works such as those of Artho and Garoche [52], Barlas and Bultan [53], Stoller [54] and Hughes *et al.* [55] propose model checking technique for distributed and networked systems using existing model checking tools such as JPF, etc. However, previous work is mostly concerned about testing and model checking of actor or distributed programs and is not focused on studying the relation of sequential consistency and semantics of actor models.

7. CONCLUSION AND FUTURE WORK

In this work we quantifiably showed how sequential inconsistencies of actor programs vary based on semantic variations of their underlying actor models. Such quantification helps a programmer, especially a sequentially-trained programmer writing concurrent actor programs, to answers questions such as: where to focus testing, between in-order messaging and synchronous messaging, in a Pipeline actor programs in a model with support for both; and interleavings of how many actor instances and their messages should be controlled to reveal most of sequential inconsistencies with the least complexity. One venue for future work is to study other properties of actor models in addition to sequential consistency.

8. ACKNOWLEDGEMENT

Authors were supported by NSF grants CCF-11-17937, CCF-10-17334 and CCF-08-46059. We would like to thank Sean L. Mooney and Bryan Shrader for their help with some refactoring.

9. REFERENCES

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

[2] Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer'96 **29**(12)

[3] Raynal, M., Vidyasankar, K.: A distributed implementation of sequential consistency with multi-object operations. In: ICDCS'04

[4] Adve, S.V., Boehm, H.J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM'10 **53**(8)

[5] Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: PLDI'09

[6] Dijkstra, E.: Classics in software engineering'79

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

[8] Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI'73

[9] Erlang Programming Language. http://www.erlang.org/

[10] Armstrong, J.: Erlang. Commun. ACM'10 **53**(9)

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

[12] Typesafe. http://www.typesafe.com/company/casestudies

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

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

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

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

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

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

[19] Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y., Neuendorffer, S.: Taming heterogeneity - the Ptolemy approach. IEEE'03 **91**(1)

[20] Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Not. **36**(12) (2001) 20–34

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

[22] JavAct. http://www.javact.org/JavAct.html

[23] Sillito, J.: Stage: Exploring Erlang style concurrency in Ruby. In: IWMSE'08

[24] Parley. http://osl.cs.uiuc.edu/parley/

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

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

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

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

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

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

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

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

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

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

[35] Akka Documentation. http://doc.akka.io/docs/akka/snapshot/general

[36] 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

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

[38] Lin, Y., Dig, D.: CHECK-THEN-ACT misuse of Java concurrent collections. In: ICST'13

[39] 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

[40] Rinard, M.C., Diniz, P.C.: Commutativity analysis: A new analysis framework for parallelizing compilers. In: PLDI'96

[41] Long, Y., Bagherzadeh, M., Lin, E., Mooney, S.L., Upadhyaya, G., Rajan, H.: Quantification of sequential consistency in actor-like systems: An exploratory study. Technical Report 14-03, Iowa State U.

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

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

[44] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. TC'79 **28**(9)

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

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

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

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

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

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

[51] Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. AAMAS'06 **12**(2)

[52] Artho, C., Garoche, P.L.: Accurate centralization for applying model checking on networked applications. In: ASE'06

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

[54] Stoller, S.: Model-checking multi-threaded distributed Java programs. In: SPIN Model Checking and Software Verification'00

[55] Hughes, D., Greenwood, P., Coulson, G.: A framework for testing distributed systems. In: Peer-to-Peer Computing'04

APPENDIX

	Applications	*base*	*base +sync*	*base +sync +inorder*	*base +sync +inorder +trans*
EC	Bank	6 / 6	6 / 6	4 / 4	4 / 4
	Barbershop	15 / 15	11 / 11	8 / 8	8 / 8
	Factorial	0 / 0	0 / 0	0 / 0	0 / 0
	Philosophers	8 / 8	5 / 5	3 / 3	3 / 3
	Pi	0 / 0	0 / 0	0 / 0	0 / 0
	SC	2 / 2	2 / 2	1 / 1	1 / 1
	Signature	0 / 0	0 / 0	0 / 0	0 / 0
	PingPong	0 / 0	0 / 0	0 / 0	0 / 0
	ThreadRing	0 / 0	0 / 0	0 / 0	0 / 0
	Server	1 / 1	1 / 1	0 / 0	0 / 0
	Total	**32 / 32**	**25 / 25** (↓**22 / 22%**)	**16 / 16** (↓**50 / 50%**)	**16 / 16** (↓**50 / 50%**)
	Unresolved	16 / 16 (50 / 50%)			
FL	BT	55 / 30	5 / 5	0 / 0	0 / 0
	CG	4 / 2	0 / 0	0 / 0	0 / 0
	FT	11 / 4	2 / 2	0 / 0	0 / 0
	IS	4 / 2	0 / 0	0 / 0	0 / 0
	LU	101 / 56	7 / 7	0 / 0	0 / 0
	MG	22 / 4	0 / 0	0 / 0	0 / 0
	SP	72 / 42	6 / 6	0 / 0	0 / 0
	LUFact	4 / 2	1 / 1	0 / 0	0 / 0
	MolDyn	21 / 18	5 / 5	0 / 0	0 / 0
	Series	1 / 0	1 / 0	0 / 0	0 / 0
	SOR	4 / 2	4 / 2	0 / 0	0 / 0
	Matmult	1 / 0	1 / 0	0 / 0	0 / 0
	Crypt	3 / 1	1 / 1	0 / 0	0 / 0
	RayTracer	0 / 0	0 / 0	0 / 0	0 / 0
	MonteCarlo	2 / 0	1 / 0	1 / 0	1 / 0
	Pi	1 / 1	1 / 1	0 / 0	0 / 0
	Total	**306 / 164**	**35 / 30** (↓**89 / 82%**)	**1 / 0** (↓**99.7 / 100%**)	**1 / 0** (↓**99.7 / 100%**)
	Unresolved	1 / 0 (0.3 / 0%)			
PL	Histogram	3 / 3	3 / 3	0 / 0	0 / 0
	Pipeline	5 / 5	5 / 5	0 / 0	0 / 0
	Download	3 / 3	3 / 3	0 / 0	0 / 0
	Pipesort	3 / 3	3 / 3	0 / 0	0 / 0
	Prime	7 / 7	7 / 7	0 / 0	0 / 0
	Total	**21 / 21**	**21 / 21** (↓**0 / 0%**)	**0 / 0** (↓**100 / 100%**)	**0 / 0** (↓**100 / 100%**)
	Unresolved	0 / 0 (0 / 0%)			
MW	Fibonacci	1 / 1	1 / 1	1 / 1	1 / 1
	PiPrec	11 / 11	11 / 11	9 / 9	9 / 9
	Sudoku	24 / 23	24 / 23	19 / 18	14 / 13
	Total	**36 / 35**	**36 / 35** (↓**0 / 0%**)	**29 / 28** (↓**19 / 20%**)	**24 / 23** (↓**33 / 34%**)
	Unresolved	24 / 23 (67 / 66%)			
	Total	**395 / 252**	**117 / 111** (↓**70 / 56%**)	**46 / 44** (↓**88 / 83%**)	**41 / 39** (↓**90 / 85%**)
	Unresolved	41 / 39 (10 / 15%)			

Figure 11: Quantification of sequential inconsistencies over two semantics spectrums of actor models. Numbers before the slash show sequential inconsistencies for the spectrum with data sharing among actors, whereas numbers after the slash show inconsistencies for spectrum with data isolation.

Messages \ Actors		*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**
EC	**1**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	2	38%	0%	0%	**38%**	28%	0%	0%	**28%**	0%	0%	0%	**0%**	39%	0%	0%	**39%**	38%	0%	0%	**38%**
	3	0%	6%	0%	**6%**	0%	4%	0%	**4%**	0%	12%	0%	**12%**	0%	3%	0%	**3%**	0%	6%	0%	**6%**
	4+	0%	6%	50%	**56%**	0%	8%	60%	**68%**	0%	0%	88%	**88%**	0%	6%	52%	**58%**	0%	6%	50%	**56%**
	Total	**38%**	**13%**	**50%**		**28%**	**12%**	**60%**		**0%**	**12%**	**88%**		**39%**	**10%**	**52%**		**38%**	**13%**	**50%**	
FL	**1**	10%	0%	0%	**10%**	9%	0%	0%	**9%**	23%	0%	0%	**23%**	10%	0%	0%	**10%**	0%	0%	0%	**0%**
	2	55%	35%	0%	**90%**	86%	6%	0%	**91%**	1%	25%	0%	**26%**	55%	35%	0%	**90%**	100%	0%	0%	**100%**
	3	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	13%	0%	**13%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	4+	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	38%	0%	**38%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	Total	**65%**	**35%**	**0%**		**94%**	**6%**	**0%**		**24%**	**76%**	**0%**		**65%**	**35%**	**0%**		**100%**	**0%**	**0%**	
PL	**1**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	2	33%	0%	0%	**33%**	33%	0%	0%	**33%**	0%	0%	0%	**0%**	33%	0%	0%	**33%**	33%	0%	0%	**33%**
	3	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	4+	0%	19%	48%	**67%**	0%	19%	48%	**67%**	0%	0%	0%	**0%**	0%	19%	48%	**67%**	0%	19%	48%	**67%**
	Total	**33%**	**19%**	**48%**		**33%**	**19%**	**48%**		**0%**	**0%**	**0%**		**33%**	**19%**	**48%**		**33%**	**19%**	**48%**	
MW	**1**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**0%**
	2	17%	0%	0%	**17%**	17%	0%	0%	**17%**	3%	0%	0%	**3%**	19%	0%	0%	**19%**	17%	0%	0%	**17%**
	3	0%	3%	3%	**6%**	0%	3%	3%	**6%**	0%	3%	0%	**3%**	0%	0%	3%	**3%**	0%	3%	0%	**3%**
	4+	22%	53%	3%	**78%**	22%	53%	3%	**78%**	34%	52%	7%	**93%**	26%	52%	0%	**77%**	23%	54%	3%	**80%**
	Total	**39%**	**56%**	**6%**		**39%**	**56%**	**6%**		**38%**	**55%**	**7%**		**45%**	**52%**	**3%**		**40%**	**57%**	**3%**	

Figure 12: Quantification of minimum number of actor instances and messages to reliably trigger sequential inconsistencies for each concurrent pattern [31].

		base				*base +sync*				*base +sync +inorder*				*base +sync +inorder +trans*			
		2	**3**	**4+**	**Total**	**2**	**3**	**4+**	**Total**	**2**	**3**	**4+**	**Total**	**2**	**3**	**4+**	**Total**
EC	**1**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	2	38%/ 38%	0%/ 0%	0%/ 0%	**38%/ 38%**	28%/ 28%	0%/ 0%	0%/ 0%	**28%/ 28%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	3	0%/ 0%	6%/ 6%	0%/ 0%	**6%/ 6%**	0%/ 0%	4%/ 4%	0%/ 0%	**4%/ 4%**	0%/ 0%	6%/ 6%	0%/ 0%	**6%/ 6%**	0%/ 0%	6%/ 6%	0%/ 0%	**6%/ 6%**
	4+	0%/ 0%	6%/ 6%	50%/ 50%	**56%/ 56%**	0%/ 0%	8%/ 8%	60%/ 60%	**68%/ 68%**	0%/ 0%	0%/ 0%	94%/ 94%	**94%/ 94%**	0%/ 0%	0%/ 0%	94%/ 94%	**94%/ 94%**
	Total	**38%/ 38%**	**13%/ 13%**	**50%/ 50%**		**28%/ 28%**	**12%/ 12%**	**60%/ 60%**		**0%/ 0%**	**6%/ 6%**	**94%/ 94%**		**0%/ 0%**	**6%/ 6%**	**94%/ 94%**	
FL	**1**	10%/ 0%	0%/ 0%	0%/ 0%	**10%/ 0%**	9%/ 0%	0%/ 0%	0%/ 0%	**9%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	2	55%/ 100%	35%/ 0%	0%/ 0%	**90%/ 100%**	86%/ 100%	6%/ 0%	0%/ 0%	**91%/ 100%**	0%/ 0%	100%/ 0%	0%/ 0%	**100%/ 0%**	0%/ 0%	100%/ 0%	0%/ 0%	**100%/ 0%**
	3	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	4+	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	Total	**65%/ 100%**	**35%/ 0%**	**0%/ 0%**		**94%/ 100%**	**6%/ 0%**	**0%/ 0%**		**0%/ 0%**	**100%/ 0%**	**0%/ 0%**		**0%/ 0%**	**100%/ 0%**	**0%/ 0%**	
PL	**1**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	2	33%/ 33%	0%/ 0%	0%/ 0%	**33%/ 33%**	28%/ 28%	0%/ 0%	0%/ 0%	**28%/ 28%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	3	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	4+	0%/ 0%	19%/ 19%	48%/ 48%	**67%/ 67%**	0%/ 0%	16%/ 16%	40%/ 40%	**56%/ 56%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	Total	**33%/ 33%**	**19%/ 19%**	**48%/ 48%**		**28%/ 28%**	**16%/ 16%**	**40%/ 40%**		**0%/ 0%**	**0%/ 0%**	**0%/ 0%**		**0%/ 0%**	**0%/ 0%**	**0%/ 0%**	
MW	**1**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	2	17%/ 17%	0%/ 0%	0%/ 0%	**17%/ 17%**	17%/ 17%	0%/ 0%	0%/ 0%	**17%/ 17%**	3%/ 4%	0%/ 0%	0%/ 0%	**3%/ 4%**	4%/ 4%	0%/ 0%	0%/ 0%	**4%/ 4%**
	3	0%/ 0%	3%/ 3%	0%/ 0%	**3%/ 3%**	0%/ 0%	3%/ 3%	0%/ 0%	**3%/ 3%**	0%/ 0%	3%/ 4%	0%/ 0%	**3%/ 4%**	0%/ 0%	0%/ 0%	0%/ 0%	**0%/ 0%**
	4+	22%/ 23%	53%/ 54%	6%/ 3%	**81%/ 80%**	22%/ 23%	53%/ 54%	6%/ 3%	**81%/ 80%**	34%/ 36%	52%/ 54%	7%/ 4%	**93%/ 93%**	42%/ 43%	50%/ 52%	4%/ 0%	**96%/ 96%**
	Total	**39%/ 40%**	**56%/ 57%**	**6%/ 3%**		**39%/ 40%**	**56%/ 57%**	**6%/ 3%**		**38%/ 39%**	**55%/ 57%**	**7%/ 4%**		**46%/ 48%**	**50%/ 52%**	**4%/ 0%**	

Figure 13: Quantification of minimum numbers of actors instances and messages to reliably trigger sequential inconsistencies over two semantics spectrums of actor models. Numbers before the slash show the percentage of sequential inconsistencies for the spectrum with data sharing among actors, whereas numbers after the slash show inconsistencies for spectrum with data isolation.