7-2-2013

Methods for Modeling and Analyzing Concurrent Software

Reng Zeng
Florida International University, rzeng001@fiu.edu

DOI: 10.25148/etd.FI13080908

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

METHODS FOR MODELING AND ANALYZING CONCURRENT SOFTWARE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Reng Zeng

2013

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Reng Zeng, and entitled Methods for Modeling and Analyzing Concurrent Software, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Shu-Ching Chen

Peter J. Clarke

Ronald M. Lee

Xudong He, Major Professor

Date of Defense: July 2, 2013

The dissertation of Reng Zeng is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2013

DEDICATION

To my parents, my wife and my newborn baby girl.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Xudong He, who offered invaluable advice and financial support throughout my Ph.D study at Florida International University. Dr. He taught me not only what a researcher needs to learn for critical thinking and paper writing, but also how to conduct reseach by giving me freedom to identify problems that interest me. I cannot thank Dr. He enough.

I would also like to thank all my committee members for taking their time to serve on the committee and provide feedback for my dissertation work.

This work was partially supported by the NSF of U.S. under award HRD-0833093. I also appreciate the financial support from the University Graduate School of Florida International University in the form of Presidential Fellowship and Dissertation Year Fellowship.

ABSTRACT OF THE DISSERTATION

METHODS FOR MODELING AND ANALYZING CONCURRENT SOFTWARE

by

Reng Zeng

Florida International University, 2013

Miami, Florida

Professor Xudong He, Major Professor

Concurrent software executes multiple threads or processes to achieve high performance. However, concurrency results in a huge number of different system behaviors that are difficult to test and verify. The aim of this dissertation is to develop new methods and tools for modeling and analyzing concurrent software systems at design and code levels. This dissertation consists of several related results. First, a formal model of Mondex, an electronic purse system, is built using Petri nets from user requirements, which is formally verified using model checking. Second, Petri nets models are automatically mined from the event traces generated from scientific workflows. Third, partial order models are automatically extracted from some instrumented concurrent program execution, and potential atomicity violation bugs are automatically verified based on the partial order models using model checking.

Our formal specification and verification of Mondex have contributed to the world wide effort in developing a verified software repository. Our method to mine Petri net models automatically from provenance offers a new approach to build scientific workflows. Our dynamic prediction tool, named McPatom, can predict several known bugs in real world systems including one that evades several other existing tools. McPatom is efficient and scalable as it takes advantage of the nature of atomicity violations and considers only a pair of threads and accesses to a single shared variable at one time. However, predictive tools need to consider the tradeoffs

between precision and coverage. Based on McPatom, this dissertation presents two methods for improving the coverage and precision of atomicity violation predictions: 1) a post-prediction analysis method to increase coverage while ensuring precision; 2) a follow-up replaying method to further increase coverage. Both methods are implemented in a completely automatic tool.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Concurrent software execute multiple threads or processes to achieve high performance. However, concurrency results in a huge number of different system behaviors that are difficult to test and verify. The aim of this dissertation is to develop new methods and tools for modeling and analyzing concurrent software systems at design and code levels. Figure 1.1 gives an overview of the work in this dissertation, from the perspective of design level and code level, as well as forward engineering and reverse engineering. This dissertation firstly focuses on the design level, makes a shift from forward engineering to reverse engineering, then focuses on atomicity violation bugs where reverse engineering is very useful for analysis.

<table>
<tr><td colspan="2"></td><td colspan="2">Design Level</td><td colspan="2">Code Level</td></tr>
<tr><td colspan="2">Projects</td><td>Mondex</td><td>Scientific Workflow</td><td colspan="2">Atomicity Violation</td></tr>
<tr><td rowspan="3">Modeling</td><td>Create model</td><td>Build Petri nets manually</td><td>Mine traces to build Petri nets automatically to aid design</td><td colspan="2">1. Collect traces by instrumentation
2. Extract partial order from traces</td></tr>
<tr><td>Model translation</td><td>Translate Petri nets to Promela</td><td>N/A</td><td colspan="2">Encode partial orders to Promela automatically</td></tr>
<tr><td>Tool</td><td>N/A</td><td>Based on ProM</td><td colspan="2">McPatom</td></tr>
<tr><td rowspan="2">Analyzing</td><td>Properties</td><td>Write two required security properties manually</td><td rowspan="2">N/A</td><td>Generate a complete set of the patterns of unserializable interleavings automatically in LTL</td><td>Extract read-after-write relationship automatically from original trace</td></tr>
<tr><td>Tool</td><td>SPIN Model Checker</td><td>SPIN Model Checker</td><td>PPA</td></tr>
<tr><td colspan="2"></td><td>Forward Engineering</td><td colspan="3">Reverse Engineering</td></tr>
</table>

Figure 1.1: Overview of this dissertation (Contributions in this dissertation are highlighted in green background)

1.1 Motivation

In recent years, both the Computing Research Association in the U.S. and the UK Computing Research Committee proposed a set of grand challenges in computing sciences. These grand challenges involve great technical difficulties and have tremendous significance. One common grand challenge proposed by the above organizations is on developing dependable software systems [1]. One of the research themes of this grand challenge is to develop a verified software repository [2]. The Mondex smart card, an electronic purse, was chosen as the first pilot project in 2006. The objectives were to demonstrate how research groups can collaborate and compete in scientific experiments, and to generate artifacts to populate the verified software repository [3]. This dissertation contributes to the world wide effort in developing a verified software repository by: developing a formal model of Mondex using Petri nets and temporal logic, then applying model checking techniques to analyze the formal model. On the other hand, formal models are often missing or incomplete, therefore this dissertation develops methods to build formal models automatically for scientific workflows. In many disciplines, individual workflows are large, due to the large quantities of data used, so it is often very hard to create and maintain scientific workflows.

Scientific computing has entered a new era of large scaled sharing provided by the cyberinfrastructure. Scientific workflows have recently emerged as a new paradigm for declarative representation of scientific applications as complex compositions of software components and the dataflow among them [4]. Recent efforts from the scientific workflow community aiming at large-scale capturing of provenance present a new opportunity for using provenance to provide recommendation during creating or updating scientific workflows. Provenance, in the scientific workflow community,

refers to the sources of information, including entities and processes, involved in producing or delivering an artifact. Provenance is important for scientists to assess data quality, validate results, and reproduce experiments. Consequently provenance capture becomes an important scientific workflow research area. Many existing scientific workflow management systems, such as Taverna [5], Kepler [6], VisTrails [7] and Pegasus [8], capture provenance information implicitly in an event log that records events related to the start and end of particular steps in the workflow execution and the corresponding data read and write events. Based on provenance of a combination of system-level monitoring and workflow-based systems, this dissertation aims at providing a general method to mine workflows from provenance to aid designing scientific workflows. Besides mining models from traces to aid model building, this dissertation goes a step further to analyze models built on traces. An interesting concurrent software to explore the methods of building models then analyzing models automatically is multi-threaded programs.

Multi-threaded programs are the most difficult ones to develop and verify because of the huge interleaving space. Multi-core hardware is a growing industry trend, for both high performance servers and low power mobile devices. Multi-threaded programs can exploit multi-core processors at their full potential. Therefore, multi-threaded programs are desired to improve performance. And in the real world, most servers and high-end critical software are multi-threaded. Unfortunately, multi-threaded programs are prone to bugs due to the inherent complexity caused by concurrency. It is difficult to detect concurrency bugs due to the huge number of possible interleavings. Many concurrency bugs escape from testing into software releases and cause some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity [9]. Among different types of concurrency bugs, atomicity violation bugs are the most common

one. Atomicity violation bugs are caused by violations to the atomicity of certain code regions without proper synchronization. They widely exist in the real world systems and contributed to about 70% of the examined non-deadlock concurrency bugs [10]. Therefore, techniques for detecting atomicity violation bugs are extremely important. Toward dependable software systems, this dissertation proposes methods to analyze multi-threaded programs at the code level using model checking to find atomicity violation bugs.

1.2 Model Checking

Testing is an essential part of each software development process, but cannot ensure every possible scenario is covered. In concurrent systems, it is even more difficult to test every possible scenario due to non-determinism, making concurrency bugs the most troublesome in all types of software bugs. Nowadays, it is becoming more and more important to address concurrency bugs with the prevalence of multi-core hardware and concurrent programs. As concurrency bugs are non-deterministic, only exposed on specific thread or process scheduling, they are hard to trigger. This frustrates both testing and reproduction for bug diagnosis.

Model checking is an automatic and efficient method for analyzing finite state systems, to verify whether a given model satisfies given properties, by exhaustive exploration of non-determinism. To use model checking, one has to formulate both the model and desired properties of a system into some precise mathematical language, that is a formal specification. For example, Petri nets or PROMELA can be used to model a system while temporal logic can be used to specify the properties desired. The analysis work in this dissertation is based on model checking techniques.

1.3 Contributions

This dissertation addresses the following work, as highlighted in green in Figure 1.1. All work attempts to improve software reliability using model checking techniques, while the initial work is based on building models manually and the following-up work aim at building models automatically, respectively, in the area of scientific workflows and atomicity violation bugs.

Model checking Petri nets at the design level This dissertation presents a unique solution to the grand challenge Mondex, by specifying Mondex with high level Petri nets and temporal logic, and offering a new systematic method to translate high level Petri net to PROMELA. Our formal specification and verification of Mondex have contributed to the world wide effort in developing a verified software repository. This work is based on models built manually.

Automatically building Petri net models from provenance Aiming at building models in Petri nets automatically, this part of the dissertation presents a method based on provenance to mine models for scientific workflows, including data and control dependency. The mining result can either suggest part of other workflows for consideration, or make familiar parts of workflow easily accessible, thus providing recommendation support for scientific workflow composition. This offers a new approach to build workflows in the context of scientific workflows. Given the fact that provenance captured in any scientific workflow based systems or system level monitoring systems contains information about tasks and their temporal order, the proposed algorithm can give both control and data dependency for recommendation during scientific workflows composition. The method provided in this dissertation can be applied to any scientific workflow management systems.

Automatically building models from traces of program execution Our method checking formal models in Petri nets requires translation from Petri nets to Promela code, this part of the dissertation considering building models in Promela code directly in the context of atomicity violation bugs. I present a method to extract a thread model from an instrumented interleaved trace that only records events related to atomicity violations. Such an interleaved trace is much smaller than the program behavior in a complete execution. Furthermore the extracted thread model enables the checking of all alternative traces with the same causal relationships as the interleaved trace. The completeness of instrumented interleaved traces and the extracted thread models is proved.

Model checking atomicity violation at code level This dissertation presents a complete set of the patterns of unserializable interleavings involving two threads (most concurrency bugs involve only two threads [11]) containing any number of accesses to a shared variable (either user defined or every word sized dynamically allocated memory accessed by multiple threads). These patterns generalize and cover the three accesses proposed in [10][12]. These atomicity violation patterns become property specifications to be checked. Based on the extracted model and the property specifications, this dissertation offers a unique prediction tool - McPatom, for detecting atomicity violation bugs through model checking. McPatom instruments interleaved executions, extracts thread models from interleaved traces, automatically converts (1) thread models into Promela programs and (2) atomicity violation patterns into property specifications. By constraining the checking within a pair of threads involving one shared variable at a time, the interleaving space to be checked is vastly reduced. As a result, McPatom is applicable to large software

systems. McPatom can predict atomicity violations that do not manifest during testing or runtime.

Improving the coverage and precision of atomicity violation prediction Predictive methods and tools need to consider the tradeoffs between precision and coverage. An imprecise tool may report a large number of false positives and thus is not very useful since it is extremely time-consuming if not impossible to manually validate all false positives. On the other hand, a tool lacking coverage can miss significant real bugs and thus provides no assurance for software reliability. This dissertation presents two methods for improving the coverage and precision of atomicity violation predictions: 1) a post-prediction analysis method on relaxing the under-approximate models to increase coverage while ensuring precision; and 2) a follow-up replaying method to further increase coverage. The post-prediction analysis method is lightweight and fast, and makes the precise predictions and achieves better coverage than other existing methods using under-approximate models. The replaying method reduces context switches to the minimal level to improve scalability. Both methods are implemented in a completely automatic tool.

1.4 Chapter Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents our work in model checking Mondex, a grand challenge project, at the design level using Petri nets. Chapter 3 presents a method to build models in Petri nets automatically in the context of scientific workflows. Chapter 4 describes our predictive analysis tool for atomicity violation using model checking at code level. Chapter 5 explains methods for improving the coverage and precision of atomicity violation prediction.

CHAPTER 2

ANALYZING PETRI NETS USING MODEL CHECKING

In this chapter we build a formal specification of Mondex using Petri nets, and provide a way of using model checking to verify the formal specification of Mondex, including the abstract model and concrete model.

2.1 Overview

In recent years, both the Computing Research Association in the U.S. and the UK Computing Research Committee proposed a set of grand challenges in computing sciences. One common grand challenge proposed by the above organizations is on developing dependable software systems [1] [2]. The Mondex smart card, an electronic purse, was chosen as the 1st pilot project in 2006. The objectives were to demonstrate how research groups can collaborate and compete in scientific experiments, and to generate artifacts to populate the verified software repository [3].

Mondex is a payment system, an electronic purse system, based on smart card technology, which offers an alternative to paying cash for goods and services, allowing person-to-person payment. In 1999, Mondex was awarded a security rating of ITSEC Level E6 [13] - the highest possible rating achievable in ITSEC (Information Technology Security Evaluation Criteria).

During the development of Mondex, Z was used to specify and to prove the correctness of Mondex design [14]. Since no network access was required for transaction, it demanded critically high security level on each Mondex purse itself. Z Specification was used to prove the following security properties of Mondex:

1. no value may be created in the system: the sum of all the purses' balances does not increase; and

2. all values must be accounted for in the system: the sum of all purses' balances and lost components does not change.

The security properties were proved manually, which was evaluated by a third party group, and a sanitized version of the proof was published in 2000 [13]. The proof has critically helped Mondex be granted ITSEC security level 6 , the highest level.

In [15], we presented a formal specification of Mondex in SAM [16], a formal software architecture model integrating high-level Petri nets and temporal logic. In this section, we present a way using model checking to analyze the formal specification of Mondex in SAM. This formal specification and verification contributes to the world wide effort on developing a verified software repository.

2.2 Specifying Mondex in SAM

A formal specification of Mondex in SAM was developed in [15]. This section gives a brief SAM specification of the abstract model.

2.2.1 SAM

SAM [16], an architectural description model based on Petri nets and temporal logic, is well-suited for modeling distributed systems. A SAM specification is hierarchical consisting of multiple compositions. Each composition may contain multiple element. Each element $C = (B, S)$ has a behavior model B (modeled in a high level Petri net [17]), and a property specification S (defined by a temporal logic formula). An element is correctly designed if the behavior model B satisfies the property specification S, denoted by $B \models S$. The correctness of a SAM architecture description is defined recursively from the correctness of all elements.

A high level Petri net B is a tuple $(P, T, F, Spec, \varphi, R, \mathcal{L}, M_0)$ where (P, T, F) is the net structure, *Spec* is the underlying algebraic specification that defines the static semantics of net elements, and $(\varphi, R, \mathcal{L}, M_0)$ is the net inscription that maps net elements to terms in the algebraic specification. φ associates each place in P with a type in *Spec*. R associates each transition in T with a boolean term in *Spec*. M_0 is the initial marking which associates each place in P with type respecting ground terms in *Spec*. We assume that the reader has some knowledge of Petri nets and temporal logic, and thus omit their formal definitions, which can be found in [16]. In the sequel, we simply use Petri nets to refer to high level Petri nets.

2.2.2 The Abstract Model

In the Z Specification of Mondex [14], `ether` is used to model the communication channel. Messages between purses could be lost, and also could be read by third parties as there may be somebody eavesdropping, so `ether` is designed as lossy and public, all request messages are initialized in `ether`. Each purse interacts with card reader via connector, contact or contactless. Each purse accepts input from card reader, which could be either an initial request in `ether`, or the message sent out by another purse. Each purse produces an output to `ether`.

Accordingly in the SAM model of Mondex, two places, `msg_in` and `msg_out`, are used to model the communication channel, shown in Fig. 2.1, in which `msg_in` contains tokens for input messages, and `msg_out` contains tokens for output messages. All request messages are initialized in `msg_in`, and each purse accepts input messages from `msg_in`. For output messages, each purse sends them to `msg_out`. All messages in `msg_in` comes from `ether`, and all messages in `msg_out` goes to `ether`.

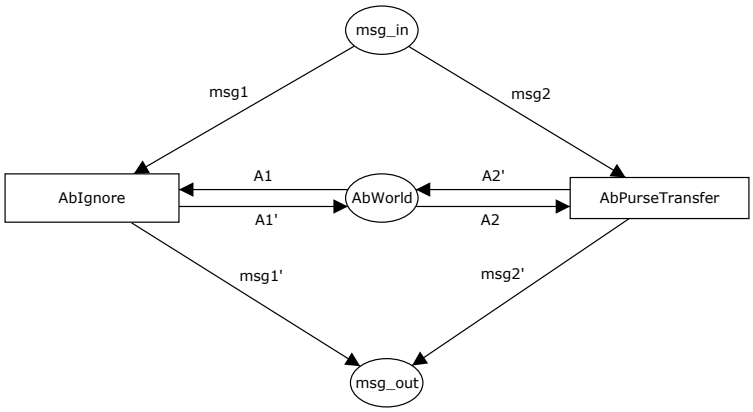


Figure 2.1: The Abstract Model

The abstract model has only one atomic operation to transfer balance from paying purse to receiving purse. It corresponds to transition *AbPurseTransfer* in Fig. 2.1. Transition *AbIgnore* is introduced in Fig. 2.1 to handle invalid messages.

The whole world of abstract purses is modeled using a power set of purses, *AbWorld*.

The net inscription for abstract model is given below, which defines the types of places, constraints of transitions, and the initial marking. The definition of arc labels are omitted since they are self evident in Fig. 2.1.

The Types of Places

The type of *msg_in* contains information of operations and parameters. An operation can be *aNullIn* or *transfer*, and parameters provide transferring details including the name of *from* side (paying party), the name of *to* side (receiving party), and the value to transfer. The type of *msg_in* is thus defined as below.

$$OP = \{aNullIn, transfer\} \tag{2.1}$$

$$\varphi(msg\_in) = OP \times string \times string \times \mathbb{N} \tag{2.2}$$

The type of *AbWorld* is a power set of purses, in which each purse has 3 fields, the first field defines the name of each purse, the second one defines balance and the third one defines lost value.

$$\varphi(AbWorld) = \mathbb{P}(string \times \mathbb{N} \times \mathbb{N}) \tag{2.3}$$

The type of *msg_out* is modeled as *aNullOut*.

$$\varphi(msg\_out) = \{aNullOut\} \tag{2.4}$$

The Constraints of Transitions

The precondition of transition *AbIgnore* tests that the message *msg1* contains operation *aNullIn*, and its postcondition keeps *AbWorld* unchanged.

$$R(AbIgnore) = (msg1[1] = aNullIn) \wedge (A1' = A1) \tag{2.5}$$

For transition *AbPurseTransfer*, its inputs are a message from *msg_in* denoted by *msg2* and all abstract purses from *AbWorld* denoted by *A2*. R(AbPurseTransfer) is the constraint for transition *AbPurseTransfer*, which assures the purse *m* is the *from* side and purse *n* is the *to* side, and *m* is not the same purse as *n*. It also

updates the balance in abstract world.

$$
\begin{aligned}
R(AbPurseTransfer) = (msg2\,[1] = transfer) \wedge \\
\exists\,(m \in A2, n \in A2)\,\centerdot\,(\\
m[1] = msg2[2] \wedge n[1] = msg2[3] \wedge msg2[2] \neq msg2[3] \\
\hline
\wedge\, A2' = A2 \setminus \{m, n\} \cup \\
\{(m[1], (m[2] - msg2[4]), m[3]), \\
(n[1], (n[2] + msg2[4]), n[3]) \\
\} \\
)
\end{aligned}
\tag{2.6}
$$

The Initial Marking

Any permissible initial marking can be provided. To demonstrate the dynamic behavior of our specification, the following initial marking is used.

$$
\begin{aligned}
M_0(msg\_in) &= \{(transfer, 1, 2, 50)\} \\
M_0(msg\_out) &= \{\} \\
M_0(AbWorld) &= \{\{(P1, 100, 0), (P2, 200, 0), (P3, 150, 0)\}\}
\end{aligned}
\tag{2.7}
$$

2.2.3 The Concrete Model of Mondex in SAM

The concrete model deals with the following security issues: (1) a purse could disconnect at any time due to power failure; (2) a message could be lost in the *ether*, the communication channel; and (3) messages in the *ether* are public and could be read by any purses.

The concrete model follows the protocol shown in Fig. 2.2: The wallet starts the transfer with the following messages sequence, message *req*, message *val*, and

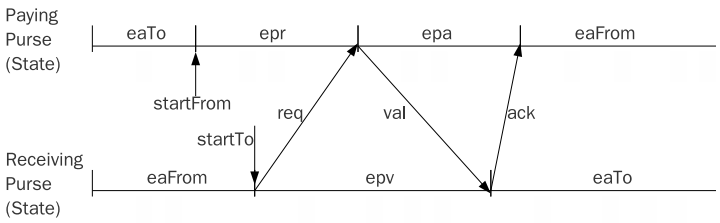


Figure 2.2: The Protocol in Concrete Model

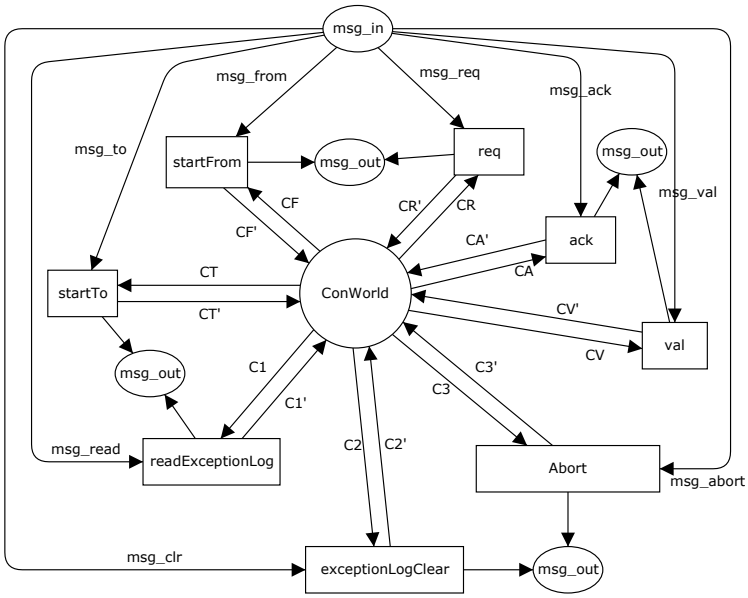


Figure 2.3: The Concrete Model

message *ack*. Message *startFrom* and *startTo* come from card reader, that is triggered by pressing buttons with value to transfer.

Actually state *eaFrom* and *eaTo* can be merged into one state: *idle*, since a purse cannot stay in both *eaFrom* and *eaTo* states.

Fig. 2.3 shows a Petri net model of the concrete purse, in which *msg_in* is the input port in SAM model, and *msg_out* is the output port in SAM model.

There are seven operations that have corresponding transitions in Petri net above, which are listed in Table 2.1.

Table 2.1: Operations List

Operation Name	Operation Description
startFrom	The operation to process the initial message `startFrom` for paying purse.
startTo	The operation to process the initial message `startTo` for payee purse.
req	The operation to process message `req`, requesting payment from paying purse.
val	The operation to process message `val`, transferring balance to payee purse.
ack	The operation to process message `ack`, confirming the paying purse that the transfer is completed.
readExceptionLog	The operation to process message `readExceptionLog`, reading the exception log from purse, and putting the output message into ether.
ExceptionLogClear	The operation to process message `exceptionLogClear`, to clear the exception logs in purse which are already in archive.

Following is the net inscription for the concrete model including types of places, constraints of transitions. The initial markings and definitions of arcs are obvious and thus are omitted.

There is one transition called *abort*, which does not have a corresponding message. *Abort* is triggered in case the message input is *startFrom*, *startTo* or *clearExceptionLog*, and the purse state is *epv* or *epa*.

Operations interact with *ConWorld*, which is a power set of concrete purses.

CounterPartyDetails consists of *name*, *value* and *nextSeqNo*.

$$CPDetails = NAME \times \mathbb{N} \times \mathbb{N}$$

PayDetails contains *TransferDetails*, *fromSeqNo*, *toSeqNo.*

$$FROM = NAME$$

$$TO = NAME$$

$$XferDetails = FROM \times TO \times VALUE$$

$$PayDetails = XferDetails \times \mathbb{N} \times \mathbb{N}$$

The type of *msg_in* includes operation, parameter and name. Operations are listed in Table 2.1 above. A parameter can be *CounterpartyDetails*, or *PayDetails*, for corresponding operations. Name is used to specify which purse to receive the message.

$$OP = \{startFrom, startTo, readExceptionLog, req, val, ack,$$
$$exceptionLogResult, exceptionLogClear, forged\}$$
$$PARAM = CPDetails \times PayDetails$$

Therefore, the type of *msg_in*, is $OP \times PARAM \times NAME$, defined as follows.

$$\begin{aligned}\varphi(msg\_in) =& OP \times NAME \times \mathbb{N} \times \mathbb{N} \\ & \times FROM \times TO \times VALUE \times \mathbb{N} \times \mathbb{N} \times NAME\end{aligned}$$

For *forged* defined in *OP*, all messages emitted by any operation ignoring an input message, or emitted by non-authentic purses, could be *forged*.

The status can be *idle*, *epr*, *epv*, *epa*. *Idle* is the one merged from *eaFrom* and *eaTo* in Z Specification, the initial state, *epr* is the state waiting for message *req*, *epv* is the state waiting for message *value*, and *epa* is the state waiting for message *ack*.

$$STATUS = \{idle, epr, epv, epa\}$$

ConPurse is the concrete purse including fields: name of purse, balance, exception log, next sequence number, pay details and status.

$$\begin{aligned}ConPurse =& NAME \times \mathbb{N} \times \mathbb{P}PayDetails \times \mathbb{N} \times PayDetails \times STATUS \\ =& NAME \times \mathbb{N} \times \mathbb{P}PayDetails \times \mathbb{N} \times FROM \times TO \times VALUE \\ & \times \mathbb{N} \times \mathbb{N} \times STATUS\end{aligned}$$

Message is defined as the same as *msg_in*.

$$Message = msg\_in$$

ConWorld is composed of a power set of concrete purses, *ether*, and *archive*, in which *ether* is a power set of *Message*, for public communication channel, and

archive is *LogBook*, for persistent storage of exception logs.

$$\begin{aligned} LogBook =& \mathbb{P}(NAME \times PayDetails) \\ =& \mathbb{P}(NAME \times FROM \times TO \times VALUE \times \mathbb{N} \times \mathbb{N}) \\ \varphi(ConWorld) =& \mathbb{P}(ConPurse) \times \mathbb{P}Message \times LogBook \\ \varphi(msg\_out) =& msg\_in \end{aligned}$$

As *ConWorld* involves a power set of *ConPurse*, and a *ConPurse* involves a power set of *PayDetails*, thus making *ConWorld* a nested power set. Our tool under development does not support nested power set for the consideration of simplifing its implementation, given the fact that there is always an equivalent non-nested power set. For *ConPurse*, we can transform it as below to remove power set, thus making *ConWorld* a non-nested power set. A *ConPurse* can have a set of *PayDetails* as exception logs, so we use a bool to indicate emptiness of the set of *PayDetails*. If the size of the set of *PayDetails* is greater than 1, we can put another *ConPurse* into *ConWorld*, with different *PayDetails*.

$$\begin{aligned} ConPurse =& NAME \times \mathbb{N} \times bool \times \mathbb{N} \times FROM \times TO \times VALUE \\ & \times \mathbb{N} \times \mathbb{N} \times STATUS \times PayDetails \\ =& NAME \times \mathbb{N} \times bool \times \mathbb{N} \times FROM \times TO \times VALUE \\ & \times \mathbb{N} \times \mathbb{N} \times STATUS \times FROM \times TO \times VALUE \\ & \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

The types of *ConPurse* and *msg_in* are summarized in Table 2.2 and Table 2.3 to facilitate understanding. The mapping relation can also be implemented in a tool for syntax checking against the constraints in the future.

Table 2.2: Summarization of type *ConPurse*

Number	Type	Description
1	NAME	Name of purse
2	$\mathbb{N}$	Balance
3	bool	Emptiness of exception log
4	$\mathbb{N}$	Next Sequence Number
5	FROM	Name of paying side in PayDetails
6	TO	Name of payee side in PayDetails
7	VALUE	Value to transfer in PayDetails
8	$\mathbb{N}$	fromSeqNo in PayDetails
9	$\mathbb{N}$	toSeqNo in PayDetails
10	STATUS	Status
11	FROM	Name of paying side in an exception log
12	TO	Name of payee side in an exception log
13	VALUE	Value to transfer in an exception log
14	$\mathbb{N}$	fromSeqNo in an exception log
15	$\mathbb{N}$	toSeqNo in an exception log

Table 2.3: Summarization of type *msg_in*

Number	Type	Description
1	OP	Operation or message type
2	NAME	Name in CounterPartyDetails
3	$\mathbb{N}$	Value in CounterPartyDetails
4	$\mathbb{N}$	Next Sequence Number in CounterPartyDetails
5	FROM	Name of paying side in PayDetails
6	TO	Name of payee side in PayDetails
7	VALUE	Value to transfer in PayDetails
8	$\mathbb{N}$	fromSeqNo in PayDetails
9	$\mathbb{N}$	toSeqNo in PayDetails
10	NAME	Name of destination purse of this message

The constraint of each transition consists of a precondition and a postcondition. The precondition defines the enabling condition of a transition and the postcondition defines the firing result of the transition. We only provide a detailed explanation of the precondition and the postcondition of transition *startFrom*. For all other transitions, we just give the formula defining its precondition and postcondition.

Transition *startFrom* defines the operation upon receiving *startFrom* message. The precondition tests whether there is purse in concrete world meeting the following conditions:

1. The purse's name matches the name specified in received message, and does not equal the counterparty name in message;
2. The balance of the purse is greater than or equal to the value specified in *startFrom* message; and

3. The purse is in state *idle*.

The postcondition is as follows:

1. Its new *nextSeqNo* is greater than the one before firing transition;
2. Payment details are stored, as paying purse name, payee purse name, value to transfer, paying purse *nextSeqNo*, payee purse *nextSeqNo*;
3. Move to *epr* state;
4. No output message; and
5. The concrete world is updated with new purse and output message.

$$
\begin{aligned}
R(startFrom) = \ & (msg\_from[1] = startFrom) \\
& \wedge \exists (purse \in CF[1]) \centerdot (\\
& (purse[1] = msg\_from[10]) \wedge (purse[1] \neq msg\_from[2]) \\
& \wedge (purse[2] \geq msg\_from[3]) \wedge (purse[10] = idle) \\
& \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2]) \\
& \wedge (purse'[3] = purse[3]) \wedge (purse'[11] = purse[11]) \\
& \wedge (purse'[12] = purse[12]) \wedge (purse'[13] = purse[13]) \\
& \wedge (purse'[14] = purse[14]) \wedge (purse'[15] = purse[15]) \\
& \wedge (purse[4] < purse'[4]) \wedge (purse'[5] = purse[1]) \\
& \wedge (purse'[6] = msg\_from[2]) \wedge (purse'[7] = msg\_from[3]) \\
& \wedge (purse'[8] = purse[4]) \wedge (purse'[9] = msg\_from[4]) \\
& \wedge (purse'[10] = epr) \wedge (msg\_from' = (forged)) \\
& \wedge (CF'[1] = CF[1] \setminus purse \cup purse') \\
& \wedge (CF'[2] = CF[2] \cup msg\_from') \wedge (CF'[3] = CF[3]) \\
&)
\end{aligned}
$$

Transition *startTo* defines the operation upon receiving message *startTo*. The following formula defines the precondition and the postcondition of this transition:

$$
\begin{aligned}
R(startTo) = \ & (msg\_to[1] = startTo) \\
& \wedge \exists (purse \in CT[1]) \,.\, (\\
& \quad (purse[1] = msg\_to[10]) \wedge (purse[1] \neq msg\_to[2]) \\
& \quad \wedge (purse[2] \geq msg\_to[3]) \wedge (purse[10] = idle) \\
& \quad \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2]) \\
& \quad \wedge (purse'[3] = purse[3]) \wedge (purse'[11] = purse[11]) \\
& \quad \wedge (purse'[12] = purse[12]) \wedge (purse'[13] = purse[13]) \\
& \quad \wedge (purse'[14] = purse[14]) \wedge (purse'[15] = purse[15]) \\
& \quad \wedge (purse[4] < purse'[4]) \wedge (purse'[5] = msg\_to[2]) \\
& \quad \wedge (purse'[6] = purse[1]) \wedge (purse'[7] = msg\_to[3]) \\
& \quad \wedge (purse'[8] = purse[4]) \wedge (purse'[9] = msg\_to[4]) \\
& \quad \wedge (purse'[10] = epv) \wedge (msg\_to' = (req, msg\_to[2], msg\_to[3], \\
& \qquad msg\_to[4], purse'[5], purse'[6], purse'[7], purse'[8], \\
& \qquad purse'[9], msg\_to[2])) \\
& \quad \wedge (CT'[1] = CT[1] \setminus purse \cup purse') \\
& \quad \wedge (CT'[2] = CT[2] \cup msg\_to') \wedge (CT'[3] = CT[3]) \\
& \quad) \\
\end{aligned}
$$

Transition *req* is fired upon receiving corresponding message in place *msg_in*. Its inputs are a message from *msg_in* denoted by *msg_req* and all concrete purses from *ConWorld* denoted by *CR*, its outputs are a message denoted by *msg_req'*

to *msg_out*, and all concrete purses denoted by *CR'* to send back to *ConWorld* with necessary change. The precondition and the postcondition of transition *req* is defined by the following formula:

$$
\begin{aligned}
R(req) = \ & (msg\_req[1] = req) \\
& \wedge \exists (purse \in CR[1]) \,.\, (\\
& \quad (purse[1] = msg\_req[10]) \wedge (purse[10] = epr) \\
& \quad \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2] - msg\_req[7]) \\
& \quad \wedge (purse'[3] = purse[3]) \wedge (purse'[4] = purse[4]) \\
& \quad \wedge (purse'[5] = purse[5]) \wedge (purse'[6] = purse[6]) \\
& \quad \wedge (purse'[7] = purse[7]) \wedge (purse'[8] = purse[8]) \\
& \quad \wedge (purse'[9] = purse[9]) \wedge (purse'[10] = epa) \\
& \quad \wedge (purse'[11] = purse[11]) \\
& \quad \wedge (purse'[12] = purse[12]) \wedge (purse'[13] = purse[13]) \\
& \quad \wedge (purse'[14] = purse[14]) \wedge (purse'[15] = purse[15]) \\
& \quad \wedge (msg\_req' = (val, msg\_req[2], msg\_req[3], \\
& \qquad msg\_req[4], purse'[5], purse'[6], purse'[7], purse'[8], \\
& \qquad purse'[9], msg\_req[6])) \\
& \quad \wedge (CR'[1] = CR[1] \setminus purse \cup purse') \\
& \quad \wedge (CR'[2] = CR[2] \cup msg\_req') \wedge (CR'[3] = CR[3]) \\
& \quad) \\
\end{aligned}
$$

Transition *val* defines the operation upon receiving message *val*. The precondition and postcondition of transition *val* are defined by the following formula:

$$
\begin{aligned}
R(val) = \ & (msg\_val[1] = val) \\
& \wedge \exists (purse \in CV[1]) \,.\, (\\
& \quad (purse[1] = msg\_val[10]) \wedge (purse[10] = epv) \\
& \quad \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2] + msg\_val[7]) \\
& \quad \wedge (purse'[3] = purse[3]) \wedge (purse'[4] = purse[4]) \\
& \quad \wedge (purse'[5] = purse[5]) \wedge (purse'[6] = purse[6]) \\
& \quad \wedge (purse'[7] = purse[7]) \wedge (purse'[8] = purse[8]) \\
& \quad \wedge (purse'[9] = purse[9]) \wedge (purse'[10] = idle) \\
& \quad \wedge (purse'[11] = purse[11]) \\
& \quad \wedge (purse'[12] = purse[12]) \wedge (purse'[13] = purse[13]) \\
& \quad \wedge (purse'[14] = purse[14]) \wedge (purse'[15] = purse[15]) \\
& \quad \wedge (msg\_val' = (ack, msg\_val[2], msg\_val[3], \\
& \qquad msg\_val[4], purse'[5], purse'[6], purse'[7], purse'[8], \\
& \qquad purse'[9], msg\_val[5])) \\
& \quad \wedge (CV'[1] = CV[1] \setminus purse \cup purse') \\
& \quad \wedge (CV'[2] = CV[2] \cup msg\_val') \wedge (CV'[3] = CV[3]) \\
& \quad) \\
\end{aligned}
$$

Transition *ack* defines the operation upon receiving message *ack*. The precondition and the postcondition are defined by the following formula:

$$
\begin{aligned}
R(ack) = \ & (msg\_ack[1] = ack) \\
& \wedge \exists (purse \in CA[1]) \,.\, (\\
& (purse[1] = msg\_ack[10]) \wedge (purse[10] = epa) \\
& \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2]) \\
& \wedge (purse'[3] = purse[3]) \wedge (purse'[4] = purse[4]) \\
& \wedge (purse'[5] = purse[5]) \wedge (purse'[6] = purse[6]) \\
& \wedge (purse'[7] = purse[7]) \wedge (purse'[8] = purse[8]) \\
& \wedge (purse'[9] = purse[9]) \wedge (purse'[10] = idle) \\
& \wedge (purse'[11] = purse[11]) \\
& \wedge (purse'[12] = purse[12]) \wedge (purse'[13] = purse[13]) \\
& \wedge (purse'[14] = purse[14]) \wedge (purse'[15] = purse[15]) \\
& \wedge (msg\_ack' = (forged)) \\
& \wedge (CA'[1] = CA[1] \setminus purse \cup purse') \\
& \wedge (CA'[2] = CA[2] \cup msg\_ack') \wedge (CA'[3] = CA[3]) \\
&)
\end{aligned}
$$

Transition *readExceptionLog* defines the operation upon receiving message *readExceptionLog*. The precondition and the postcondition are defined below:

$$
\begin{aligned}
R(readExceptionLog) = \ & (msg\_read[1] = readExceptionLog) \\
& \wedge \exists (purse \in C1[1]) \,.\, (\\
& (purse[1] = msg\_read[10]) \wedge (purse[10] = idle)
\end{aligned}
$$

$$
\begin{aligned}
&\wedge ((purse[3] = true) \wedge (msg\_read' = \\
&\quad (exceptionLogResult, msg\_read[2], msg\_read[3], \\
&msg\_read[4], purse[11], purse[12], purse[13], \\
&purse[14], purse[15], msg\_read[10])) \\
&\quad \vee (purse[3] = false \wedge msg\_read' = (forged)) \\
&) \\
&\wedge (C1'[1] = C1[1]) \wedge (C1'[3] = C1[3]) \\
&\wedge (C1'[2] = C1[2] \cup msg\_read') \\
&)
\end{aligned}
$$

Transition *clearExceptionLog* defines the operation upon receiving message *clearExceptionLog*. The precondition and the postcondition are defined below:

$$
\begin{aligned}
R(clearExceptionLog) = \ & (msg\_clr[1] = clearExceptionLog) \\
& \wedge \exists (purse \in C2[1]) \centerdot (\\
& \quad (purse[1] = msg\_clr[10]) \wedge (purse[10] = idle) \\
& \quad \wedge (purse[3] = true) \wedge (msg\_clr' = (forged)) \\
& \quad \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2]) \\
& \quad \wedge (purse'[3] = false) \wedge (purse'[4] = purse[4]) \\
& \quad \wedge (purse'[5] = purse[5]) \wedge (purse'[6] = purse[6])
\end{aligned}
$$

$$
\begin{aligned}
&\wedge (purse'[7] = purse[7]) \wedge (purse'[8] = purse[8]) \\
&\wedge (purse'[9] = purse[9]) \wedge (purse'[10] = purse[10]) \\
&\wedge (C2'[1] = C2[1] \setminus purse \cup purse') \\
&\wedge (C2'[2] = C2[2] \cup msg\_clr') \wedge (C2'[3] = C2[3]) \\
)
\end{aligned}
$$

Transition `Abort` defines the operation to deal with exception. The precondition and the postcondition are defined by the following formula:

$$
\begin{aligned}
R(Abort) = \ & ((msg\_abort[1] = startFrom) \vee (msg\_abort[1] = startTo) \\
& \vee (msg\_abort[1] = clearExceptionLog)) \\
& \wedge \exists (purse \in C3[1]) \,.\, (\\
& (purse[1] = msg\_abort[10]) \\
& \wedge ((purse[10] = epv) \vee (purse[10] = epa)) \\
& \wedge (purse'[1] = purse[1]) \wedge (purse'[2] = purse[2]) \\
& \wedge (purse'[4] = purse[4]) \\
& \wedge (purse'[5] \geq purse[5]) \wedge (purse'[6] = purse[6]) \\
& \wedge (purse'[7] = purse[7]) \wedge (purse'[8] = purse[8]) \\
& \wedge (purse'[9] = purse[9]) \wedge (purse'[10] = idle) \\
& \wedge (purse'[3] = true) \wedge (purse'[11] = purse[5])
\end{aligned}
$$

$$
\begin{aligned}
&\wedge (purse'[12] = purse[6]) \wedge (purse'[13] = purse[7]) \\
&\wedge (purse'[14] = purse[8]) \wedge (purse'[15] = purse[9]) \\
&\wedge (C3'[1] = C3[1] \cup purse') \\
&\wedge (C3'[2] = C3[2]) \wedge (C3'[3] = C3[3]) \\
)
\end{aligned}
$$

The definitions of arcs are self evident from Fig. 2.3.

2.3 Analyzing the Specification in SAM

Model checking is an automatic and effective method for analyzing finite state systems, which is well suited for this SAM specification. In SAM, model checking is to ensure $B \models S$, that is the behavior model B satisfies the property specification S. The behavior model B uses high level Petri net, which employs sets and power sets as the type of places. The property specification S uses linear temporal logic. SPIN uses PROMELA as its input language to model the behavior, and uses linear temporal logic to specify the properties. In order to use SPIN for model checking SAM specification, the behavior model B is translated to PROMELA code, and the property specification S remains the same. Translation between formal models are often useful, various issues with regard to formal model translation were discussed in [18].

2.3.1 SPIN and PROMELA

SPIN [19] is a well known model checking tool used in the verification of finite state systems. PROMELA, as the input language of SPIN, consists of processes, channels,

and variables. For the channels, there are operations to fetch messages from them randomly or first-in-first-out, and to fetch the messages with desired field value. It is also possible to test the existence of desired messages in channels while not changing anything.

Specifically, single question mark "?" is a PROMELA operator that returns the first message in the channel, double question mark "??" is a PROMELA operator that returns the first matched message in the channel, "[...]" is a PROMELA testing operator returning true or false, while does not block the execution and does not copy messages in the channel, and "<...>" is a PROMELA channel poll operator which copys a message without removing it from the channel if a desired message exists in the channel. There is a predefined unary function in PROMELA called *eval* to turn an expression into a value. "!" is a PROMELA operator that sends a message to the channel.

2.3.2 Rules to Translate High Level Petri Net to PROMELA

This section introduces the rules to translate a high level Petri net to PROMELA, with the abstract model of Mondex (Fig. 2.1) as the example, however, the rules are also applied to the concrete model of Mondex for model checking discussed in Section 2.4. Before discussing the details of rules, we outline the translation by explaining the mapping from a high level Petri net to PROMELA code, as shown in Table 2.4.

Without the loss of generality, we assume all the types in a Petri net model are directly definable in PROMELA in this section, since we can always make a type conversion before the translation.

Table 2.4: Outline of mapping relationships from Petri Nets to PROMELA

Petri Nets	Description
Places	Places contain tokens,while in PROMELA channel contains messages, thus places are translated into channels.
Transitions	Each transition is translated into a PROMELA inline function.
Transition constraints	The contraints for each transition have 2 parts: precondition and postcondition.
Initial markings	The initial marking is translated to initial messages in the channel.

2.3.2.1 Step 1. Define places as channels

Each place is translated into a PROMELA channel; and tokens are translated into messages. Specifically, let $p \in P$ be a place in Petri net with type $\varphi(p) = s_1, s_2, ..., s_n$, we define a bounded channel in PROMELA as follows.

```
#define Bound_p const
chan type_p = [Bound_p] of {s_1, s_2, ..., s_n};
```

where *const* is a user defined positive integer value. Line 5 in Section 2.5 is a translation example of place AbWorld in Fig. 2.1 with type defined in Formula 2.3.

2.3.2.2 Step 2. Define the inline functions for the precondition of a transition

The inline function works like usual preprocessor macro. It is introduced here to offer better translation structure and facilitate automated translation.

Formally, for each transition $t \in T$ with constraint:

$$R(t) = PreCond(t) \wedge PostCond(t) \tag{2.8}$$

where $PreCond(t)$ is the precondition of transition t and $PostCond(t)$ is the postcondition of transition t. $R(t)$ contains basic relational expression connected through logical conjunction $\wedge$ or logical disjunction $\vee$, in which $PreCond(t)$ contains only variables on input arcs and $PostCond(t)$ contains variables on output arcs with or without variables on input arcs. Let $v \in \mathcal{L}(p,t)$ denote a simple variable in case v does not have a power set type. Let $v \in S,\ S \in \mathcal{L}(p,t)$, S has a power set type, v denotes a quantified variable. We assume the first field of either simple variables or quantified variables be the key field, and for those variables v containing only one field, each reference of v is viewed as $v[1]$.

We use the constraint (Formula 2.6) of transition *AbPurseTransfer* as an example in this section, in which the part above the line is the precondition and the part below the line is the postcondition.

We define an inline function to check the enabledness of the precondition of each transition. First, we define a boolean variable *t_is_enabled* to store the truth value of the checking for transition *t*, with initialized value false, refer to Step 5 below. Second, for the fields of each simple variable or quantified variable, we define corresponding variables. Let v be the name of simple variable or quantified variable containing n fields, $TYPE(i)$ be the type of i^{th} field, we define $TYPE(i)\ v\_fieldi$; for $i \in 2..n$. For example, we define Line 29-30 in Appedix 2.5 for Formula 2.6.

Table 2.5 gives the general mapping for basic relational expression connected through logical conjunction $\wedge$ or logical disjunction $\vee$. We use single question mark for simple variables such that messages in the channel are retrieved in FIFO order, and we use double question mark for quantified variables since existential quantification implies a search throughout the whole power set. We use "<...>" to make a guard statement for *if* statement in PROMELA, so that only in case there is a desired message the statements following guard statement are executed and the matched

Table 2.5: General Mapping from basic relational expressions in the precondition of each transition in a Petri Net to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$v[1] = Exp$ where $v \in \mathcal{L}(p,t)$, $p \in P$, $t \in T$ and v is a simple variable containing n fields, Exp does not contain any first field.	$type\_p$? <$eval(Exp)$, $v\_field2, v\_field3, ..., v\_fieldn$>
$\exists(v \in S)$. $(v[1] = Exp)$ where $v \in S$, $S \in \mathcal{L}(p,t)$, $p \in P$, $t \in T$ and v is a quantified variable containing n fields, Exp does not contain the first field of any quantified variable.	$type\_p$?? [$eval(Exp)$, $v\_field2, v\_field3, ..., v\_fieldn$]

Table 2.6: Mapping from the precondition in Formula 2.6 to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$msg2[1] = transfer$	$type\_msg\_in? < eval(transfer), msg2\_field2, msg2\_field3, msg2\_field4 >$
$m[1] = msg2[2]$	$type\_AbWorld??[eval(msg2\_field2), m\_field2, m\_field3]$
$n[1] = msg2[3]$	$type\_AbWorld??[eval(msg2\_field3), n\_field2, n\_field3]$
$msg2[2] \neq msg2[3]$	$msg2\_field2 != msg2\_field3$

message is copied, for example, in Section 2.5, Line 31 is a guard statement for Line 60, where the matched message is copied to msg2_field2 to msg2_field4 for each field; and we use "[...]" to test the existence of messages in case a truth value is needed for *if* statement and the matched message does not require a copy.

Table 2.6 gives the mapping for the precondition in Formula 2.6.

Line 26-37 in Section 2.5 is the resulted PROMELA code.

Table 2.7: General Mapping from basic relational expressions in the postcondition of each transition in a Petri Net to PROMELA Expressions

Basic Relational Expression	PROMELA Expressions
$v[1] = Exp$ where $v \in \mathcal{L}(p,t)$, $p \in P,\ t \in T$, and v is a simple variable containing n fields.	$type\_p$? $eval(Exp)$, $v\_field2, v\_field3, ...,$ v_fieldn
$S' = S \backslash \{v\}$ where $v \in S,\ S \in \mathcal{L}(p,t)$, $S' \in \mathcal{L}(t,p)$ $p \in P,\ t \in T$, and v is a quantified variable containing n fields, $v[1] = Expression$ is a part of the precondition.	$type\_p$?? $eval(Exp)$, $v\_field2, v\_field3, ...,$ v_fieldn
$v' = Exp$ where $v' \in \mathcal{L}(t,p)$, $p \in P,\ t \in T$.	$type\_p$! Exp
$S' = S \cup \{(Exp_1, Exp_2, ..., Exp_n)\}$ where $S \in \mathcal{L}(p,t),\ S' \in \mathcal{L}(t,p)$, $p \in P,\ t \in T$.	$type\_p$! $Exp_1, Exp_2, ..., Exp_n$

2.3.2.3 Step 3. Define the inline function for the postcondition of a transition

For each transition, once its precondition is met, it can fire. This section introduces the rules to define an inline function for the postcondition of a transition firing.

In the rules for the precondition, we test enabledness without moving any tokens, thus as part of the postcondition we move tokens through input arcs. For a simple variable v on an input arc a message from the head of channel obtained from place p is retrieved, according to the constraint $v[1] = Exp$ in the precondition. For a simple variable v' on an output arc, a message is sent to the channel obtained from place p. For a quantified variable $v \in S$, if $S' = S \backslash \{v\}$ is a part of the postcondition, a message is retrieved by searching throughout the channel obtained from place p, according to the constraint $v[1] = Exp$ in the precondition. Besides the cases above, we need to deal with $\cup\{(Exp_1, Exp_2, ..., Exp_n)\}$ in case $S' = S \backslash \{v\} \cup \{(Exp_1, Exp_2, ..., Exp_n)\}$is a part of the postcondition, by sending a message to the

Table 2.8: Mapping from the postcondition in Formula 2.6 to PROMELA Expressions

Basic Relational Expression	PROMELA Code
$msg2[1] = transfer$	$type\_msg\_in?eval(transfer), msg2\_field2, msg2\_field3, msg2\_field4$
$A2' = A2 \backslash \{m\}$	$type\_AbWorld??eval(msg2\_field2), m\_field2, m\_field3;$
$\backslash \{n\}$	$type\_AbWorld??eval(msg2\_field3), n\_field2, n\_field3;$
$\cup\{(m[1], (m[2] - msg2[4]), m[3])\}$	$type\_AbWorld!msg2\_field2, m\_field2 - msg2\_field4, m\_field3;$
$\cup\{(n[1], (n[2] + msg2[4]), n[3])\}$	$type\_AbWorld!msg2\_field3, n\_field2 + msg2\_field4, n\_field3;$

channel obtained from place p, using the values of $(Exp_1, Exp_2, ..., Exp_n)$. Table 2.7 gives the general mapping. After firing the transition, $t\_is\_enabled$ is set to false.

Table 2.8 gives the mapping for the postcondition in Formula 2.6, in which $m[1]$ is replaced with $msg2\_field2$ and $n[1]$ is replaced with $msg2\_field3$ as the precondition since we do not declare variables in PROMELA for the first field of each simple variable or quantified variable.

Line 38-47 in Section 2.5 is the resulted PROMELA code.

2.3.2.4 Step 4. Define an inline function for each transition

Each transition has its precondition and postcondition, we define an inline function for each transition $t \in T$ using the inline functions for its precondition and postcondition. Firing transition is defined as atomic operations using PROMELA keyword *atomic*.

```
inline t()
{
    is_enabled_t(); /*Set t_is_enabled to true/false*/
```

```
    if
    ::  t_is_enabled -> atomic{fire_t()}
    ::  else -> skip
    fi
}
```

For example, Line 48-54 in Section 2.5 is the inline function for transition *AbPurseTransfer* in Fig. 2.1.

2.3.2.5 Step 5. Define a process for the whole net

The dynamic semantics of a Petri net is to non-deterministically fire enabled transitions. We define the following PROMELA process with a loop to capture the dynamic semantics of a Petri net.

```
proctype ModelName(){
 bool t1_is_enabled = false;
 bool t2_is_enabled = false;  ...
 bool tn_is_enabled = false;
 do
  ::t1()
  ::t2()    ...
  ::tn()
 od
}
```

where $T = \{t_1, t_2, ...t_n\}$. For example, we define a process as Line 55-62 in Section 2.5, for abstract model of Mondex in Fig. 2.1.

2.3.2.6 Step 6. Define the initial marking and run the processes

Let $P = \{p_1, ..., p_n\}$, for each place $p \in P$, with initial marking $M_0(p) = \{m_1, m_2, ..., m_k\}$. We define $sort\_p$! m_i for each i, $i \in 1..k$ and run the process *ModelName* defined in the steps above.

```
init {
  type_p1!m1;... type_p1!mk1;
  ...
  type_pn!m1;... type_pn!mkn;
  run ModelName()
}
```

For example, we define Line 63-67 in Section 2.5 for abstract model of Mondex in Fig. 2.1, according to Formula 2.7.

2.3.3 Translation Correctness

Katz et al. [18] proposed a framework for translating models and specifications, in which atomicity of transitions and variables with unspecified next values were discussed as issues in translation. In our work, we use the *atomic* keyword in PROMELA to make the transition atomic, and we use temporal logic to specify the postcondition for each variable.

We introduce the definitions of completeness and consistency before defining translation correctness. Completeness ensures that each place, transition and initial marking has its representation in PROMELA code.

Definition 1. *Translation Completeness:* Each entity in a Petri net is mapped to a language construct in PROMELA.

Lemma 1. *Given a Petri net N, there exists a* PROMELA *program P_N representing N.*

Proof. The rules in Section 2.3.2 cover the translation from N to P_N. □

Consistency ensures that the PROMELA code preserves the semantics of a Petri net. While there are several well known semantic models of Petri nets, we adopt the interleaving semantics, which is adequate for studying the system properties defined in temporal logic.

Definition 2. *Translation Consistency:* The dynamic behaviour of a Petri net is preserved in PROMELA code. The interleaved execution is a sequence $\sigma = M_0 t_o M_1 t_{1...} t_{n-1} M_n$, where $n \geqslant 0$, $M_i (i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ is a marking and $t_i (i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ is a transition firing. PROMELA code execution is $\sigma' = S_0 Run(p_{t_0}) S_1 Run(p_{t_1}) ... Run(p_{t_{n-1}}) S_n$, where $S_i (i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ is a snapshot of values in variables defined in PROMELA code, and $Run(p_{t_i}) (i \in \mathbb{N} \wedge 0 \leqslant i \leqslant n)$ denotes the execution of inline function p_{t_i} translated from t_i as the rules in Section 2.3.2.

Lemma 2. *(Initial Marking Consistency) The initial marking of a Petri net N is consistent with the initial values of variables in translated* PROMELA *P_N.*

Proof. According to Step 1 in Section 2.3.2, marked places are translated into channels, and Step 6 in Section 2.3.2, the initial marking is used to initialize the channel variables. The initial marking of a Petri net N is M_0, and S_0 is the snapshot of initial values of variables in translated PROMELA P_N. According to Step 6 in Section 2.3.2, S_0 is mapped from M_0. □

Lemma 3. *(Semantic Consistency) P_N bisimulates N.*

Proof. Let σ be an execution of N, we proof P_N simulates N by induction on the length of sequence n.

Base case, $n = 0$. It is the initial marking consistency proved above.

Suppose it is true for $n = k$ that the claim holds, that is, $\sigma = M_0 t_o M_1 t_{1...} t_{k-1} M_k$ is consistent with $\sigma' = S_0 Run(p_{t_0}) S_1 Run(p_{t_1})_{...} Run(p_{t_{k-1}}) S_k$.

If $n = k + 1$, as the Step 2 in Section 2.3.2, the precondition of p_{t_k} is the mapping of precondition of t_k; as the Step 3 in Section 2.3.2, the postcondition of p_{t_k} is the mapping of postcondition of t_k, that is, S_{k+1} is the mapping of M_{k+1}; as the Step 4 in Section 2.3.2, $Run(p_{t_k})$ generates S_{k+1}, which denotes marking M_{k+1} obtained from firing t_k. So, $\sigma_{k+1} = M_0 t_o M_1 t_{1...} t_k M_{k+1}$ is consistent with $\sigma'_{k+1} = S_0 Run(p_{t_0}) S_1 Run(p_{t_1})_{...} Run(p_{t_k}) S_{k+1}$.

The reverse direction is proved in the same way, hence, P_N bisimulates N. □

Definition 3. *Translation Correctness:* Translation correctness consists of translation completeness and translation consistency.

Theorem 1. *Given a Petri net N, the* PROMELA *program P_N obtained from the translation rules in Section 2.3.2 preserves the semantics of N.*

Proof. We prove the translation correctness by proving translation completeness and consistency. It is straightforward from Lemma 1 to 3. □

2.3.4 Analysis Result

There are two security properties to verify for Mondex [14], the details of these properties are listed in Table 2.9.

We use the model checker SPIN to verify the properties in exhaustive mode. Here are the LTL properties we used in SPIN to do verification, in which $bal\_sum = \sum_{a \in A, A \in AbWorld} a[2]$ is the sum of balances, $lost\_sum = \sum_{a \in A, A \in AbWorld} a[3]$ is the

Table 2.9: The Properties of Mondex to Verify

Property Name	Property Description
All Value Accounted	all value must be accounted for in the system: the sum of all purses' balances and lost components does not change.
No Value Created	no value may be created in the system: the sum of all the purses' balances does not increase.

sum of lost amounts, and 450 is exactly the sum of *bal_sum* and *lost_sum* in all initial marking.

$$\Box\, bal\_sum + lost\_sum = 450 \tag{2.9}$$

$$\Box\, bal\_sum \leqslant 450 \tag{2.10}$$

The verification result is that all these LTL properties are satisfied with given initial marking.

2.4 Related Works

Several research groups around the world have tackled this 1st pilot project in recent years. In [20], Z/Eves was used to mechanize the original specification of Mondex in Z [14], which took about eight weeks to complete the mechanization of the entire specification, refinement and its proof. In [21], Alloy was used to specify Mondex and Alloy Analyzer was used to check the specification that resulted in the discovery of several bugs. The specification and analysis took about 6 months for a research internship to finish. [22] used the KIV to specify and verify Mondex using a single refinement, which took about one person month. [23] presented an Event-B specification of Mondex using B4free, which consists of 10 levels, an abstract model and 9 levels of refinement. The development took approximately 2 weeks of total

effort spread over several months. In [24], RAISE was used to specify Mondex. The specification consists of three levels: abstract, intermediate, and concrete. Half of the proofs were done automatically.

Other works on Mondex mainly focus on the automation of the proof of Mondex, while [24] not only made effort on proof of Mondex, but also did some model checking with limits such that there are only 2 purses in the world, and money is in the range 0 to 3, to reduce states as much as possible. Our approach using model checking offers great scalability to verify the properties of Mondex.

Regarding the translation from Petri net to PROMELA, this section offers a unique way to translate high level Petri net to PROMELA. [25] provides an approach to translate SAM to PROMELA in which the embedded C code was used as the main approach, while we do not use embedded C code. [26] had the similar idea to ours on translation rules from Petri net to PROMELA, but it only dealt with low level Petri nets, while we propose an approach to translating high level Petri nets to PROMELA codes.

2.5 A Promela program translated from Abstract Model of Mondex

```
1 #define BOUND_msg_in 10
2 #define BOUND_AbWorld 10
3 #define BOUND_msg_out 10
4 chan type_AbWorld=[BOUND_AbWorld] of {short, int, int};
5 mtype = {aNullIn, transfer};
```

```
6  chan type_msg_in = [BOUND_msg_in] of {mtype, short, short
       , int};
7  mtype = {aNullOut};
8  chan type_msg_out = [BOUND_msg_out] of {mtype};
9  int bal_sum = 450,lost_sum = 0,seed = 0,last_seed = 0;
10 inline is_enabled_AbIgnore() {
11   short msg1_field2;short msg1_field3;int msg1_field4;
12   type_msg_in?<aNullIn,msg1_field2, msg1_field3,
       msg1_field4> ->
13     AbIgnore_is_enabled = true
14 }
15 inline fire_AbIgnore(){
16   type_msg_in?aNullIn,msg1_field2, msg1_field3,
       msg1_field4;
17   AbIgnore_is_enabled = false
18 }
19 inline AbIgnore(){
20   is_enabled_AbIgnore();
21   if
22   ::  AbIgnore_is_enabled -> atomic{fire_AbIgnore()}
23   ::  else -> skip
24   fi
25 }
26 inline is_enabled_AbPurseTransfer(){
27   short msg2_field2, msg2_field3;int msg2_field4;
28   int m_field2, m_field3, n_field2, n_field3;
```

```
29    type_msg_in?<transfer,msg2_field2, msg2_field3,
         msg2_field4>;
30    if
31    ::  msg2_field2 != msg2_field3 &&
32      type_AbWorld??[eval(msg2_field2), m_field2, m_field3]
            &&
33      type_AbWorld??[eval(msg2_field3), n_field2, n_field3]
            ->
34        AbPurseTransfer_is_enabled = true
35    ::  else -> skip
36    fi
37  }
38  inline fire_AbPurseTransfer() {
39    type_msg_in?transfer,msg2_field2, msg2_field3,
         msg2_field4;
40    type_AbWorld??eval(msg2_field2), m_field2, m_field3;
41    type_AbWorld??eval(msg2_field3), n_field2, n_field3;
42    atomic{type_AbWorld!msg2_field2, m_field2 - msg2_field4
         , m_field3;
43    bal_sum = bal_sum - msg2_field4;}
44    atomic{type_AbWorld!msg2_field3, n_field2 + msg2_field4
         , n_field3;
45    bal_sum = bal_sum + msg2_field4;}
46    AbPurseTransfer_is_enabled = false
47  }
48  inline AbPurseTransfer() {
```

```
49    is_enabled_AbPurseTransfer();
50    if
51    ::  AbPurseTransfer_is_enabled -> atomic{
          fire_AbPurseTransfer()}
52    ::  else -> skip
53    fi
54  }
55  proctype AbstractMondex (){
56    bool AbIgnore_is_enabled = false;
57    bool AbPurseTransfer_is_enabled = false;
58    do
59    :: AbIgnore ()
60    :: AbPurseTransfer ()
61    od
62  }
63  init {
64    type_msg_in!transfer,1,2,50;type_AbWorld!1,100,0;
65    type_AbWorld!2,200,0; type_AbWorld!3,150,0;
66    run AbstractMondex()
67  }
```

2.6 Summary

We provide a way of using model checking to verify the formal specification of Mondex in SAM [15], including the abstract model and concrete model. this section is presented with the abstract model as an example.

Effort

It took us two person months to complete the specification[15], and 80 person-hours to translate the SAM model into PROMELA code for Mondex concrete model and to verify the model automatically using SPIN.

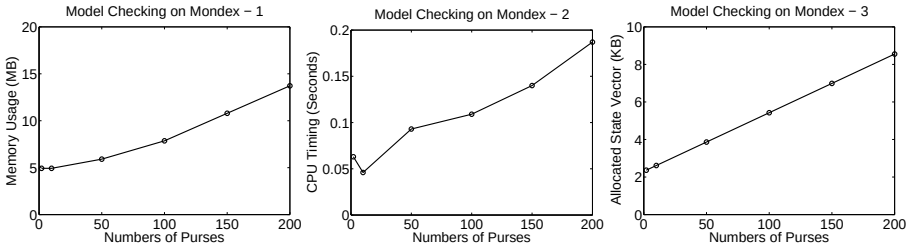
Bugs found

[21] found three bugs in the Z specification, in which one bug is for missing constraints about authenticity, also found by KIV method [22], two bugs are related with reasoning errors during refinement. For the authenticity bug, Z specification gives no constraints for authenticity so that a purse could be making a transaction with a non-authentic purse. For example, a purse is in *epv* status, which is *to* purse, waiting for *val* message, there should be constraints preventing this purse from receiving *req* message as *from* purse. Similarly there should also be constraints preventing the purse in *epa* status as *from* purse from receiving *val* message as *to* purse. Without these constraints for authenticity, the actual role of purse could be inconsistent in the transaction. The other two bugs are both for reasoning errors during refinement which is not present in this section as we using model checking do not do that refinement. Our specification avoids the authenticity bug through adding proper constraints and does not have refinement bugs.

Scalability

We conducted the model checking of Mondex concrete model with a Windows based PC which has 1.8Ghz CPU and 2GB memory. Since the Mondex system is not a network system and only contains atomic operations involving two purses; it is adequate to model and analyze the system with one randomly chosen initial message.

Figure 2.4: Scalability of Model Checking on Mondex



Therefore, we created a random message in the initial markings, the range for value of money was $0 \ldots 2^{31} - 1$. We conducted an experiment by increasing the number of purses in the initial markings, to show the scalability of memory usage, cpu timing and allocated state vector, as the Fig. 2.4 below.

CHAPTER 3

A METHOD TO MINE TRACES FOR BUILDING PETRI NETS TO AID DESIGNING SCIENTIFIC WORKFLOWS

In this chapter, we develop methods to mine traces to build Petri nets automatically to aid designing scientific workflows.

3.1 Using Existing Process Mining Algorithms

This section presents existing process mining algorithms using scientific workflows as examples. Section 3.1.1 presents a method using process mining based on provenance to create and analyze scientific workflows. Figure 3.1 shows a high level view of the context to mine provenance. Applying process mining in the context of scientific workflow needs to address the following issues. In this section we focus on control flow mining, and discuss the other two issues in Section 3.1.4.

1. Control flow mining: To mine control flows from provenance, we need to extract information and to present it in the format acceptable to existing process mining tools. We also need to select appropriate process discovery algorithms depending on the context of scientific workflows.
2. Data dependency: Data dependency contained in provenance can contribute to process mining for improving the mining results. It is critical to enhance the existing control flow based process mining algorithms with data flow capabilities.
3. Incremental mining: Given a scientific workflow template [27], scientists need to fine-tune it many times, which makes updating large scientific workflows a challenge for scientists. Mining from scratch is neither efficient for large scale scientific workflows nor effective to address existing scientific workflow

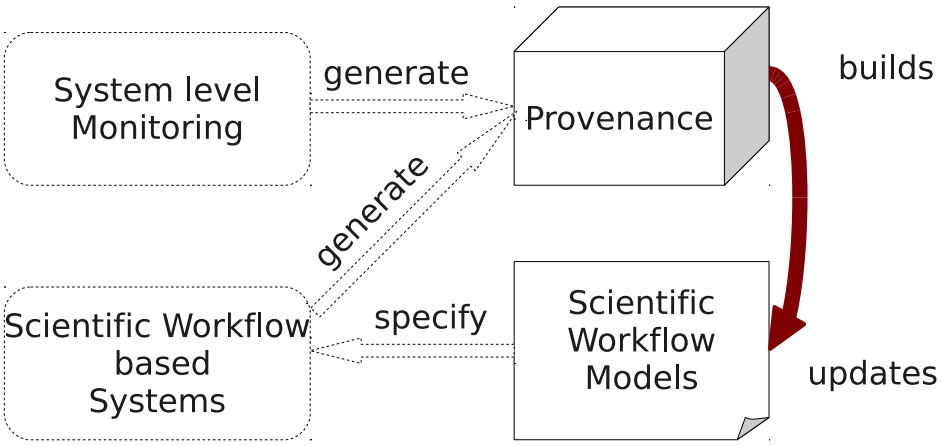


Figure 3.1: Mining provenance

templates. Incremental mining can utilize the information in existing scientific workflow templates to make mining more efficient and effective.

Section 3.1.2 presents a method to convert provenance to XES format that is accepted by existing process mining tools, and provides a method using process mining to create and analyze scientific workflows. Section 3.1.3 contains a brief discussion of related works. Section 3.1.4 discusses our research direction for using process mining to address specific issues in the context of scientific workflows.

3.1.1 Overview

Provenance, in scientific workflow community, refers to the sources of information, including entities and processes, involved in producing or delivering an artifact. More specifically, provenance is captured at four levels [28]. First, the process level captures information about the invoked processes, their inputs/outputs and start/end times. Second, the data level, inferred from the process level, provides derivation paths of intermediate and final products. Third, the organization level

stores the metadata for the experiments. Fourth, the knowledge level connects the scientific experiments' discovery with other provenance levels as supporting evidence. The stored information is used to infer the provenance of intermediate and final results and to verify the quality of the data through tracing the processing steps.

Recent efforts from the scientific workflow community aiming at large-scale capturing of provenance present a new opportunity for building scientific workflow using provenance. Several researchers [29] have investigated how to synthesize a process model from event logs. The research area of process mining focuses on extracting information about processes by examining event logs. Practical experience has shown that typical information recorded in event logs includes information about which activities are performed, at what time, by whom and in the context of which case (i.e., process instance). By explicitly using the case context, process discovery algorithms are capable of constructing process models that accurately describe the process [29]. Since both event logs and provenance contain process information, a given scientific workflow may be executed multiple times [30] thus creating multiple workflow execution instances. Scientific experiments are exploratory in nature thus change are the norm. As a result, mining processes from scientific workflows is highly valuable. Provenance does not record control flows associated no data flows, we are interested in building scientific workflows by combining data flows from provenance and control flows mined from provenance. Our work provides a new direction in using captured provenance.

3.1.1.1 Process Mining and XES format of ProM tool

The goal of process mining, or more specifically control flow discovery is to extract information about processes from event logs, such that the control flow of a process is captured in a process model. In process mining an activity refers to an atomic

part of a process, which may be executed over any length of time and by anyone. We refer to a case (also a process instance) as the execution trace of a process. The starting point for control flow discovery is an event log that contains events such that:

1. Each event refers to an activity (i.e., a well-defined step in the process),
2. Each event refers to a case (i.e., a process instance) and
3. Events are totally ordered (for example by a timestamp).

The (Pro)cess (M)ining framework ProM [31] has been developed as a generic open-source framework where various process mining algorithms have been implemented. Currently, over 280 plug-ins have been added. The framework provides researchers an extensive base to implement new algorithms in the form of plug-ins. The framework provides easy to use user interface functionality, a variety of model type implementations (e.g. Petri nets) and common functionality like reading and writing files. In most cases the starting input is an event log. ProM can read event logs stored in the formats MXML [32] and from Version 6 also in the new event log format XES [33]. For more information on process mining and the ProM framework, we refer to the website www.processmining.org.

XES is an open standard for storing and managing event log data. Its objective is to provide a generic framework onto which all event log meta-models found in practice can be mapped with relative ease, without assuming a specific field of application, or any purpose of the event logs whatsoever. The XES meta-model recognizes and treats all extensions as equal, independent from their source or level of proliferation. This allows users to extend it at will to fit any purpose or domain setting, and thus makes XES a flexible format for all applications. Due to the flexible handling of extensions, and the attributes defined by those, the XES meta-

model allows using applications to interpret also previously unknown information. To provide universally understood semantics, a number of extensions have been standardized, and thus equipped with a fixed semantics. The currently standardized extensions include concept extension, lifecycle extension, organizational extension, time extension, semantic extension and classification extension.

3.1.2 A Method to Build Scientific Workflows from Provenance

Figure 3.1 shows a high level view of the context to mine provenance, to build and update scientific workflows. This section uses provenance generated from scientific workflow management systems, thus results of the method in this section can be compared with existing scientific workflows. Note that the method can be applied to provenance from both sources in Figure 3.1. Figure 3.2 shows a high level view of the method presented and evaluated in this section.

3.1.2.1 Converting Provenance to XES format

XESame [33] is a tool to extract event logs from a data source. The conversion consists of two steps: conversion definition and execution. Conversion definition specifies a mapping, to map concepts of the data source onto concepts of the event log. Conversion execution produces event logs as specified in the mapping. In this section, we use XESame to convert provenance to event logs as the input for process mining tool ProM.

In conversion definition, the most important extension is the concept extension that includes instances and names. Providing names for each event is desired as it is very informative. Names of events are the names of the executed activity

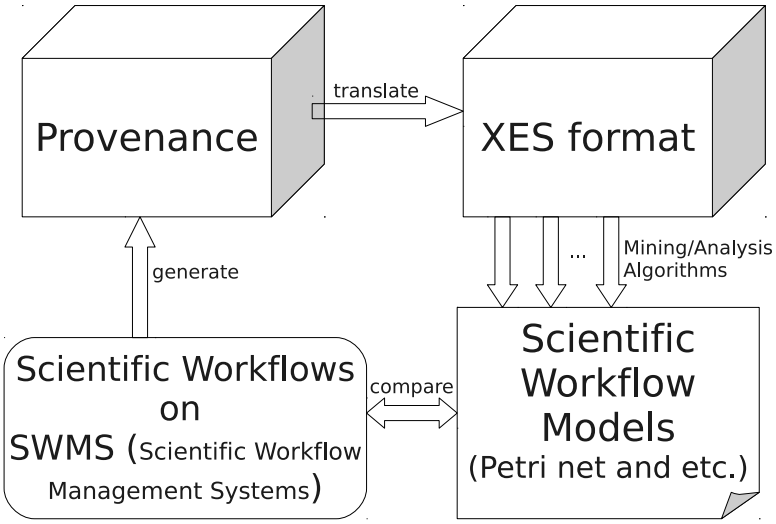


Figure 3.2: Overview of the method

represented by the event. Instances represent identifiers of the activity instances whose executions have generated the events. Another important extension is the time extension. Time extension specifies a timestamp attribute for events, which enables events to be ordered to infer control dependency, and enables performance analysis. For example, using Taverna provenance system, shown in Figure 3.3, we join two tables PROCESSORENACTMENT and PROCESSOR on their PROCESSORID, PROCESSORENACTMENT provides event identifiers and corresponding start time while PROCESSOR provides a event name for each event identifier. Since we need steps in scientific workflow and corresponding start time, we set instance as PROCESSORID, name as PROCESSORNAME, and timestamp as ENACTMENTSTARTED.

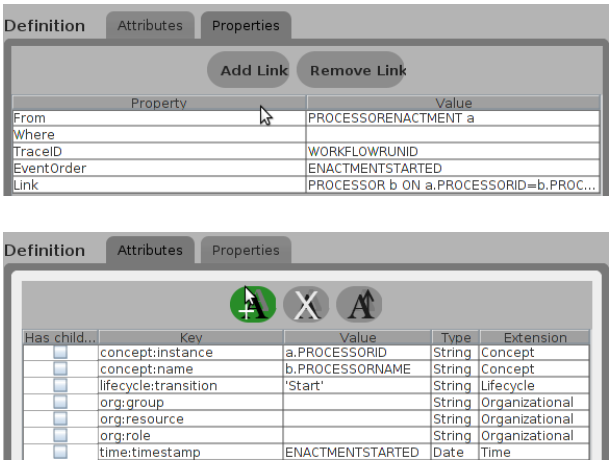


Figure 3.3: Configuration of XESame

3.1.2.2 Building Scientific Workflows through Process Discovery

The Running Example We adopt the challenge workflow from the third Provenance Challenge as an example (http://www.myexperiment.org/workflows/750), which contains both control flow and data flow. While there are several teams implemented the challenge workflow, we choose Taverna as it is connected well to the open scientific workflow repository myExperiment. Provenance provided by Taverna records the tasks executed and its timestamp, together with data links between tasks. As the ongoing research work [7] and [34], the provenance in the near future will be applicable to non-workflow systems that enable provenance to record tasks users perform in their familiar environment, so that the methods investigated in this section are able to build scientific workflows automatically.

Using the Fuzzy Miner The fuzzy miner [35] assumes that problems in mining large scale processes are caused by mismatch between fundamental assumptions of traditional process mining, and the characteristics of real-life processes. Fuzzy miner developed an adaptive simplification and visualization technique for process models, which is based on two metrics, significance and correlation. The two metrics are

similar to the concept of data clustering domain where a binary distance metric is inferred to find related subsets of attributes. In the context of scientific workflows, significance, which can be determined both for tasks and precedence relations over them, measures the relative importance of behavior. As such, it specifies the level of interest we have in tasks and their control dependency. Correlation is only relevant for precedence relations over tasks, which measures how closely related two events following one another is.

As scientific workflows are usually quickly evolving, change can be made to the example workflow several times, including the activities and data. Using the fuzzy miner, a workflow can be mined to provide an abstract view of what does not change, which offers insight of evolving workflows. For the running example, we run it for 10 times, then remove ReadCSVReadyFile and run it for 10 times again, after that we undo removing ReadCSVReadyFile, remove IsMatchCSVFileTables and run it for 10 times. Using XESame provenance can be transformed to a XES file, based on which the fuzzy miner can be applied. Figure 3.4 shows a resulting model in which there is every task but IsMatchCSVFileTables, when significance cutoff is increased to 0.392, as Figure 3.5, ReadCSVReadyFile disappeared so that the unchanged part is shown, which can be the key part of the whole workflow. What's more, by double clicking "Cluster 14" that contains 2 elements, the tasks with low significance are shown, which in our context is the changing tasks. As Figure 3.6 shows, there is a process model related with low significance tasks, which exactly matches the original workflow model in the running example. Therefore, in case there is provenance from either workflow based systems or non-workflow systems that include tasks scientists perform, a scientific workflow can be built automatically at different abstract level by using the fuzzy miner.

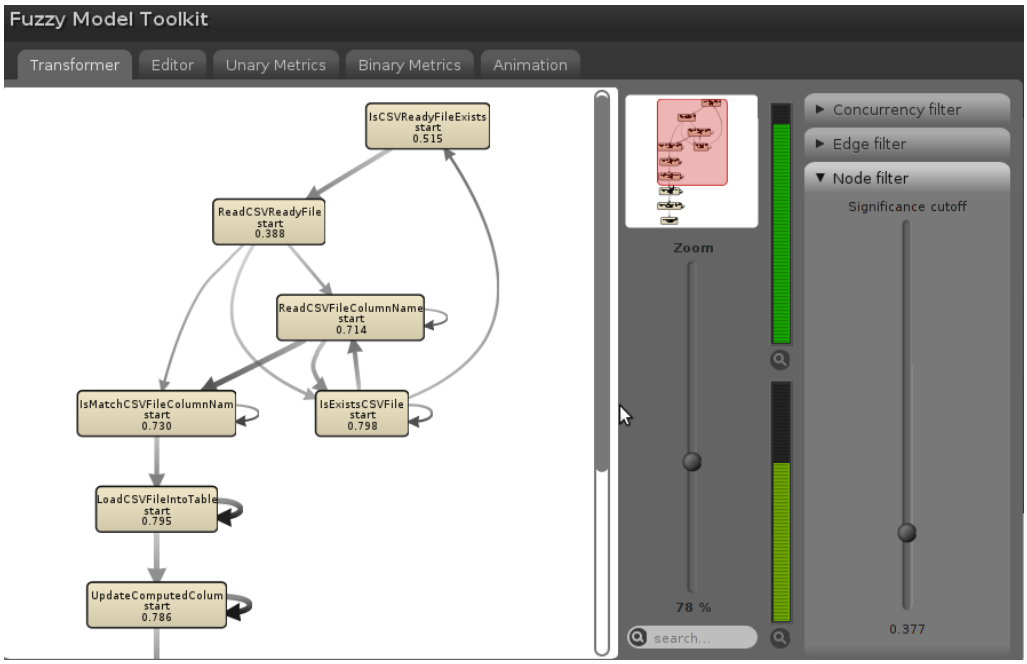


Figure 3.4: Fuzzy Mining Result - 1

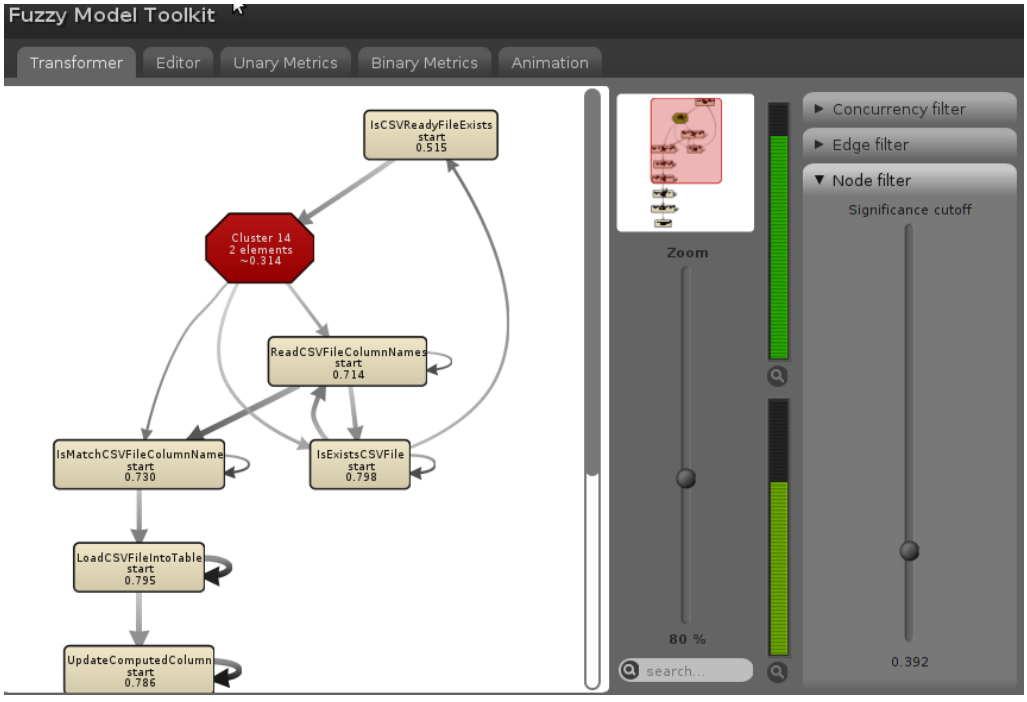


Figure 3.5: Fuzzy Mining Result - 2

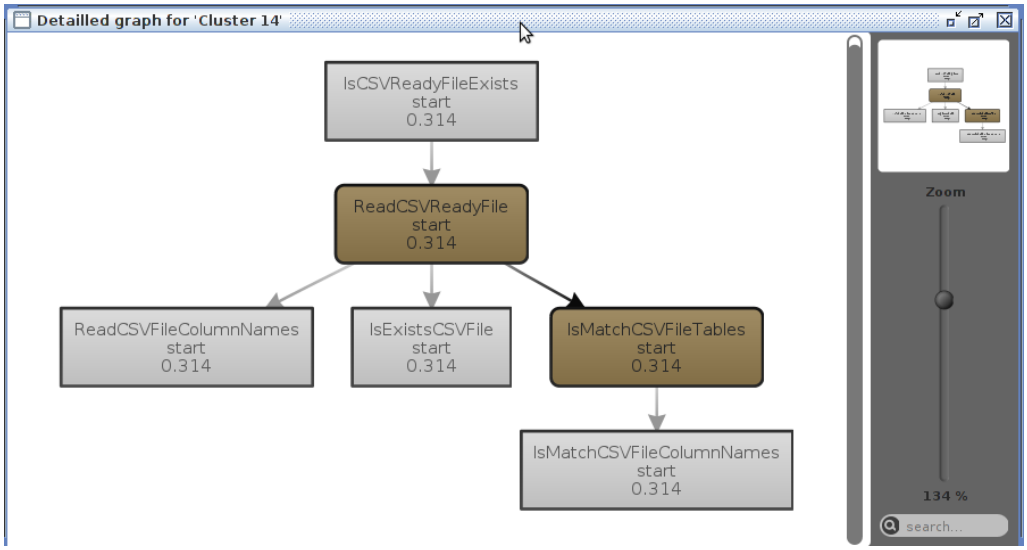


Figure 3.6: Fuzzy Mining Result - 3

Using the Alpha Miner The alpha miner assumes the completeness of direct succession (DS) such that "if two transitions can follow each other directly, then this has occurred at least once in the log", yet it may not be the case in reality, the alpha miner allow users to edit log relations manually to offer more information about direct succession, as shown in Figure 3.7. For large amount of events, manually adding log relations can be impossible. In scientific workflows context, provenance contains data dependencies that imply direct succession in time order, data dependencies can somehow be considered in the alpha miner thus making it closer to completeness of direct succession. We discuss further about data dependencies in Section 3.1.4.

Using the Genetic Miner The genetic miner is a control-flow process mining algorithm that can discover all the common control-flow structures (i.e. sequences, choices, parallelism, loops and non-free-choices, invisible tasks and duplicate tasks) while being robust to noisy logs. The genetic miner has more difficulties to mine models with constructs that allow for many interleaving situations. Figure 3.8 shows the result of the genetic miner on the running exam-

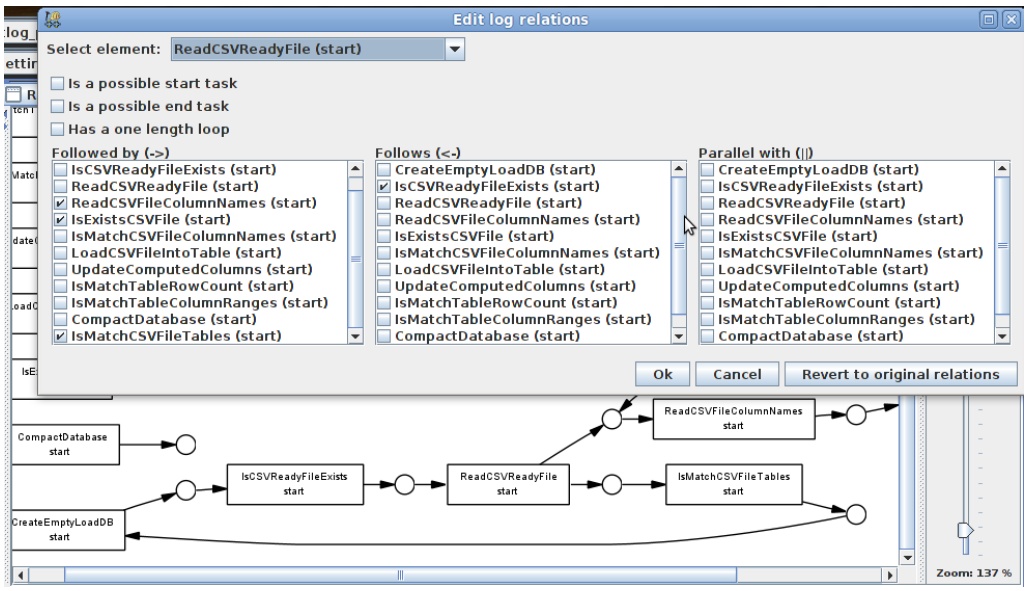


Figure 3.7: Alpha Mining Result

ple. Genetic miner successfully get a non-free-choices construct such as both IsMatchTableRowCount and IsMatchTableColumnRanges depend on UpdateComputedColumns while IsMatchTableRowCount depends on others as well that means mixture of choice and synchronization. It also successfully suggests the dependency between IsMatchTableRowCount and IsMatchTableColumnRanges that is a control link in the running example. The results also give a clear view of frequency by annotating numbers on each event and arc, where numbers in event boxes mean how many times the events happen in the event logs, and numbers on arcs mean how many times the two events directly succeed each other.

Using the Heuristic Miner The heuristics Miner is a practical applicable mining algorithm that can deal with noise, and can be used to express the main behavior (i.e. not all details and exceptions) registered in an event log [36]. It includes three steps: (1) the construction of the dependency graph, (2) for each activity, the construction of the input and output expressions and (3) the search for long distance dependency

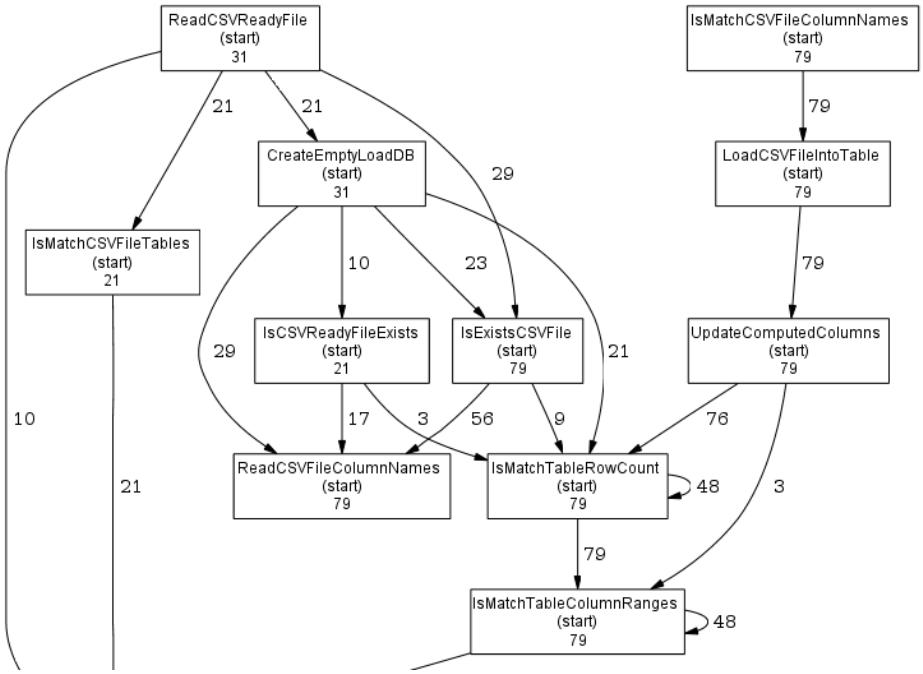


Figure 3.8: Genetic Mining Result

relations. Figure 3.9 shows the result of heuristics miner on the running example. Although IsMatchCSVFileTables does not directly succeed ReadCSVReadyFile in event logs, heuristics miner successfully suggests their dependency with reliability 0.833 and it happens 5 times in event logs considering long distance dependency relations. This is particularly useful in the context of scientific workflows, just as the running example, many scientific workflows have multiple tasks even hundreds of tasks scheduled in parallel, not each parallel task succeed the dependent task directly in provenance, therefore, long distance dependency discovery is especially important in the context of scientific workflows.

3.1.2.3 Analyzing Scientific Workflows

Using LTL Checking The size of provenance is growing large quickly, Linear Temporal Logical (LTL) checking is a great tool to help scientists discovering and

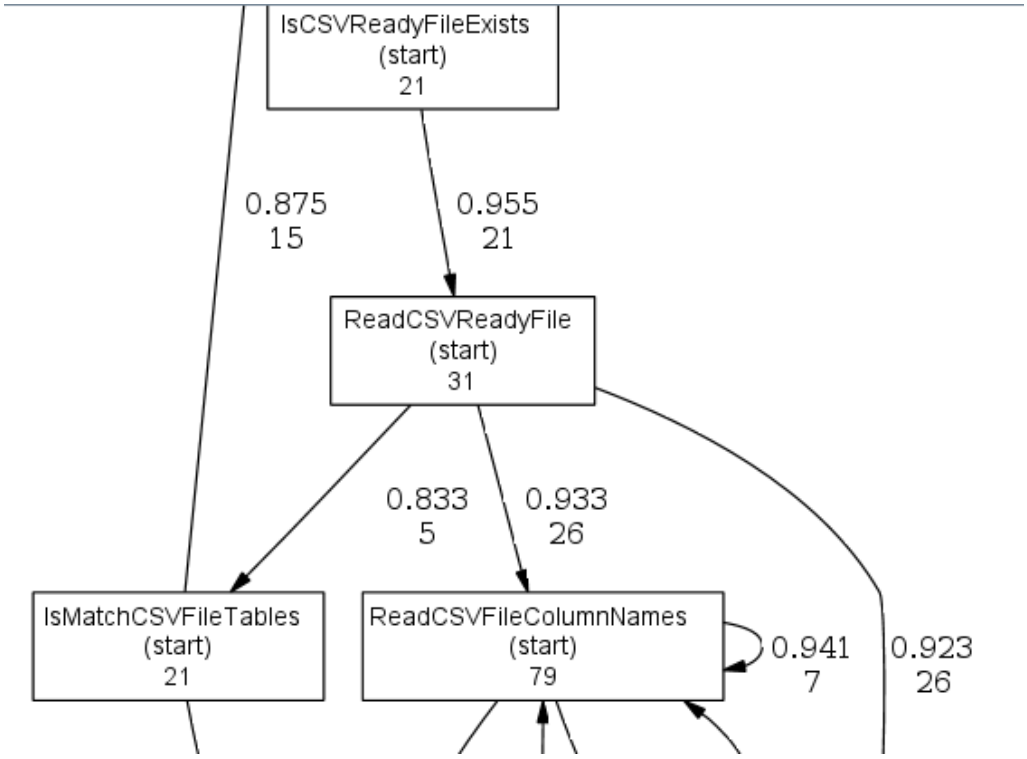


Figure 3.9: Heuristic Mining Result

double checking temporal properties of provenance. As shown in Figure 3.10, we can easily check whether ReadCSVFileColumnNames eventually happens when IsExistsCSVFile happens, it is true for 27 instances while false for 4 instances, for further information, the specific workflow run can be referred to according to workflow run identifier.

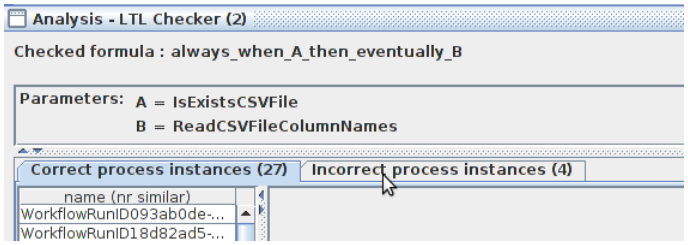


Figure 3.10: LTL Checking Example

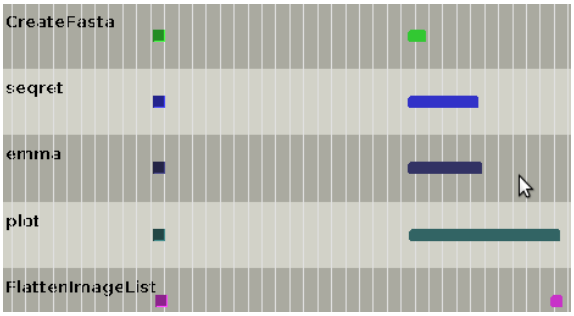


Figure 3.11: Dotted Chart Analysis

Using Dotted Chart A dotted chart offers insight of performance during scientific workflow execution, thus enables improving the performance of a workflow by exploiting an episodic memory of prior workflow executions. Figure 3.11 shows part of the result using dotted chart analysis on an example[1], in which each row is a task in workflow and each dot is an occurrence of the corresponding task along the time scale, so we can easily see the performance of scientific workflow execution in the perspective of tasks and take corresponding actions such as distributing tasks further.

3.1.3 Related Works

The cloud computing and other technologies are changing the way we create, share and use information, which offers great benefits but also exposes us to serious new problems. [34] believes that provenance will play an essential role in this revolution, providing data integrity, trustworthiness, authenticity, and availability, while offering potential benefits to information retrieval, collaboration, and scientific computation. [37] aims at mining provenance, by applying Case Based Reasoning (CBR) methods to provenance to support scientists' workflow generation process, which does not generate the whole workflow but focusing on assisting workflow composi-

[1] http://www.myexperiment.org/workflows/158.html

tion by providing recommendation to scientists regarding each interested task. [38] addresses the queries from the provenance challenge workshop such as semantic reasoning which exposes the implicit links between provenance, e.g. the implicit links between provenance of studying any part of a human's body including chest, legs, arms and etc. An abstraction over the provenance information is presented by two means: one is the users' specified annotations that draw an interpretative link between tasks, and the other is the typed views that hide or expose the execution details of an iteration or a nested run, or the data lineage of a collection and its elements. Other works such as [39], [40] and [41] also address the queries from the provenance challenge workshop, however do not deal with mining processes from provenance.

3.1.4 Discussion

3.1.4.1 Results of different process discovery algorithms

Section 3.1.2.2 presents results of four different process discovery algorithms on the running example. Table 3.1 discusses the results in the context of scientific workflows. Note that the result of each miner is correct based on given provenance, but providing different views of the provenance. It is found that the result of the fuzzy miner is closest to the original scientific workflow in the running example. Section 3.1.4.3 discusses a possible way to improve the results in Table 3.1.

3.1.4.2 Number of Traces in Provenance

As Figure 3.2 shows, this section uses provenance from scientific workflow management systems. A question that current tools can not address is how many times should the scientific workflow be run to get enough traces. There should be a fixed

Table 3.1: Discussion on results of process discovery algorithms

	Description	Result
Fuzzy Miner	Provides a zoom-able view of scientific workflows by controlling significance cutoff to show tasks at different importance level.	Under certain significance cutoff, the fuzzy miner successfully gives the changed part and unchanged part. Comparing with original scientific workflow, the fuzzy miner gets most dependency correctly, but concludes some dependency that does not exist.
Alpha Miner	Provides a view of direct succession between tasks in provenance.	Assuming the completeness of direct succession, the alpha miner fails to give a view close to the original scientific workflow.
Genetic Miner	Provides a view of frequency for both tasks and succession between tasks, and discovers all common control-flow structures assuming the existence of noise.	The genetic miner gets a good view of structures and frequencies, yet gives some wrong dependencies which does not exist in both the original scientific workflow and the results of the fuzzy miner.
Heuristic Miner	Provides a view of scientific workflows by considering long distance dependency.	The heuristic miner gives long distance dependency successfully, but gives too much dependency for some tasks such as ReadCSVFileColumnNames.

point after that no more precedence relations to be discovered even given additional provenance. This section manually find a point after that the mined results do not change significantly with additional provenance.

3.1.4.3 Build Scientific Workflows using Data Dependency

Scientific workflows include data dependency and control dependency, provenance provides data dependency besides temporal sequences. The method provided in this section only uses the temporal sequences of tasks in provenance to mine dependency among tasks. Data dependency can contribute to process mining for improving the mining result, but process mining and its existing tools do not accept explicit data dependency as source. Since provenance provides data dependency, we can derive causality relation from data dependency, which compliments the causality relation extracted from the precedence of tasks.

3.1.4.4 Incremental Scientific Workflow Mining

Scientific problem solving is an evolving process. Scientists start with a set of questions then observe phenomenon, gather data, develop hypotheses, perform tests, negate or modify hypotheses, reiterate the process with various data, and finally come up with a new set of questions, theories, or laws. Often before this process can end in results, scientists will fine-tune the experiments, going through many iterations with different parameters [28]. Updating scientific workflows is hence a challenge for scientists. We believe with pre-existing scientific workflow template, created either manually or automatically through mining, we can apply process mining to update it based on new provenance obtained from either workflow based systems or non-workflow systems. We are working on incremental scientific workflow mining. Incremental mining can utilize the information in existing scientific

workflow templates to make mining more efficient for large scale scientific workflows and more effective for addressing existing scientific workflow templates.

3.1.5 Conclusion

This section provides a method using process mining to build and analyze scientific workflows, which offers a new approach to build large scale workflows in the context of scientific workflows. Recent efforts from scientific workflow community on capturing provenance present a new opportunity for using provenance. This section presents a method using process mining based on provenance to build and analyze scientific workflows, which provides a new direction in using captured provenance. Given the fact that provenance captured in any scientific workflow based systems or system level monitoring systems contains information about tasks and their temporal order, there is always a way to translate the provenance to XES format acceptable to process mining tools, the method provided in this section can be applied to any scientific workflow management systems.

3.2 A Method to Mine Petri Nets by Improving Process Mining Algorithms with Data Dependency

This section presents a method of improving process mining algorithms with data dependency. The method is applied to find a scientific workflow model from provenance and to provide recommendation support during scientific workflows composition based on the mined workflows, as shown in Figure 3.12.

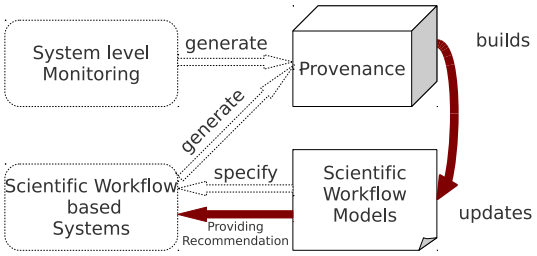


Figure 3.12: Background of the method described in this section (denoted by solid arrows)

3.2.1 What are Scientific Workflows and Provenance?

There are many works on scientific workflows and provenance, that use different terminology for scientific workflows and different ways to organize provenance [42][43][28][5]. The common basics of scientific workflow and provenance this section relies on are given as follows.

A *task* is a procedure or a group of procedures to execute computational activities. A *data product* can be a single data object or a collection of data objects. A *scientific workflow* is a directed graph where nodes are tasks and edges between nodes represent either data dependency or control dependency. *Provenance* records the task invocations and data products used or generated by each invocation. Formally, $Provenance \subseteq \mathbb{P}(Data \times Task \times Data)$. *Data dependency* is the relationship between two tasks t_1 and t_2 when t_2 need t_1's output as input, denoted as $t_1 \prec_d t_2$. Formally, $t_1 \prec_d t_2$ iff $\exists d_1, d_2, d_3 \in Data \centerdot (d_1, t_1, d_2) \in Provenance \wedge (d_2, t_2, d_3) \in Provenance$. Data dependency can be derived from provenance as causality relation pairs, such as $t_1 \prec_d t_2$. *Control dependency* is the relationship between two tasks t_1 and t_2 when a task t_1 is required to be invoked before invoking another task t_2, it is denoted as a causality relation pair $t_1 \prec_c t_2$. A *causality relation pair* infers data dependency or control dependency, denoted as $t_1 \prec t_2$. A *task trace*, corre-

sponds to a run of a scientific workflow, is a sequence of task invocations. Formally, let Σ be the set of all tasks that appear in task traces, a task trace is a sequence of task invocations, denoted as $t_1, t_2, t_3, ..., t_m$ where $t_i \in \Sigma$ for $1 \leq i \leq m$.

3.2.2 Scientific Workflow Models in Petri Nets

The algorithm in this section mines a Petri net as a model to represent a scientific workflow. Tasks are modeled by transitions and causal relations are modeled by places and arcs. A place corresponds to a condition which can be used as pre-condition and/or post-condition for tasks. An AND-split corresponds to a transition with two or more output places, and an AND-join corresponds to a transition with two or more input places. OR-splits/OR-joins correspond to places with multiple outgoing/ingoing arcs.

This section uses WF-nets [44] that is based on Place/Transition nets, a variant of the classic Petri net model.

Definition 4. Place/Transition nets

A Place/Transition net, or simply a P/T-net, is a tuple (P, T, F, M) where

1. P is a finite set of places,
2. T is a finite set of transitions such that $P \cap T = \emptyset$, and
3. $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs.
4. $M : P \to \mathbb{N}$ is a function that associate each place with a natural number.

A place p is an input place of a transition t, also called pre-condition, if there is a directed arc from the place p to the transition t, i.e. $(p, t) \in F$. Similarly a place p is an output place of a transition t, also called post-condition, if $(t, p) \in F$.

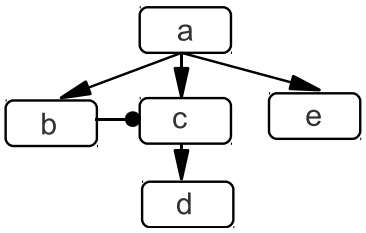


Figure 3.13: A sample workflow (the circle arrow denotes control dependency, and the other arrows denote data dependency)

Definition 5. Workflow nets

Let $N = (P, T, F, M)$ be a P/T-net and t' be a transition such that $t' \notin P \cup T$, N is a workflow net (WF-net) iff:

1. Object creation: P contains an input place i such that $\bullet i = \emptyset$,
2. Object completion: P contains an output place o such that $o\bullet = \emptyset$,
3. Connectedness: $N' = (P, T \cup \{t'\}, F \cup \{(o, t'), (t', i)\})$ is strongly connected.

3.2.3 A Simple Example

To illustrate the principle of the algorithm in this section, we consider the task trace extracted from provenance shown in Table 3.2. Suppose the workflow that generated the provenance is given in Figure 3.13. Consider the fact that scientific workflows evolve quickly thus the change is recorded in provenance, and provenance capturing systems support non-workflow environment, the workflows behind the provenance are often unknown before mining. Since the control flow does not generate any data product, we cannot get the control flow from provenance. This section aims at mining the control dependency from provenance to provide recommendation support during workflow composition.

Table 3.2: A task trace in provenance

Workflow Running Identifier	Task Identifier
1	a
2	a
1	b
2	e
1	c
2	b
2	c
1	d
2	d
1	e

Table 3.3: Direct precedence table

	a	b	c	d	e
a	0	1	0	0	1
b	0	0	2	0	0
c	0	0	0	2	0
d	0	0	0	0	1
e	0	1	0	0	0

3.2.4 Construction of a Causality Table

Definition 6. Direct precedence table

For n tasks, the direct precedence table is a $n \times n$ matrix P, $P = [p_{ij}]$ where $1 \leq i, j \leq n$ and p_{ij} is the number of times that task t_i directly precede task t_j.

Using the example above, a direct precedence table is shown in Table 3.3.

Definition 7. Indirect precedence table

For n tasks, the indirect precedence table is a $n \times n$ matrix S, $S = [s_{ij}]$ where $1 \leq i, j \leq n$ and s_{ij} is calculated as follows. For task t_i and t_j, in each workflow run, if there is a sequence $t_i, t_k, ..., t_m, t_j$, suppose the number of tasks from t_k to t_m is $m - k + 1$, add δ^{m-k+1} ($\delta = 0.8$) to s_{ij}. 0.8 is chosen after experimentation

Table 3.4: Indirect precedence table

	a	b	c	d	e
a	0	0.8	1.44	1.152	0.512
b	0	0	0	1.6	0.64
c	0	0	0	0	0.8
d	0	0	0	0	0
e	0	0	0.8	0.64	0

Table 3.5: Weight table

	a	b	c	d	e
a	0	3.8	3.44	1.152	3.512
b	0	0	2	1.6	0.64
c	0	0	0	4	0.8
d	0	0	0	0	1
e	0	1	0.8	0.64	0

which satisfies two requirements: 1) for direct precedence, $\delta^{m-k+1} = 1$; 2) The longer distance, the smaller addition.

Using the example above, an indirect precedence table is shown in Table 3.4.

According to the definition, data dependency can be derived from provenance as causality relation pairs. For the example above, it is $a \prec_d e, a \prec_d c, a \prec_d b, c \prec_d d$. Following is the construction of weight table combining both precedence tables and data dependency.

Definition 8. Weight table

For n tasks, the weight table is a $n \times n$ matrix W, $W = [w_{ij}]$ where $1 \leq i, j \leq n$ and w_{ij} is calculated as follows. First, $w_{ij} = p_{ij} + s_{ij}$; second, if $i \prec_d j$ is present in causality relation pairs derived from provenance as data dependencies, add σ to w_{ij} where σ is the number of workflow running.

Using the example above, $\sigma = 2$, the weight table is shown in Table 3.5.

Table 3.6: Confidence table

	a	b	c	d	e
a	0	3.8	3.44	1.152	3.512
b	0	0	2	1.6	-0.36
c	0	0	0	4	0
d	0	0	0	0	0.36
e	0	0.36	0	-0.36	0

Table 3.7: Causality table

Causality Relation Pair	Weight
$a \prec b$	3.8
$b \prec c$	2
$c \prec d$	4
$d \prec e$	0.36
$a \prec e$	3.512
$e \prec b$	0.36
$a \prec c$	3.44
$a \prec d$	1.152
$b \prec d$	1.6
$e \prec d$	-0.36

Definition 9. Confidence table

For n tasks, the confidence table is a $n \times n$ matrix C, $C = [c_{ij}]$ where $1 \leq i, j \leq n$ and c_{ij} is calculated as follows: $c_{ij} = w_{ij} - w_{ji}$.

Using the example above, the confidence table is shown in Table 3.6.

The causality table is shown in Table 3.7. For each pair $t_1 \prec t_2$, if it is not data dependency $t_1 \prec_d t_2$, then it is control dependency $t_1 \prec_c t_2$.

Rules are designed as below to update causality table:

1. For tasks t_k, t_m and its causality $t_k \prec t_m$, if its weight is lower than 1, remove it.

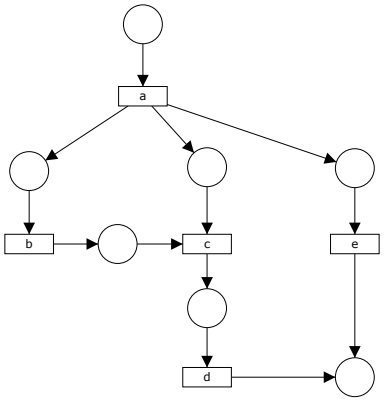


Figure 3.14: A resulting Petri net (all causality pairs are included, and an AND-split is used for task a)

2. For tasks t_k, t_m and its causality $t_k \prec t_m$, if there is $t_k \prec ... \prec t_m$, in which each pair has higher confidence than the one of $t_k \prec t_m$, remove $t_k \prec t_m$ from causality table.

Firstly, for each valid causality, there are many chances to get higher than 1, such as direct precedence, data dependency, or indirect precedence (e.g. $0.8^2 + 0.8^4 = 1.0496 > 1$). Secondly, For $t_k \prec ... \prec t_m$, it is highly possible that $t_k \prec t_m$ get higher than 1 for multiple indirect precedences, but actually there is no direct causality between t_k and t_m.

Using the steps and rules above for the example, valid causality pairs are derived: $a \prec b, b \prec c, c \prec d, a \prec e, a \prec c$, a Petri net can be constructed as Figure 3.14, which matches exactly the original workflow.

The construction of a causality table is summarized in Algorithm 3.1.

3.2.5 Generating a Petri Net from a Causality Table

It is straightforward to derive a causality graph from a causality table, but it requires additional information to generate a Petri net from a causality table, including

Algorithm 3.1 Construction of a Causality Table

Input: Provenance for σ running of a workflow that contains n tasks
Output: A causality table T

1: **for all** workflow running instances Run **do**
2: **for all** (t_i, t_j) t_i directly precedes t_j **do**
3: p_{ij}++
4: **end for**
5: **for all** (t_i, t_j) t_i indirectly precedes t_j **do**
6: Assume the sequence as $t_i, t_k, ..., t_m, t_j$
7: $\delta = 0.8$
8: $s_{ij} += \delta^{m-k+1}$
9: **end for**
10: **end for**
11: **for** $i = 1$ to n **do**
12: **for** $j = 1$ to n **do**
13: $w_{ij} = p_{ij} + s_{ij}$
14: **if** $\exists d_1, d_2, d_3 \in Data \centerdot (d_1, t_i, d_2) \in Run \wedge (d_2, t_j, d_3) \in Run$ **then**
15: $w_{ij} += \sigma$
16: **end if**
17: **end for**
18: **end for**
19: **for** $i = 1$ to n **do**
20: **for** $j = 1$ to n **do**
21: $c_{ij} = w_{ij} - w_{ji}$
22: **if** $c_{ij} \geq 1$ **then**
23: $T = T \cup \{(t_i \prec t_j, c_{ij})\}$
24: **end if**
25: **end for**
26: **end for**
27: **for all** $(t_i \prec t_j, c_{ij}) \in T$ **do**
28: **if** $\exists t_k \prec ... \prec t_m \centerdot (t_i \prec t_k) \wedge (t_m \prec t_j) \wedge (c_{ik} > c_{ij}) \wedge (...) \wedge (c_{mj} > c_{ij})$ **then**
29: $T = T \backslash \{(t_i \prec t_j, c_{ij})\}$
30: **end if**
31: **end for**

parallelism and choice of tasks. In Petri nets, parallelism can be represented with a AND-split, and choice can be represented with a OR-split. For instance, if there are causality pairs $t_i \prec t_j$ and $t_i \prec t_k$, the type of a split from t_i to t_j AND/OR t_k has to be detected, to generate a Petri net. The principle of detection is to employ the weight table above to check the pattern of t_j and t_k: 1) $w_{jk} = 0$ and $w_{kj} = 0$, that shows the pattern $t_j t_k$ or $t_k t_j$ cannot appear, it is an OR-split; 2) Otherwise, that shows the pattern $t_j t_k$ or $t_k t_j$ can appear, it is an AND-split. The algorithm to detect the type of a split is given in Algorithm 3.2. It is assumed that a OR-split is placed after an AND-split, i.e. it is conjunctions of clauses, and each clause is a disjunction of tasks.

Algorithm 3.2 Detection of the type of the splits

Input: Weight table w, task t_0 and tasks $t_1, ...t_i, ..., t_n$ in which $t_0 \prec t_i$ $(1 \leq i \leq n)$ is a causality pair

Output: A set of clauses $Disj$ in which each is a set of tasks that are in the OR-relation
A set of tasks $Conj$ in which each is in the AND-relation

1: Derive W' from W with the rows and columns related with $t_1, ...t_i, ..., t_n$
2: Let $W' = [w'_{ij}]$; $Disj = \emptyset$; $Conj = \emptyset$
3: Create n empty sets: Set_i where $1 \leq i \leq n$
4: **for** $i = 1$ to n **do**
5: **for** $j = 1$ to n **do**
6: **if** w'_{ij}=0 & w'_{ji}=0 & $i \neq j$ **then**
7: $Set_i = Set_i \cup \{i, j\}$
8: $Set_j = Set_j \cup \{i, j\}$
9: **end if**
10: **end for**
11: **end for**
12: **for** $i = 1$ to n **do**
13: **if** Set_i is empty **then**
14: $Conj = Conj \cup \{t_i\}$
15: **else**
16: $Disj = Disj \cup \{< Set_i >\}$
17: **end if**
18: **end for**

3.2.6 Providing Recommendation for Scientific Workflow Composition

Given a partial workflow, based on a set of causality tables and a set of Petri nets, this section provides a method to recommend a next most likely task and related part of a Petri net.

A *causality path* is a sequence of tasks $t_1, ..., t_i, ...t_n$ in which $t_i \prec t_{i+1}$ $(1 \leq i < n)$ are causality pairs. The length of causality path is n.

For the current task t_0 selected in the partial workflow, a set of possible next tasks can be easily found by looking up the set of causality tables as $\{p_i \mid t_0 \prec p_i$ and $1 \leq i \leq m\}$, where m is the number of tasks found. A method is given in Algorithm 3.3 to provide recommendation. Firstly, the method gives an indicator on each causality table how it matches the given partial workflow. Secondly, for each possible next task, the method gets a recommendation rate by two factors: the weight of the corresponding causality pairs and the indicator of match level. Finally, the method gives recommendation confidence $Conf_i$ for each possible next task:

$$Conf_i = \frac{rate_i}{\sum_{j=1}^{m} rate_j}$$

where $rate_i$ is the recommendation rate for each possible next task.

3.3 Evaluation

The method described in this chapter is evaluated using a Java program for the accuracy of recommendation. The provenance being used in this section are generated with a real scientific workflow from the open scientific workflow repository myExperiment, that is the challenge workflow from the third Provenance Challenge (http://www.myexperiment.org/workflows/750).

Algorithm 3.3 Providing Recommendation

Input: A causality path that end at current task t_0 in the partial workflow $t_n, ..., t_i, ...t_0$;
a set of causality tables $\{T_j \mid 1 \leq j \leq k\}$

Output: A set of possible next tasks with recommendation rates R

1: Let the set of possible next tasks be
$\{p_i \mid t_0 \prec p_i \text{ and } 1 \leq i \leq m\}$
2: **for** $j = 1$ to k **do**
3: $match_j = 1$
4: **for** $i = 1$ to n **do**
5: **if** $\exists\ (t_x \prec t_y, w_{xy}) \in T_j$. $t_x = t_{i-1} \wedge t_y = t_i$ **then**
6: $match_j$++
7: **end if**
8: **end for**
9: **end for**
10: **for** $i = 1$ to m **do**
11: $rate_i = 0$
12: **for** $j = 1$ to k **do**
13: **if** $\exists\ (t_0 \prec p_i, w_i) \in T_j$ **then**
14: $rate_i$+=$w_i \times match_j$
15: **end if**
16: **end for**
17: **end for**
18: **for** $i = 1$ to m **do**
19:
$$R = R \cup \{(p_i, \frac{rate_i}{\sum_{j=0}^{m} rate_j})\}$$
20: **end for**

To evaluate the accuracy of recommendation, the method is applied to each task of workflows. For each task t_i, there are n dependent tasks, and there are p possible next tasks with recommendation confidence. n tasks are picked up from the set of possible next tasks with highest confidence if available, in which there are m tasks matched with one of n dependent tasks, that is m hits out of n real ones. And, it is also m hits out of p recommendations. The accuracy of recommendation for each task is defined as:

$$accuracy_i = \frac{m^2}{n \times p}$$

.

Figure 3.15 compares the method described in this chapter to the methods using only control dependency or only data dependency for recommendation. The α algorithm only mine control dependency while most recommendation algorithms uses only data dependency, this chapter combines both control dependency and data dependency to improve the recommendation accuracy. As shown in Figure 3.15, our method performs better than the method that mines only control dependency, because the data dependency is utilized in the algorithm to assist mining control dependency; and our method performs better than the method that uses only data dependency except in a task, because for some tasks that has only data dependency, our method may give false control dependency, thus lower down the accuracy.

3.4 Related Works

The α algorithm [44] assumes completeness of event logs, this chapter proposes an algorithm based on the α algorithm to use data dependency improving the mining result. There are also a number of process mining algorithms implemented to mine

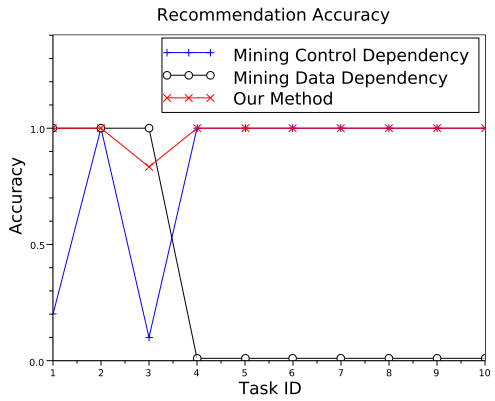


Figure 3.15: Comparison of Recommendation Accuracy for Different Methods

incomplete events logs, such as fuzzy miner, heuristic miner. The fuzzy miner [35] assumes that problems in mining large scale processes are caused by mismatch between fundamental assumptions of traditional process mining, and the characteristics of real-life processes. Fuzzy miner developed an adaptive simplification and visualization technique for process models, which is based on two metrics, significance and correlation. The two metrics are similar to the concept of data clustering domain where a binary distance metric is inferred to find related subsets of attributes. In the context of scientific workflows, significance, which can be determined both for tasks and precedence relations over them, measures the relative importance of behavior. As such, it specifies the level of interest we have in tasks and their control dependency. Correlation is only relevant for precedence relations over tasks, which measures how closely related two events following one another are. The heuristics miner is a practical applicable mining algorithm that can deal with noise, and can be used to express the main behavior (i.e. not all details and exceptions) registered in an event log [36]. It includes three steps: (1) the construction of the dependency graph, (2) for each activity, the construction of the input and output expressions and (3) the search for long distance dependency relations. All those miners do not

utilize data dependency, the algorithm of this chapter can be applied to any of them enhancing the mining result.

This chapter is related to workflow recommendation papers based on provenance. Besides the difference in recommendation technique with other papers, this chapter has a unique advantage that it can build a whole workflow model for general reference. The work in [37], based on large scale databases of workflow execution traces, proposes exploiting these databases with a "knowledge light" approach to reuse, applying case based reasoning (CBR) methods to those traces to support scientists' workflow generation process in two phases. The first phase is retrieving from a database the entries for all workflows containing any one of the current tasks, the second phase is similarity assessment based on the ranking by the size of the largest mapping produced between current tasks and retrieved cases. This chapter uses a different approach to do recommendation for workflow generation, which has two advantages compared with [37]: this chapter does not use expensive graph matching algorithms, thus is more efficient; and this chapter can make recommendation on both data dependency and control dependency while [37] only considers data dependency in their analysis. [45] makes recommendation based on the path in partial workflow, instead of last node in partial workflow. Provenance are synthetically generated from a set of nodes, as a set of node sequences. If there is a path, that has 5 possible following nodes, each of 5 nodes then has 20% confidence, it would be difficult to determine the threshold. [46] proposes a framework for service oriented scientific workflow reuse, its recommendation is based on searching a collection of workflows with the help of annotation. They first collect scientific workflows from centralized repositories such as myExperiment, then integrates annotations generated from various heterogeneous data sources such as author annotations at different levels (for example, workflow, service, or data channels), user comments at runtime,

best practices, and statistical data of existing scientific workflows and services, including popularity and usage patterns. They also support manual annotation. With the collected workflows and integrated annotations, they uses Apache Lucene, an open source search engine, to index the information in collection and associated annotations. Their method can provide relevant information, but cannot suggest a confidence level of each recommendation.

There is also a related work in data mining area that focuses on pairwise temporal patterns [47]. They state the problem of mining event relationships as: given event sequence, finding all pairwise statistically dependent patterns that can be characterized as temporal patterns, that assert dependency between events and specify the timing information, such as "event a happens after event b, say, about 5 minutes". Their result is in fact the precedence table in this chapter. Since this chapter focuses on scientific workflow area, provenance provides workflow running identifier for each event so that it is obvious to get the precedence table, which is the pairwise event dependency in [47]. Combined with algorithms in [47], this chapter can be applied to unstructured data, or semi-structured data, such as computer system log files.

3.5 Summary

This chapter presents a method based on provenance to mine models for scientific workflows, including data and control dependency. The mining result can either suggest part of others' workflows for consideration, or make familiar part of workflow easily accessible, thus provide recommendation support for scientific workflows composition, which offers a new approach to build workflows in the context of scientific workflows. Given the fact that provenance captured in any scientific workflow based systems or system level monitoring systems contains information about tasks

and their temporal order, the proposed algorithm can give both control and data dependency for recommendation during scientific workflows composition. The method provided in this chapter can be applied to any scientific workflow management systems.

CHAPTER 4

MCPATOM: A PREDICTIVE ANALYSIS TOOL FOR ATOMICITY VIOLATION USING MODEL CHECKING

4.1 Overview

Multi-core hardware is a growing industry trend, for both high performance servers and low power mobile devices. Multi-thread programs can exploit multi-core processors at their full potential. In the real world, most servers and high-end critical software are multi-threaded. Unfortunately, multi-thread programs are prone to bugs due to the inherent complexity caused by concurrency. It is difficult to detect concurrency bugs due to the huge number of possible interleavings. Many concurrency bugs escape from testing into software releases and cause some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity [9].

Among different types of concurrency bugs, atomicity violation bugs are the most common one. Atomicity violation bugs are caused by violations to the atomicity of certain code regions without proper synchronization. They widely exist in the real world systems and contributed to about 70% of the examined non-deadlock concurrency bugs [10]. Therefore, techniques for detecting atomicity violation bugs are extremely important.

This chapter presents a dynamic prediction tool McPatom to predict atomicity violation bugs involving a pair of threads accessing a shared variable using model checking, based on binary executables that use POSIX thread library. McPatom uses memory access patterns instead of subroutine atomicity. The only input needed by McPatom is a binary executable, while source code is optional for locating bugs.

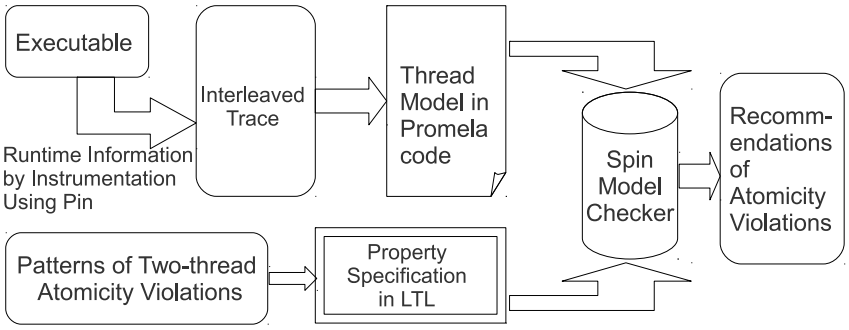


Figure 4.1: Overview of McPatom Framework to predict atomicity violation bugs using model checking

The McPatom framework contains the following major steps: (1) using Pin [48] to instrument an interleaved execution of a multi-thread program and to record an interleaved trace containing only atomicity violation impacting events including all shared variable accesses and all synchronization routines (locks, condition variables, barriers and thread management events); (2) projecting the interleaved trace into a partial order thread model of abstract threads, which maintains the causal relation within actual threads imposed by the synchronization routines; (3) automatically translating the partial order thread model into a Promela program for model checking in Spin [19]; (4) defining a complete set of atomicity violation patterns involving a pair of threads accessing every single shared variable and automatically translating them into temporal logic formulas; (5) using Spin to model check the atomicity violation patterns; and (6) mapping the violation reported in Spin to the execution trace in the original multi-thread program. Figure 4.1 gives an overview of McPatom framework.

Our work makes the following contributions:

1. A method to extract a thread model from an instrumented interleaved trace that only records events related to atomicity violations. Such an interleaved

trace is much smaller than the program behavior in a complete execution. Furthermore the extracted thread model enables the checking of all alternative traces with the same causal relationships as the interleaved trace. The completeness of instrumented interleaved traces and the extracted thread models is proved.

2. A complete set of the patterns of unserializable interleavings involving two threads (most concurrency bugs involve only two threads [11]) containing any number of accesses to a shared variable (either user defined or every word sized dynamically allocated memory accessed by multiple threads). These patterns generalize and cover the three accesses proposed in [10][12]. These atomicity violation patterns become property specifications to be checked.

3. A unique prediction tool - McPatom, for detecting atomicity violation bugs through model checking. McPatom instruments interleaved executions, extracts thread models from interleaved traces, automatically converts (1) thread models into Promela programs and (2) atomicity violation patterns into property specifications. By constraining the checking within a pair of threads involving one shared variable at a time, the interleaving space to be checked is vastly reduced. As a result, McPatom is applicable to large software systems. McPatom can predict atomicity violations that do not manifest during testing or runtime.

We applied McPatom to predict several known atomicity violations in real world systems as well as an atomicity violation that cannot be detected by several existing tools. We obtained favorable experimental results with regard to atomicity violation predictability, accuracy and performance of using McPatom.

4.2 Extracting Partial Order Thread Models from Multi-thread Program Executions

4.2.1 Description of the Partial Order Thread Model

A multi-thread program has a set of threads and a set of shared variables. Shared variables are addresses of global variables and every word sized dynamically allocated memory accessed by multiple threads. The same memory address is considered as another shared variable if it is released and reallocated through the invocations of memory functions. An execution $\sigma = s_1, ..., s_n$ of a multi-thread program P is a sequence of executed statements. A trace is the projection of an execution to a sequence of annotated shared variable accesses and synchronization events. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle tid_i, timestamp_i, action_i \rangle$ in which tid_i is a thread handle, $timestamp_i$ is a time stamp based on real time and $action_i$ is one of the following: (read/write, a shared variable), (a synchronization routine, a synchronization variable) or (a thread management operation, a thread handle). McPatom uses POSIX Threads in which a synchronization routine is a routine related to semaphores, mutex locks, condition variables and barriers, does not handle user-defined synchronization primitives. McPatom also assumes a shared variable as a synchronization variable if it is accessed by synchronization routines, thus does not treat its accesses as shared variable accesses.

Lemma 4. *A trace $\tau = e_1, ..., e_m$ extracted from an execution sequence $\sigma = s_1, ..., s_n$ is sound and complete with respect to σ in terms of atomicity violation predictability.*

Proof. (1) Soundness: An atomicity violation revealed in τ must exist in σ. This is obvious since τ is a projection of σ. An atomicity violation pattern appearing in τ exists in σ.

(2) Completeness: Any existing atomicity violation in σ remains in τ. Since atomicity violations do not depend on general program states, and only depend on the execution orders of shared variable accesses and synchronization events, that are completely captured in τ. □

Definition 10 (Partial Order Thread Model)**.** Given a trace $\tau = e_1, ..., e_m$ containing shared variable accesses and synchronization events, a partial order thread model $(E_\tau, \prec)$ is defined as follows:

1. $E_\tau = \{e_i \mid e_i \; in \; \tau\}$
2. $\prec$ is a partial order relation such that, for any $e_i, e_j \in E$ $(i \neq j)$, $e_i \prec e_j$ iff
   (a) $tid_i = tid_j$ and $i < j$, or
   (b) $tid_i \neq tid_j$, $action_i = (Signal, cvar)$, $action_j = (Wait, cvar)$ and $\forall k \,.\, ((j < k < i) \wedge (action_k \neq (Signal, cvar))$ in which $cvar$ is a condition variable, or
   (c) $tid_i \neq tid_j$, $action_i = (Wait, bvar)$ and $(i < j) \wedge \exists k \,.\, ((tid_k = tid_j) \wedge (k < j) \wedge action_k = (Wait, bvar) \wedge \forall h \,.\, ((tid_h = tid_k) \Rightarrow \neg(k < h < j)))$ in which $bvar$ is a barrier variable, or
   (d) $tid_i \neq tid_j$, $action_i = (Create, tid_j)$, or
   (e) $tid_i \neq tid_j$, $action_j = (Join, tid_i)$.
3. Mutual exclusion: for any $e_i, e_j, e_m, e_n \in E$ $(i \neq j \neq m \neq n)$, $e_j \prec e_m$ or $e_n \prec e_i$ iff

(a) $tid_i = tid_j$, $action_i = (Lock, lvar)$, $action_j = (Unlock, lvar)$, and

(b) $tid_m = tid_n$, $action_m = (Lock, lvar)$, $action_n = (Unlock, lvar)$.

The above partial order relation (or simply causal relation) is similar to the happened-before relation given in [49]. From the above definition, we have (1) shared variable accesses within the same thread are ordered, and (2) a pair of shared variable accesses from two different threads are only ordered if and only if they are constrained by some intermediate synchronization events such as one thread creating the other.

While the partial order thread model $(E_\tau, \prec)$ respects the causal relation in trace τ, it captures an equivalent class of alternative traces that obey the same causal relation as τ, in which each alternative trace τ' is a result of rearranging some shared variable accesses not constrained by $\prec$. The partial order thread model allows us to explore all possible alternative traces that correspond to a set of feasible interleavings in a multi-thread program, however, the model provides an over-approximation without considering data-flow, thus cannot guarantee each permissible trace in the model is covered by some feasible interleaved execution in the multi-thread program P.

4.2.2 Implementation of the Partial Order Thread Model

4.2.2.1 Capturing runtime traces and related source code

McPatom uses Pin binary instrumentation framework [48] to collect runtime trace information, specifically including, every access to every shared variable and every synchronization event using POSIX Thread (locks, condition variables, barriers, thread joining and etc.). For each collected event, McPatom also finds the corresponding source code information including file name and line number. The source

```
3047143104, 1, thread.c-624, Read, threads
3047143104, 1, thread.c-172, Create, 3020999536
3020999536, 1, thread.c-240, Lock, init_lock
3020999536, 1, thread.c-241, Read, init_count
3020999536, 1, thread.c-241, Write, init_count
3020999536, 1, thread.c-242, Signal, init_cond
3020999536, 1, thread.c-243, Unlock, init_lock
```

Figure 4.2: A Sample of a Partial Trace (The format of each line: thread handle, timestamp, file name - line number, action)

code information can be used to help locating the predicted bugs. A sample of a partial trace is shown in Figure 4.2.

4.2.2.2 Automatically encoding traces to Promela code

McPatom uses Spin model checker to detect atomicity violations in a partial order thread model. This section shows how we realize a partial order thread model from a recorded trace in Spin's underlying language Promela.

Defining Shared Variable Accesses McPatom defines every shared variable v as a *short* in Promela, automatically assigns a unique value for all reading accesses and a unique value for all writing accesses in each thread. Formally, let $rw \in \{r, w\}$ and *tid* be thread ID, each access of v is defined as *v=rw+tid*. Since the maximum number of threads per process is limited to 64 in POSIX threads, McPatom sets r to *0*, and w to 64. For example, given two threads: *t1(tid=1)* and *t2(tid=2)*, and a shared variable *v,* McPatom makes the following assignments :

1. v = 64+*1* for each writing access of v in thread *t1*,
2. v = *1* for each reading access of v in thread *t1,*
3. v = 64+*2* for each writing access of v in thread $t2$,
4. v = *2* for each reading access of v in thread *t2.*

```
#define NUM_LOCKS 100
short locked[NUM_LOCKS] = -1;
inline Lock(l) {
 if
 ::atomic{(locked[l] == -1) -> locked[l] = _pid}
 fi;
}
inline Unlock(l) {
 assert(locked[l] == _pid);
 locked[l] = -1;
}
```

Figure 4.3: Promela Code Modeling Mutex Locks

Defining Synchronization Primitives McPatom automatically generates Promela code for all synchronization primitives. Due to space limit, we only present Promela code for mutex locks. McPatom models synchronization events to capture the causal relationships between threads, to prune infeasible interleavings. The Promela code shown in Figure 4.3 models the POSIX Thread routines *pthread_ mutex_ lock* and *pthread_ mutex_ unlock*. The atomic construct groups indivisible statements together to ensure no interleaving within an atomic sequence. *Lock* inline function accepts a lock l as its argument. If lock l is not locked, *Lock* function locks it and sets the owner to the thread that is the predefined variable *_pid* for the executing process in Promela. If lock l is in locked status, no guards are executable so that the thread is blocked until lock l is available according to Promela semantics. *Unlock* inline function simply sets lock l to unlocked status. It is exactly what is required to model locking and unlocking of a mutex lock.

Defining Threads All events with regard to a particular thread from the recorded trace are grouped into a Promela process in which each event is represented by its corresponding Promela code defined in previous steps as shown in Figure 4.4. Since the maximum number of threads per process in POSIX threads is 64, which is well

```
proctype t1() { ... }
proctype t2()
{
  Lock(init_lock);     /* thread.c - 240 */
  init_count = 0 + 2;     /* thread.c - 241 */
  init_count = 64 + 2;     /* thread.c - 241 */
  Signal(init_cond);     /* thread.c - 242 */
  Unlock(init_lock);     /* thread.c - 243 */
  ...
}
init
{
  run t2();     /* thread.c - 172 */
  ...
}
```

Figure 4.4: A Sample of Partial Promela Code

below the maximum number (256) of processes allowed in Promela, we do not have a problem to encode all possible threads occurring in a recorded trace. The interleaved execution of processes in the Promela program generates all alternative permissible traces in the partial order thread model.

4.3 Defining and Encoding Unserializable Interleaving Patterns between Two Threads

Atomicity is a semantic correctness property for concurrent programs. A thread interleaving is serializable if and only if it is equivalent to a serial execution, which executes a code region without other threads interleaved in between. The code region is typically enforced as atomic explicitly in the code. When proper synchronization is missing to enforce atomicity, atomicity violation bugs may occur. [50] proved that a thread interleaving is serializable if and only if its conflict graph is acyclic.

Most concurrency bugs involve two threads, instead of a large number of threads, based on the study in [11], in which 101 out of 105 bugs involved only two threads. Thus atomicity violation bugs in a multi-thread program can be explored through every pair of threads. Our work is inspired by the works in [10][12], which addressed a special case of unserializable interleavings with three accesses of the same shared variable. However, as Figure 4.5 shows, there are real world bugs involving four accesses of the same shared variable. Furthermore, there can be more accesses involved, such as reading accesses of a shared variable for logging purpose. The patterns given in this chapter cover atomicity violation bugs involving any number of accesses of a shared variable between a pair of threads.

4.3.1 Three-access and Four-access Atomicity Violation

Many recent works focused on three-access atomicity violations [10][12][11], which involve one shared variable, two threads and three accesses to the variable. For simplicity, two threads are referred as a local thread (Thread 1) and a remote thread (Thread 2), the opposite view is also explored during the detection process. If two consecutive accesses of a shared variable in a local thread are interleaved with an access to the variable from a remote thread, the interleaving is a potential unserializable one. In practice, unserializable interleavings indicate the presence of atomicity violation bugs. The explanation of unserializable interleavings of three accesses and many real world atomicity violation bugs can be found in [10].

Three-access atomicity violations are chosen by tools above because (1) there are many real world atomicity violation bugs involving only three accesses, and (2) checking only two accesses (current access and previous access) in a thread can reduce the complexity of algorithms. However, some atomicity violation bugs

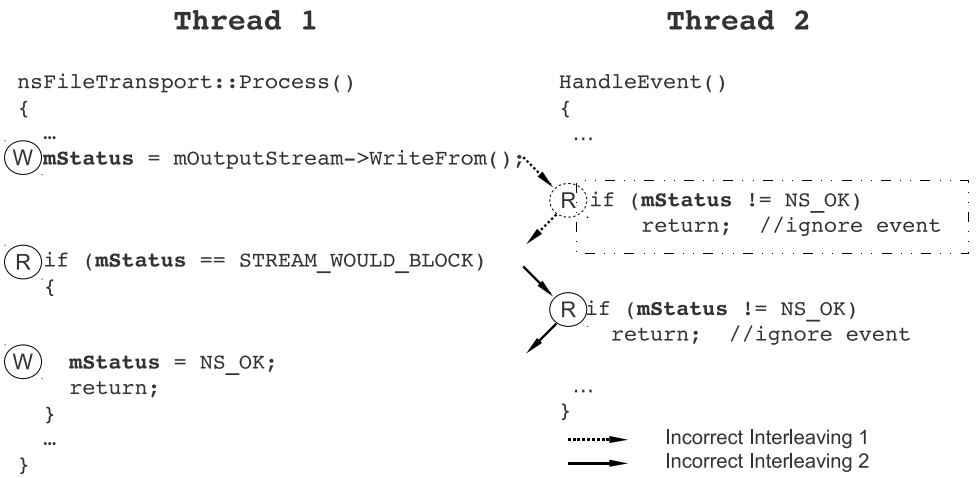


Figure 4.5: A four-access atomicity violation bug [51] in Mozilla (Incorrect interleaving 1 was detected by PSet [51] and missed by AVIO [10], while incorrect interleaving 2 cannot be detected by either PSet or AVIO.)

involve more than three accesses. A real world example [51] is shown in Figure 4.5. The shared variable accesses in Thread 1 must be in an atomic region; otherwise, a possible interleaving may result in HandleEvent function of Thread 2 returning with a missing event. PSet [51] detected this bug (incorrect interleaving 1) since PSet keeps track of either the last writer or the set of last readers for every memory location. However PSet cannot detect the mutant of the bug (incorrect interleaving 2) because in PSet's view the mutant only involves a set of last readers and the current reading access. AVIO [10] cannot detect this bug because it involves more than three accesses.

4.3.2 Patterns of Two-thread Atomicity Violations involving Any Number of Accesses

In the sequel, a two-thread atomicity violation refers to a two-thread atomicity violation involving any number of accesses of a shared variable, and $A \in \{Read, Write\}$,

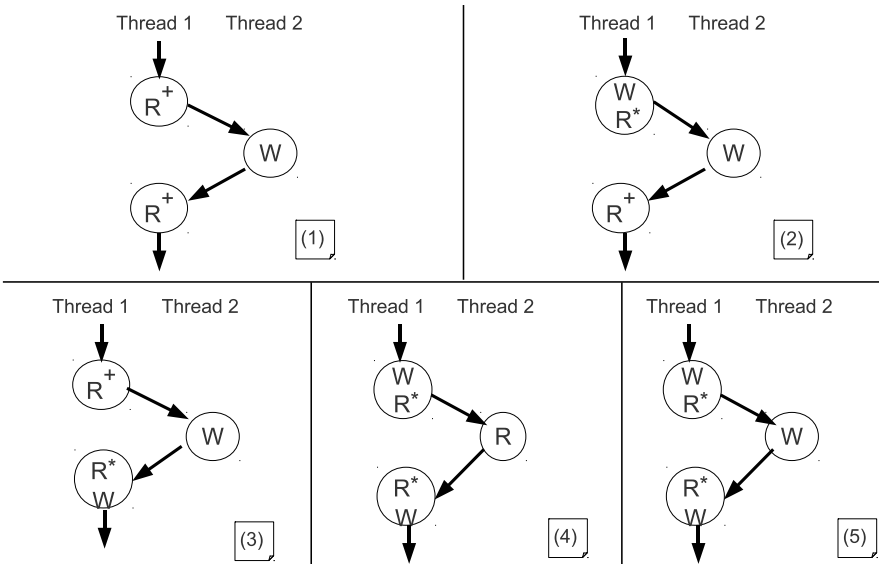


Figure 4.6: Unserializable Interleavings with two threads. In (1)(2)(3)(5), W in Thread 2 unexpectedly changes the value; In (4), An intermediate value in Thread 1 is read by Thread 2.

$R = Read$, $W = Write$, A^* denotes zero or more A, A^+ denotes one or more A, R^* denotes zero or more R and R^+ denotes one or more R. This section gives a set of patterns covering all possible two-thread atomicity violations.

Figure 4.6 shows all possible scenarios of unserializable interleavings with only one access from Thread 2. If any of the unserializable interleaving patterns is matched, it indicates a potential atomicity violation.

Theorem 2 (Completeness of the set of Patterns in Figure 4.6)**.** *The set of patterns in Figure 4.6 is complete, i.e. they cover all possible unserializable interleavings between two threads.*

Proof. Let $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be a sequence of atomic accesses in an interleaved execution of two threads, in which $A_i^{t_i}$ ($t_i \in \{1, 2\}$, $A_i^{t_i} \in \{Read, Write\}$, $1 \leq i \leq n$) denotes an atomic access from thread t_i to the same shared variable. Let every subsequence of $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be of the form B_1^1, B_2^2, B_3^1 where B_1^1 and B_3^1 of Thread 1

are sequences of $A_i^{t_i}(t_i = 1)$, B_2^2 of Thread 2 is a sequence of $A_i^{t_i}(t_i = 2)$. Let P_i be pattern i. B_2^2 is assumed to be or can be reduced without losing writing operations to a single access A_2^2. If B_1^1, A_2^2, B_3^1 does not match with any of the patterns in Figure 4.6, B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4 \wedge \neg P_5$. Since operator $\wedge$ is commutative, we can select a specific order and carry out an incremental analysis of possible B_1^1, A_2^2, B_3^1 based on each of $P_i(1 \leq i \leq 5)$.

1. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1$. B_1^1, A_2^2, B_3^1 can only be one of the following:

   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^+$

   (b) $B_1^1 = A^+$, $A_2^2 = W$, $B_3^1 = A^*WA^*$

   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$

2. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2$. B_1^1, A_2^2, B_3^1 can only be one of the following:

   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$

   (b) $B_1^1 = A^+$, $A_2^2 = W$, $B_3^1 = A^*WA^*$

   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$

3. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3$. B_1^1, A_2^2, B_3^1 can only be one of the following:

   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$

   (b) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B3 = A^*WA^*$ which is equivalent to above one.

   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$

4. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$. B_1^1, A_2^2, B_3^1 can only be one of the following:

(a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$

(b) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$

(c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$

5. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4 \wedge \neg P_5$. B_1^1, A_2^2, B_3^1 can only be one of the following:

   (a) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$

   (b) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$

According to the Serializability Theorem [50], an interleaved sequence is serializable if and only if its conflict graph is acyclic. Either 5(a) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$ or 5(b) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$ is serializable. Therefore, the completeness of the set of patterns in Figure 4.6 is proved. □

4.3.3 Automatically encoding atomicity violation patterns into Linear time Temporal Logic (LTL) Formulas

For every shared variable and every pair of threads $t1$ and $t2$, McPatom automatically defines a LTL formula (4.1) for each pattern in Figure 4.6 and another LTL formula (4.2) reversing the view of t_1 and t_2. Let v be a shared variable, $r = 0$ and

$w = 64$ as defined in section 4.2.2.2, $A_i \in \{r, w\}$, and tid_i , $\overline{tid_i} \in \{1, 2\}$.

$$
\begin{aligned}
&[]! <> ((v == A_1 + tid_1)\&\& \\
&\quad X((v == A_2 + tid_2)U((v == A_3 + tid_3)\&\& \\
&\quad X((v == A_4 + tid_4)U(v == A_5 + tid_5))))) \\
&[]! <> ((v == A_1 + \overline{tid_1})\&\& \\
&\quad X((v == A_2 + \overline{tid_2})U((v == A_3 + \overline{tid_3})\&\& \\
&\quad X((v == A_4 + \overline{tid_4})U(v == A_5 + \overline{tid_5})))))
\end{aligned}
\tag{4.1}
\tag{4.2}
$$

where "[]" denotes *Always*, "!" denotes *Logical Negation*, "<>" denotes *Eventually*, "X" denotes *Next* and "U" denotes *Until*. These formulas specify that the atomicity violation patterns do not occur.

Using Figure 4.6 (2) as a concrete example, one formula in LTL is shown below:

$$
\begin{aligned}
&[]! <> ((v == w + 1)\&\& \\
&\quad X((v == r + 1)U((v == w + 2)\&\& \\
&\quad X((v == w + 2)U(v == r + 1)))))
\end{aligned}
\tag{4.3}
$$

$(v == w+2)U(v == r+1)$ is true if and only if $v == w+2$ holds until $v == r+1$ is true or simply $v == r + 1$ holds without $v == w + 2$ holds. This subformula captures $W_2^* R_1^+$ in which W_2^* means zero or more writing accesses from Thread 2, R_1^+ means one or more reading accesses from Thread 1. Furthermore, $(v == w+2)\&\&X((v == w+2)U(v == r+1))$ captures $W_2^+ R_1^+$ and $(v == r+1)U((v == w + 2)\&\&X((v == w + 2)U(v == r + 1)))$ reflects $R_1^* W_2^+ R_1^+$. Therefore, (4.3) captures $[]! <> W_1 R_1^* W_2^+ R_1^+$ and ensures that pattern $W_1 R_1^* W_2 R_1^+$ in Figure 4.6 (2) does not occur in the partial order thread model. The reason that the LTL formula contains W_2^+ instead of W_2 is that there can be synchronization events between W_2 and R_1^+, for each of those events, W_2 needs to hold.

4.4 Predictive Analysis of Atomicity Violation using Model Checking

In this section, we discuss McPatom framework's general merits in terms of its soundness and completeness as well as specific ways in using Spin model checker [19] to show its applicability.

4.4.1 Soundness and completeness of McPatom

An important feature of a prediction method is its capability to predict as many violations as possible. Since the majority of existing prediction methods uses an abstract model extracted from one interleaved execution at a time from a multi-thread program, a prediction method's capability rests on the quality of the abstract model built and its thoroughness in exploring the permissible traces in the abstract model. McPatom extracts the least constrained partial order thread model respecting the causal relation from the observed interleaved execution and uses model checking to explore all permissible traces in the partial order thread model.

Theorem 3. *McPatom ensures the completeness of its prediction - any possible atomicity violation involving a pair of threads accessing one shared variable in the partial order thread model can be detected.*

Proof. McPatom encodes all possible atomicity violation patterns involving a pair of threads accessing one shared variable (Theorem 2) into linear time temporal logic formulas. McPatom uses model checking to exhaustively check whether any temporal logic formula fails in the partial order thread model. Thus none of possible atomicity violation will be undetected. □

In general, McPatom cannot guarantee the soundness of its prediction, i.e., each predicted atomicity violation is covered by a feasible execution, since data-flow is ignored in the partial order thread model.

One major potential problem using model checking is the state explosion problem. Fortunately, the state explosion problem will not occur in atomicity violation prediction due to the following reasons (1) the partial order thread model (capturing only shared variable accesses and synchronization events) used for model checking is drastically smaller compared to the original multi-thread program, (2) each atomicity violation pattern to be checked involves only one shared variable, and (3) checking each atomicity violation pattern does not depend on the value of the shared variable. Another possible problem with model checking is the potential exponential number of possible interleavings due to the number of threads involved and the number of shared variable accesses. This problem is partially resolved (1) due to our focus on checking atomicity violations involving only two threads, (2) due to the constraints imposed by causal relations that drastically reduce the number of potential interleavings generated by the number of shared variable accesses, and (3) due to our implementation strategies of grouping all reading event sequences in each thread into atomic blocks in Spin to achieve partial order reductions and enforcing the wait/signal order of condition variables in the observed execution while exploring alternative interleavings. Our experiment results show very good performance using model checking.

```
70: proc  2 (t13) spin_av.pml:551 (state 28) [sharedvariable
   = (0+13)]
72: proc  3 (t48) spin_av.pml:591 (state 31) [sharedvariable
   = (64+48)]
76: proc  2 (t13) spin_av.pml:552 (state 29) [sharedvariable
   = (0+13)]
```

Figure 4.7: A Sample of Atomicity Violation Trace Reported by Spin

4.4.2 Using Spin model checker to find atomicity violation traces

McPatom selects Spin model checker [19] based on its maturity, popularity, and capability. Spin is used to check every atomicity violation freedom property involving every pair of threads accessing every single shared variable one at a time in the partial order thread model extracted from a single interleaved trace recorded through instrumentation using Pin. Based on the partial order thread model encoded in Promela in section 4.2.2.2, and the atomicity violation freedom property encoded in LTL formulas in section 4.3.3, McPatom uses Spin to find atomicity violation traces or report no atomicity violations. Figure 4.7 gives an example of atomicity violation reported by Spin, which is mapped to real code in the original program.

Spin can be configured to search all errors or stop at the first error. McPatom chooses to stop at the first error, thus McPatom reports no atomicity violation if there exists no atomicity violation; when McPatom reports some atomicity violation traces, there may be additional atomicity violations not yet reported, which can be detected by re-running McPatom after grouping the previously reported violation related accesses into an atomic region so that it will not cause a new violation in the next run. For each shared variable and each pair of threads, an atomicity violation is recorded in a Spin trail file for each pattern if it exists. The Spin trail file can be

```
sharedvar=0+13; /*mod_log_config.c-1353*/  |if (len+buf->outcnt>LOG_BUFSIZE)
 sharedvar=64+48; /*mod_log_config.c-1373*/|    buf->outcnt += len;
sharedvar=0+13; /*mod_log_config.c-1369*/  |s = &buf->outbuf[buf->outcnt]
```

Figure 4.8: Promela code and the corresponding real code in the original program

Table 4.1: Bug List

Bug #	Program	Issue Number
1	Apache	25520
2	Apache	21287
3	Apache	21285
4	MySQL	644
5	MySQL	791
6	Mozilla-extract	Figure 4.5

simulated by Spin to give a clear view of those accesses involved in the atomicity violation, as shown in Figure 4.7.

4.4.3 Mapping the violations reported in Spin to the original program

Atomicity violations reported in Spin, as shown in Figure 4.7 as an example, are mapped to real code in original program. McPatom automatically identifies the related lines in Promela files, in which the comments of each line in Promela are file names and line numbers of the corresponding source code. Figure 4.8 shows the Promela code at the left and the corresponding real code at the right, for the atomicity violation in Figure 4.7.

4.5 Evaluation

We have used several real-world systems with known bugs listed in Table 4.1 (the issue numbers are the IDs in corresponding Bugzilla Databases) ([10],[51]) to examine

Table 4.2: Performance

	Program	Program Input	Trace Size (MB)	Time to Check (mins)	Number of Shared Variables	Number of Properties	Average Time per Property (secs)
1	fft	-p2 -m1024	4.3	304	3656	36560	0.499
2	fmm	Particles : 64 Processors : 2	10.8	183	1248	12480	0.88
3	lu	-p2 -n16	0.3	0.44	5	50	0.53
4	radix	-p2 -n10	3.7	328	3094	30940	0.636
5	Apache	2 concurrent httperf	9.4	15.68	151	3360	0.005

Table 4.3: Performance (Continue)

	Program	The Shared Variable with Maximum Number of Accesses		
		Number of Accesses	Number of States	Time to Check (secs)
1	fft	1041	3294	0.04
2	fmm	20064	9996	0.08
3	lu	282	941	0.02
4	radix	81	433	0.01
5	Apache	1415	16	less than 0.01

our tool's bug prediction capability, as well as four programs [10] without atomicity violations in SPLASH-2 parallel benchmark suite [52] to test the accuracy of our tool (no false positives are reported).

Bug prediction capability

McPatom has successfully predicted all the known bugs listed in Table 4.1, especially bug number 6 - an extraction of a real world atomicity violation bug reported in [51], which evades PSet [51] because this bug involves a set of last readers and the current reading access, and AVIO [10] because this bug involves more than three accesses.

Accuracy

We have chosen four programs (also used in [10]) without atomicity violations in SPLASH-2 parallel benchmark suite [52] to test whether McPatom produces violation predictions, which would certainly be false positives. McPatom passed this test without reporting any violations.

Performance

Since McPatom framework uses model checking as the underlying atomicity violation prediction method and relies on a third party tool, Spin, to perform the model checking, it is extremely important to demonstrate the applicability of McPatom. We conducted the experiments[1] on a PC with dual core 2.33GHz CPU and 2GB memory. Performance data are given in Table 4.2 and Table 4.3, where time to check included automatically running Spin, compiling generated pan.c and model checking properties for all shared variables. There are ten properties to check for

[1]Data available at http://users.cs.fiu.edu/~rzeng001/spin12/

each pair of threads accessing a shared variable based on five violation patterns and their mutants. Apache program contains more than two threads and results in more properties to be checked. Instrumentation overhead was similar to that given in [10]. Table 4.3 shows the shared variable with maximum number of accesses in each program. From Table 4.2 and Table 4.3, it shows that the number of states does not explode when the number of accesses increases since checking the shared variable with maximum number of accesses took less than 0.01 seconds (not including the time to run Spin and compile generated pan.c) while checking any shared variable on average took 0.005 seconds. These preliminary experimental results are very encouraging and demonstrate the scalability of McPatom. These results also confirm our belief that although the total number of possible interleavings to check can explode quickly as the number of accesses increase; however, the number of actual interleavings are drastically smaller due to the constraints imposed by causal relationships between threads. Other major reasons, which also vastly reduce the possible interleavings, are that McPatom takes advantage of the nature of atomicity violations and considers only a pair of threads and accesses to a single shared variable at one time, groups all reading event sequences in each thread into atomic blocks in Spin to achieve partial order reductions , and enforces the wait/signal order of condition variables in the observed execution while exploring alternative interleavings. Table 4.2 and Table 4.3 show that the experiment with Apache has even better performance than others, due to Apache's heavy use of condition variables. Since atomicity violations involving a single shared variable can be checked independently from violations involving other shared variables, we can significantly reduce the duration (not the cumulative time) of model checking by using multiple machines.

4.6 Related Works

There are many recent works on tackling atomicity violations. Some works proposed techniques to detect atomicity violations on actual program executions through testing [53] or runtime monitoring ([10], [54], and [55]). Other works developed methods to predict atomicity violations that may evade testing and runtime monitoring. In this section, we mention some recent works most relevant to ours on dynamically predicting atomicity violations. Most of these works share the following fundamental process: (1) instruments a multi-thread program P to record atomicity relevant events, (2) extracts a trace τ of atomicity relevant events from an interleaved execution σ of P, (3) projects trace τ into a partial order model M based on a causal relation defined on P, (4) explores various alternative trace τ' in M to predict potential atomicity violations in a possible corresponding interleaved execution σ' in P. Various methods and their supporting tools differ with regard to the strategies used in the above process.

How to abstract a partial order model M from a trace τ is critical. If the model is too restrictive, many feasible atomicity violations cannot be explored. If the model is too permissible, the prediction may not be sound, i.e. a predicted atomicity violation may not be a feasible interleaved execution of P. Penelope [56] ignores some causal relationships in building a partial order model and thus requires additional feasibility checking of a predicted atomicity violation. Fusion [12] abstracts a partial order model called concurrent trace program (CTP) that ignores the causal relation between different threads. Linearized atomicity violation traces in CTP are symbolically checked with additional order information from source codes to ensure their feasibility. In [57], a theoretical study was conducted to analyze the complexity of predicting atomicity violations, in which two simplified partial order models

are considered. The first one ignores all synchronization and the second one only considers lock-based synchronization. It shows the tradeoffs between efficiency and accuracy. jPredictor [58] defines a partial order model based on a concept of sliced causality and lock-atomicity, which may predict some infeasible violations. Our work abstracts a partial order model respecting the causal relationships imposed by all synchronization constructs, but without considering data-flow, our work also may produce some infeasible violations.

A variety of techniques have been proposed to explore atomicity violation traces from an abstract partial order model. CTrigger [11] and Penelope [56] developed different algorithms to generate potential violation schedules and to prune away many infeasible ones. However these algorithms may report infeasible atomicity violation traces as well as miss feasible ones. jPredictor [58] uses model checking to exhaustively check a property in the partial order model and is capable to predict other concurrency bugs in addition to atomicity violations. Fusion [12] encodes the partial order model, the source program, and three access atomicity violation patterns into a logic formula; and uses a satisfiability modulo theory solver to check the feasible interleavings for atomicity violations. Our work converts the partial order model into a Promela program, defines a complete set of atomicity violation patterns as temporal logic formulas, and then uses Spin model checker to produce atomicity violation traces.

4.7 Summary

Concurrency bugs are extremely hard to detect using testing techniques due to huge interleaving space. This chapter presents a tool McPatom using model checking to predict atomicity violation concurrency bugs. McPatom is powerful and can

explore a vast interleaving space of a multi-threaded program based on a small set of instrumented test runs. McPatom is applicable to large real-world systems.

McPatom focuses on atomicity violations involving each single shared variable, and thus cannot find atomicity violations involving multiple variables. Another limitation is that redundant model checking may be performed if two recorded interleaved traces yield the same partial order thread model.

CHAPTER 5

METHODS FOR IMPROVING THE COVERAGE AND PRECISION OF MCPATOM

5.1 Overview

Multi-threaded programs are prone to bugs due to concurrency. Concurrency bugs are hard to find and reproduce because of the large number of interleavings. Most non-deadlock concurrency bugs are atomicity violation bugs due to unprotected accesses of shared variables by multiple threads. Existing approaches for detecting atomicity violation are either static or dynamic. Static approaches [59] usually suffers from a large number of false positives due to the complexity of analyzing concurrency and pointer aliasing. Dynamic approaches are either monitor based methods that require atomicity violations to manifest during monitored runs [11][54][55], or predictive methods that explore atomicity violations in alternative interleavings from some observed runs [56][12][57][58].

Predictive methods use either (1) under-approximate models ([60][61][58][62]) by analyzing only interleavings with the same read-after-write relationships as in the observed executions, which are a subset of all feasible interleavings; or (2) over-approximate models ([63][56][64][65][66]) by exploring not only all feasible interleavings but also infeasible interleavings due to data constraints and ad-hoc synchronization, which may produce false positives. Table 5.1 shows ten interleaving scenarios of three accesses to a shared variable between two threads that will result in atomicity violations, among which only five can be predicted by methods using under-approximate models while the other five are missed because some of read-after-write relationships within three accesses are broken. Hence methods based on under-approximate models have inadequate coverage, and methods based on over-

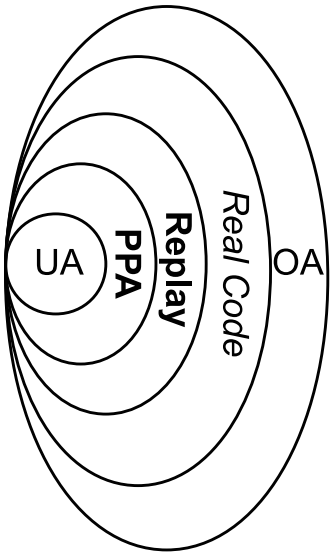


Figure 5.1: Comparison with other predictive methods on coverage and precision, in which each oval stands for the traces that can be generated in the corresponding method as explained below.
UA - Under-approximate methods [60][61][58][62].
PPA - Post-prediction analysis method in this chapter, e.g. Figure 5.3.
Replay - Methods of rescheduling predicted violation traces, e.g. Figure 5.9(c).
Real code - Real program code, captured in Concurrent Trace Programs [12].
OA - Over-approximate methods [63][56][64][65][66], e.g. Figures 5.2, 5.4, 5.8, and 5.9(b).

approximate models are not precise. Many predictive methods mentioned above explored the tradeoffs between precision and coverage.

This chapter presents two methods for improving the coverage and precision of atomicity violation predictions: 1) a post-prediction analysis method on relaxing the under-approximate models to increase coverage while ensuring precision; 2) a follow-up replaying method to further increase coverage. The post-prediction analysis method is lightweight and fast, and makes the precise predictions and achieves better coverage than other existing methods using under-approximate models. A comparison with other methods is given in Figure 5.1.

Table 5.1: Limited coverage of prediction using under-approximate models for two threads (T1 and T2)

	Observed Execution $T1$ $T2$	Predicted Execution $T1$ $T2$	Description of Unserializability or Missed Reason
Covered	R R W	R W R	Two reading accesses read from different writes
	R W W	R W W	Forwarded writing access in T2 is overwritten
	W W W	W W W	Forwarded writing access in T2 is overwritten
	R W W	W R W	An intermediate value is read
	W W W	W W W	Forwarded writing access in T1 is overwritten
Missed	W R W	None	Intra-thread read-after-write in T1 prohibits interleaved writing in T2
	W W R	None	Inter-thread read-after-write prohibits forwarded reading in T2
	W R R	None	Inter-thread read-after-write prohibits forwarded reading in T1
	W W R	None	Intra-thread read-after-write in T1 prohibits interleaved writing in T2
	W R W	None	Inter-thread read-after-write prohibits forwarded reading in T1

5.2 Preliminaries

A multi-threaded program has a set of threads and a set of shared variables. An observed execution $\sigma = s_1, ..., s_n$ of a multi-threaded program P is a sequence of executed statements. A trace is the projection of an execution to a sequence of annotated shared variable accesses and synchronization events. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle seq_i, tid_i, action_i, br_i \rangle$ in which seq_i is an increasing sequence number, tid_i is a thread handle, $action_i$ is either an atomic shared variable access or a synchronization event, and br_i is the number of branches between e_i and its immediate preceding event within the same thread. Given a trace $\tau = e_1, ..., e_m$, a partial order thread model $(E_\tau, \prec)$ can be defined, where E_τ is the set of events occurring in τ and $\prec$ is a causal relation on E_τ. The causal relation $\prec$ respects all constraints of synchronization primitives and thread-local program orders. Sequential consistency is assumed in this chapter, as it is typically accepted by other related works. A feasible atomicity violation prediction in sequential consistent memory models is also feasible in other memory models.

Definition 11. A predicted atomicity violation in an interleaved trace τ' in $(E_\tau, \prec)$ is a true violation if and only if it is contained in a feasible execution σ'.

The strength of the causal relation $\prec$ affects the size of possible interleaved traces in $(E_\tau, \prec)$. When the same read-after-write relation in τ is enforced in $\prec$, any predicted atomicity violation trace τ' is feasible. Such partial order thread models are under-approximate and may miss feasible interleaved traces. On the other hand, not enforcing the same read-after-write relation in τ within $\prec$ results in over-approximate thread models that contain all feasible interleaved traces as well as infeasible ones.

5.3 Post-prediction analysis

Methods using under-approximate models make precise (only feasible) atomicity violation predictions but have limited coverage (missing other feasible atomicity violations). This section introduces a post-prediction analysis method to improve the coverage while ensuring precision. The under-approximate models can be relaxed to become over-approximate models through removing the read-after-write relations imposed by the observed execution. Our post-prediction analysis method works on over-approximate models to remove false positives while achieving more coverage than methods using under-approximate models. This analysis method is general and is applicable to the prediction results from other methods using over-approximate models. The only information needed is an observed trace τ and three memory accesses in τ that forms an atomicity violation pattern in a predicted alternative trace [63][12].

5.3.1 Data constraints causing false predictions

Data constraints concern data dependencies that may make a predicted atomicity violation trace infeasible, such as the branch conditions that are dependent on shared variables and queue accesses that are dependent on shared indexing variables. Figure 5.2 gives an example of data constraints that need to be taken into consideration when analyzing an atomicity violation prediction. Figure 5.2(a) shows a trace of an observed execution, in which shared variable *index* is read in line 7 and line 8 after a writing of *index* in line 3, and hence there are data dependencies in two pairs of accesses to *index*: line 3 and line 7, line 3 and line 8. Figure 5.2(b) shows a trace of a predicted atomicity violation, in which line 10 has a writing access to the shared memory *item* in Thread T2 between the reading (line 1R) access and writing access

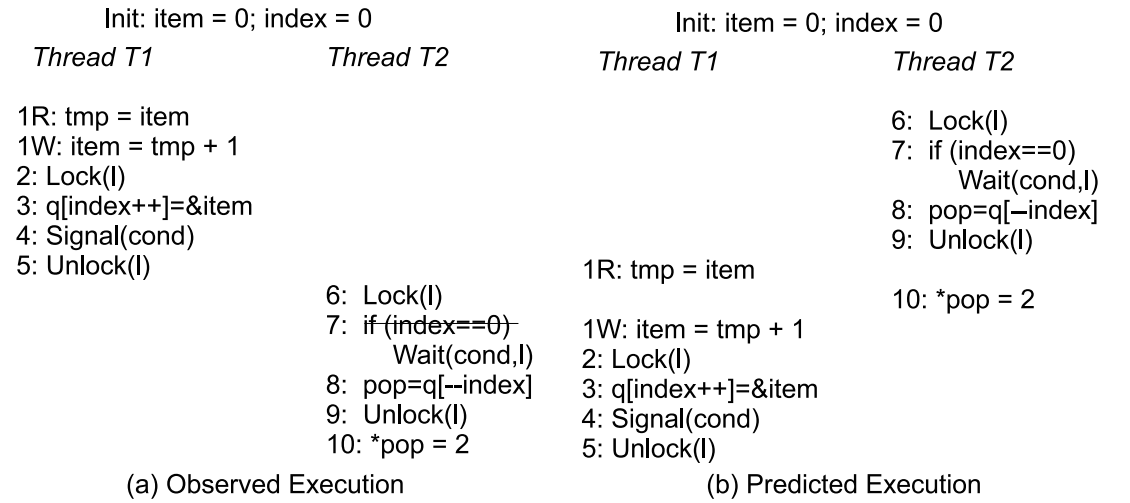


Figure 5.2: An example of data constraint analysis for false positives (extracted from Apache)

(line 1W) in Thread T1. However, both pairs of accesses to *index* above are broken, which makes the memory access in line 10 in the observed trace infeasible in the predicted atomicity violation trace.

A perfect solution to the above problem requires a precise and complete partial order thread model extracted from the observed trace. The precision ensures the feasibility of any predicted atomicity violation in the partial order thread model, and the completeness requires any feasible atomicity violation remain in the partial order thread model. Enforcing all the read-after-write relations can ensure the precision of the partial order thread models. Several methods [61][62] introduced the read-after-write relations as a simple solution to ensure the precision. However, the constraints imposed by read-after-write relations are too strong, thus make the resulting partial order thread model over restrictive and under-approximate. Figure 5.3 shows an example in which a real bug is missed if all the read-after-write relations are enforced, because the reading access can be moved forward to read from a different writing access.

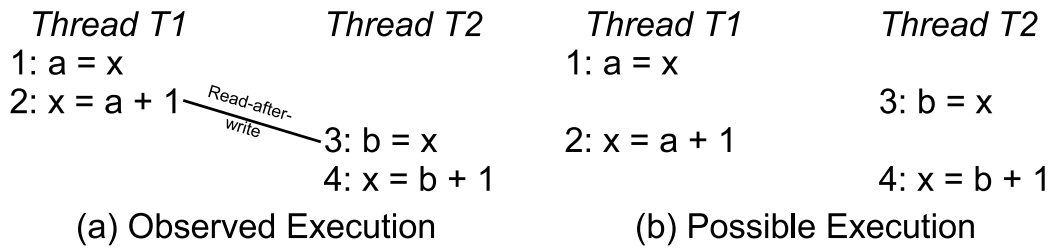


Figure 5.3: A real bug is missed due to a read-after-write relationship

5.3.2 Ad-hoc synchronization causing false predictions

Ad-hoc synchronization is often used to ensure an intended execution order of certain memory accesses. Specifically, instead of calling condition variable routines or using other synchronization primitives, programmers often use ad-hoc loops to synchronize a shared variable. A trace containing an ad-hoc synchronization includes a sequence of reading accesses and a writing access, in which there is also a read-after-write relationship as the data constraints discussed above. Figure 5.4 gives an example of false positives related to an ad-hoc synchronization. (a) is an observed trace, in which there is a sequence of reading accesses from line 2 to n+2 where $n \geq 0$, and a writing access in line n+1. Line n+2 reads after the writing in line n+1. (b) shows a predicted atomicity violation trace, in which n+3R is $a^1_{i'}$, line 1W is $a^2_{j'}$, and line n+3W is $b^1_{k'}$. Line n+2 is a moved forward reading r^1 as the case (1) in Lemma 6, which breaks the read-after-write relationship between line n+1 and line n+2 in the observed trace in (a). It is obvious the atomicity violation trace (b) is infeasible. Thus the read-after-write relations in ad-hoc synchronization need to be enforced. We treat ad-hoc synchronizations as a special case of data constraints discussed in Section 5.3.1.

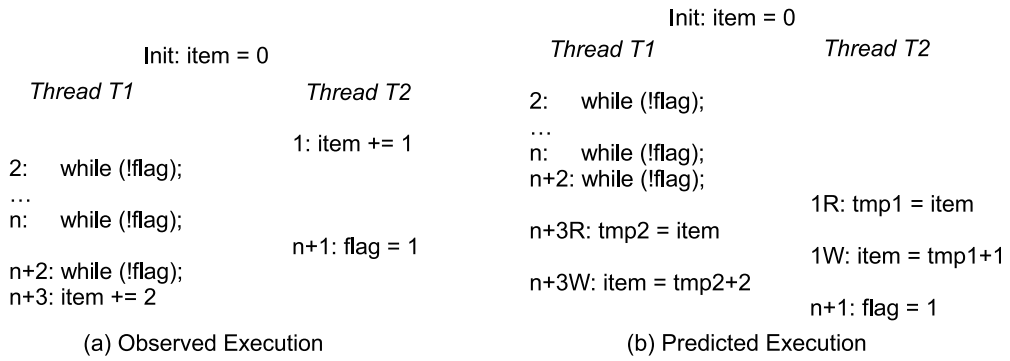


Figure 5.4: A false positive related to an ad-hoc synchronization

5.3.3 Problem formulation

The method proposed in this section aims at avoiding false atomicity violation predictions while catching as many real bugs as possible. Our method works on over-approximate models to remove false positives while achieving more coverage than methods using under-approximate models.

During post-prediction analysis, any predicted atomicity violation trace is an alternative interleaving respecting the same causal relations imposed by the synchronization events as the original observed trace. Thus we can view a trace as a sequence of atomic (reading or writing) accesses without synchronization events to simplify the discussion. Let $\tau = a_1^{t_1}, a_2^{t_2}, ..., a_n^{t_n}$ be a sequence of atomic accesses to share variables in an interleaved execution of two threads, in which a superscript indicates the thread an event belongs to, thus $t_i \in \{1, 2\}$ for $1 \leq i \leq n$; and a subscript indicates the occurrence position of an event in the interleaved trace.

Over-approximate methods in [63][11][56] were based on three-access atomicity violation patterns $a_{i'}^1, a_{j'}^2, a_{k'}^1$, where $a_{i'}^1$ and $a_{k'}^1$ are atomic accesses to a shared variable x in thread 1 and $a_{j'}^2$ is an atomic access to x in thread 2. Table 5.1 gives all possible scenarios that will result in atomicity violation patterns after reordering the event in thread 2 to occur between the two events in thread 1.

A predicted atomicity violation trace in over-approximate methods is $\tau' = ..., a^1_{i'}, ..., a^2_{j'}, ..., a^1_{k'}, ...$ with atomicity violation pattern $a^1_{i'}, a^2_{j'}, a^1_{k'}$ which are three consecutive accesses to a shared variable x. τ' is the result of reordering some accesses in a given original observed trace τ such that (1) $\tau = ..., a^1_i, ..., a^1_k, ..., a^2_j, ...$ or (2) $\tau = ..., a^2_j, ..., a^1_i, ..., a^1_k, ...$, and thus may break the read-after-write relations in τ. Note that accesses other than $a^1_{i'}, a^2_{j'}, a^1_{k'}$ are not explicitly identified in τ' but may also be reordered due to reordered $a^1_{i'}, a^2_{j'}, a^1_{k'}$, and the exact positions of i', j', k' in τ' are not important. The corresponding i, j, k are the exact positions where three accesses to x occurred in τ. τ may contain many other accesses to shared variables including x. τ' is considered feasible if its prefix up to $a^1_{k'}$ is feasible since anything happens after $a^1_{k'}$ does not affect the feasibility of τ'. Not all broken read-after-write relations due to reordering affect the feasibility of τ', but some does.

5.3.4 Our method

The underlying idea of our method is checking whether any reordered event due to reordered $a^1_{i'}, a^2_{j'}, a^1_{k'}$ may break read-after-write relations in the original trace. Before reordering, a^2_j may happen after a^1_k, or before a^1_i. The idea of our method is explained below assuming a^2_j happens after a^1_k, i.e. $a^1_i \dashrightarrow a^1_k \dashrightarrow a^2_j$, in Figures 5.5, 5.6 and 5.7. w and r are used to describe a read-after-write relationship with regard to a shared variable other than the one in $a^1_{i'}, a^2_{j'}, a^1_{k'}$. In Figure 5.5, a reading event r^2 is moved forward due to reordering of a^2_j, thus breaking the read-after-write relationship between w^1 and r^2.

In Figures 5.6 and 5.7, $Prev(a^2_j)$ denotes the immediate preceding access to the same shared variable as a^2_j. In Figure 5.6, due to reordered $a^1_{i'}, a^2_{j'}, a^1_{k'}$, $Prev(a^2_j)$ is

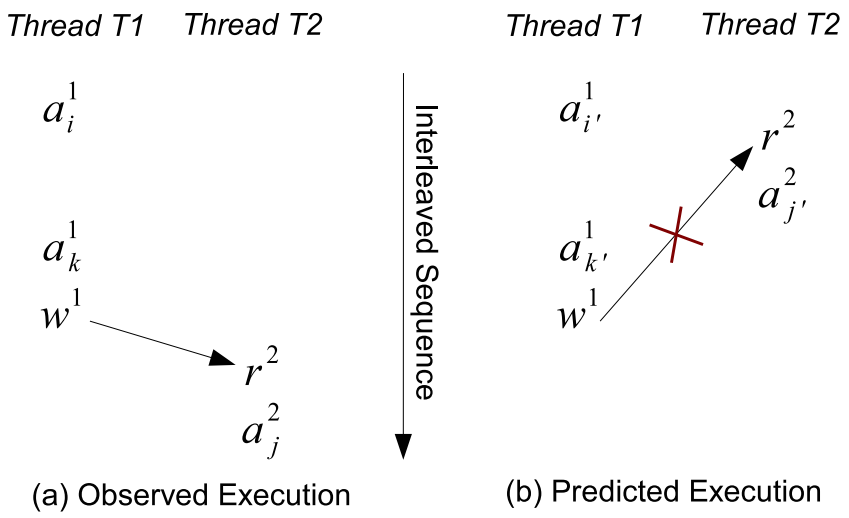


Figure 5.5: Read-after-write relationship is broken, assuming $a_i^1 \dashrightarrow a_k^1 \dashrightarrow a_j^2$ and a moved forward reading event before $a_{k'}^1$.

moved forward to happen before $a_{i'}^1$, thus r^2 is moved forward to happen before w^1, causing the read-after-write relationship between w^1 and r^2 is broken.

In Figure 5.7, due to reordered $a_{i'}^1, a_{j'}^2, a_{k'}^1$, $Prev(a_j^2)$ is moved forward to happen before $a_{i'}^1$, thus w^2 is moved forward to happen before r^1 instead of happening after r^1, causing the read-after-write relationship between r^1 and its original defining writing access is broken.

Based on ideas above, Lemmas 5 and 6 identify all cases in which a reordered event may affect the feasibility of τ'. Let $\tau(a, b)$ be accesses in τ that occur after a and before b, $\tau[a, b)$ be accesses in $\tau(a, b)$ including a, and $\tau(a, b]$ be accesses in $\tau(a, b)$ including b, $a \dashrightarrow b$ denote event a occurs before event b, $Prev(a^i)$ denote the immediate preceding atomic access to the same shared variable as a in thread i, and $Next(a^i)$ denote the immediate succeeding atomic access to the same shared variable as a in thread i.

Lemma 5. *Given a predicted atomicity violation trace* $\tau' = ..., a_{i'}^1, ..., a_{j'}^2, ..., a_{k'}^1, ...$ *with atomicity violation pattern* $a_{i'}^1$,$a_{j'}^2, a_{k'}^1$ *with regard to a shared variable* x, *and*

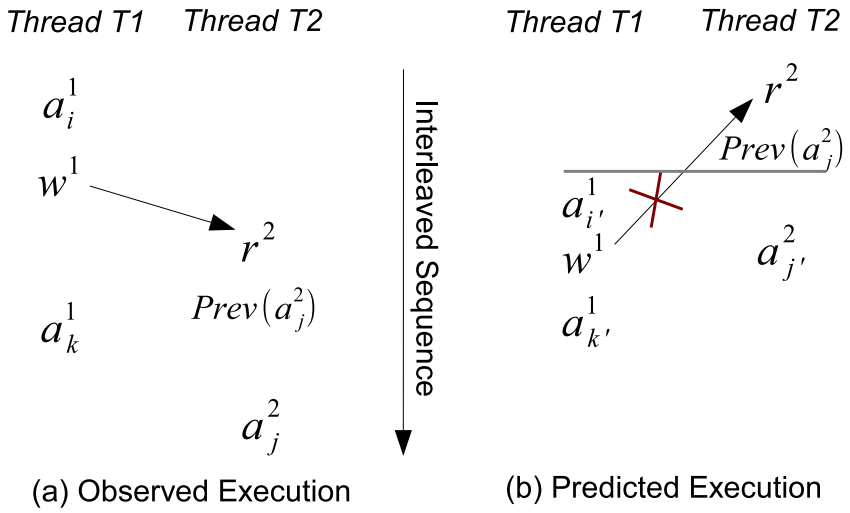


Figure 5.6: Read-after-write relationship is broken, assuming $a_i^1 \dashrightarrow a_k^1 \dashrightarrow a_j^2$ and a moved forward reading event before $a_{i'}^1$.

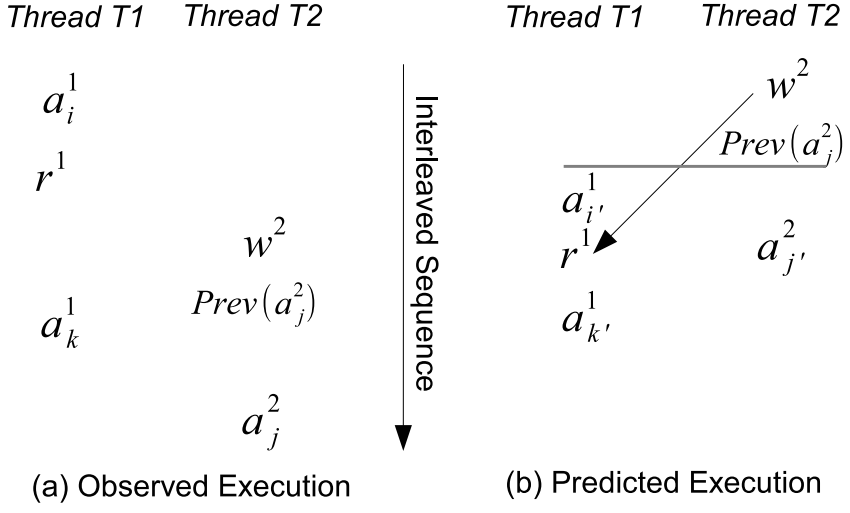


Figure 5.7: Read-after-write relationship is broken, assuming $a_i^1 \dashrightarrow a_k^1 \dashrightarrow a_j^2$ and a moved forward writing event.

the original observed trace $\tau = ..., a_i^1, ..., a_k^1, ..., a_j^2, ...$ τ' *may be infeasible due to a violated data constraint (a broken read-after-write relationship) caused by one of the following cases (1) a moved forward reading event in thread 2:* $r^2 \in \tau(a_k^1, a_j^2)$ *and* $r^2 \dashrightarrow a_{k'}^1$*; (2) a moved forward reading event in thread 2:* $r^2 \in \tau(a_i^1, Prev(a_j^2)]$ *and* $r^2 \dashrightarrow a_{i'}^1$*; or (3) a moved forward writing event in thread 2:* $w^2 \in \tau(a_i^1, Prev(a_j^2)]$*,* $w^2 \dashrightarrow a_{i'}^1$ *and there is some branch instruction between* $\tau[a_i^1, a_k^1)$*.*

Proof. Given the observed trace $\tau = ..., a_i^1, ..., a_k^1, ..., a_j^2, ...,$ to obtain the violation trace $\tau' = ..., a_{i'}^1, ..., a_{j'}^2, ..., a_{k'}^1, ...,$ all access events in thread 2 between k and j are moved before k' (we can assume $k' = j' + 1$ since the violation trace is reported as soon as a violation pattern occurs, and thus do not need to consider any thread 2 event after j). Some access events in thread 2 between i and k may be moved before i' (we can assume the last thread 2 event need to move before i' is $Prev(a_j^2)$ since the violation pattern is reached as long as there is no other access to x in thread 2 between i' and j'). We analyze all such needed moves and their impact below: (1) for a moved forward reading event: $r^2 \in \tau(a_k^1, a_j^2)$ and $r^2 \dashrightarrow a_{k'}^1$: if there is a writing event w^1 in thread 1 accessing the same shared variable as r^2 such that $w^1 \in \tau(a_k^1, a_j^2)$ and $w^1 \dashrightarrow r^2$ in τ. The read-after-write relationship $w^1 \dashrightarrow r^2$ in τ is broken since $a_{k'}^1 \dashrightarrow w^1$ in τ'. As a result, the new value of r^2 may make $\tau'[a_{j'}^2, a_{k'}^1]$ infeasible; (2) a moved forward reading event in thread 2: $r^2 \in \tau(a_i^1, Prev(a_j^2)]$ and $r^2 \dashrightarrow a_{i'}^1$: if there is a writing event w^1 in thread 1 accessing the same shared variable as r^2 such that $w^1 \in \tau(a_i^1, Prev(a_j^2)]$ and $w^1 \dashrightarrow r^2$ in τ. The read-after-write relationship $w^1 \dashrightarrow r^2$ in τ is broken since $a_{i'}^1 \dashrightarrow w^1$ in τ'. As a result, the new value of r^2 may make $\tau'[a_{i'}^1, a_{j'}^2]$ infeasible; (3) a moved forward writing event in thread 2: $w^2 \in \tau(a_i^1, Prev(a_j^2)]$ and $w^2 \dashrightarrow a_{i'}^1$: if there is a reading event $r^1 \in \tau[a_i^1, a_k^1)$ in thread 1 accessing the same shared variable as w^2 such that $r^1 \dashrightarrow w^2$ in τ'. This new read-after-write relationship may break the old read-after-write relationship of

r^1. However, the new value of r^1 does not affect the execution of any thread 2 event within $\tau(r^1, a^1_{k'})$ in τ', but may affect the execution of some thread 1 event between $\tau[r^1, a^1_{k'}]$, which can happen in two cases: (i) if the new value of r^1 is used in some branch instruction between $\tau[a^1_i, a^1_k)$; (ii) if the new value of r^1 directs the access at $a^1_{k'}$ to a different shared variable when the memory address of the access at $a^1_{k'}$ depends on the value of r^1, a new atomicity violation pattern r'^1, w^2, r^1 on shared variable y occurs, which makes τ' a feasible atomicity violation trace. As a result, the new value of r^1 may make $\tau'[a^1_{i'}, a^1_{k'}]$ infeasible only if the new value of r^1 is used in some branch instruction between $\tau[a^1_i, a^1_k)$. □

Note a moved forward writing event in thread 2: $w^2 \in \tau[Prev(a^2_j), a^1_k)$ and $w^2 \dashrightarrow a^1_{k'}$ may break read-after-write relationships after $a^1_{k'}$, but does not affect the feasibility of τ'.

Figure 5.2 shows an example of case (1) in Lemma 5, where the predicted atomicity violation trace τ' in (b) is an infeasible alternative interleaving of the original observed trace τ in (a). In (b) line 1R is $a^1_{i'}$, line 10 is $a^2_{j'}$, line 1W is $a^1_{k'}$, and line 7 is the moved forward reading r. Its read-after-write relationship with line 3 is broken. As a result, the condition in line 7 is true and *Wait* is executed that makes τ' infeasible.

Lemma 6. *Given a predicted atomicity violation trace* $\tau' = ..., a^1_{i'}, ..., a^2_{j'}, ..., a^1_{k'}, ...$ *with atomicity violation pattern* $a^1_{i'}$,$a^2_{j'}, a^1_{k'}$ *with regard to a shared variable x, and the original observed trace* $\tau = ..., a^2_j, ..., a^1_i, ..., a^1_k, ...$ τ' *may be infeasible due to a violated data constraint (a broken read-after-write relationship) caused by one of the following cases (1) a moved forward reading event in thread 1:* $r^1 \in \tau(a^2_j, a^1_i]$ *and* $r^1 \dashrightarrow a^2_{j'}$*; (2) a moved forward reading event in thread 1:* $r^1 \in \tau(Next(a^2_j), a^1_k)]$ *,* $r^1 \dashrightarrow Next(a^2_{j'})$*, and there is some branch instruction between* $\tau[a^1_i, a^1_k)$*.*

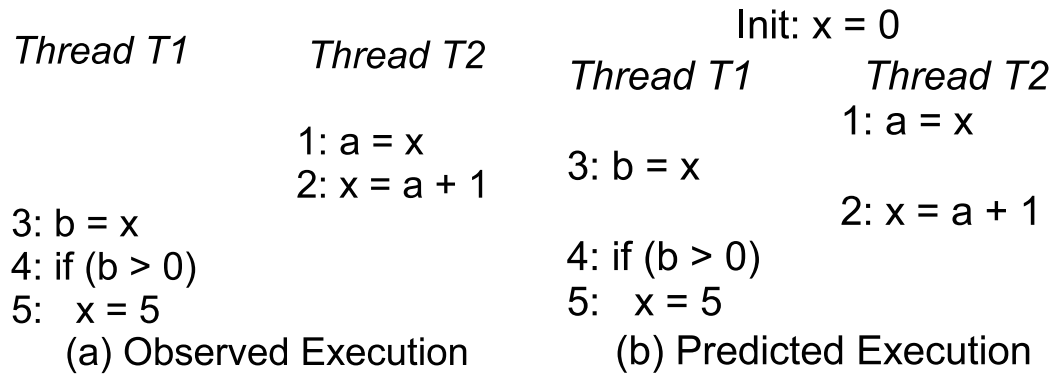


Figure 5.8: A false positive due to local dependency

Proof. The proof of case (1) is similar to that in Lemma 5 and is omitted here. In case (2), only a thread 1 reading event $r^1 \in \tau(Next(a_j^2), a_k^1)]$ needs to be moved forward to reach the violation pattern such that $Next(a_j^2)$ appears after $a_{k'}^1$ in τ'. However, the new value of r^1 does not affect the execution of any thread 2 event in $\tau(r^1, a_{k'}^1)$ in τ', but may affect the execution of some thread 1 event between $\tau[r^1, a_{k'}^1]$, which can only happen if the new value of r^1 is used in some branch instruction between $\tau[a_i^1, a_k^1)$ as shown in the proof of Lemma 5. As a result, the new value of r^1 may make $\tau'[a_{i'}^1, a_{k'}^1]$ infeasible. □

Note any moved forward writing event in thread 1 does not affect the feasibility of τ'.

Figure 5.8 shows an example of case (1) in Lemma 6, where the predicted atomicity violation trace τ' in (b) is an infeasible alternative interleaving of the original observed trace τ in (a). In (b), line 3 is $a_{i'}^1$, line 2 is $a_{j'}^2$, line 5 is $a_{k'}^1$, and line 3 is the moved forward reading $r^1 \in \tau(a_j^2, a_i^1]$, which broke the old read-after-write relationship from line 2, and now reads a new value 0. As a result, $b_{k'}^1$ will not be executed and thus τ' is infeasible.

Figure 5.3 shows another example, which is not infeasible according to of case (1) in Lemma 6.

Lemmas 5 and 6 define the necessary conditions that a violated data constraint (a broken read-after-write relationship) can occur and thus makes a predicted atomicity violation trace infeasible. Thus Lemmas 5 and 6 have ensured that any surviving predicted atomicity violation trace is a feasible one. Our post-prediction analysis method ensures precision while eliminating only a subset of predicted atomicity violation traces breaking the read-after-write relations in the original observed trace.

5.3.5 Algorithm of post-prediction analysis

An observed trace contains a sequence of events, and each event is defined by a thread identifier *tid*, a memory access type (read or write) *rw*, a shared variable *var*, and the number *br* of branches between this event and its immediate preceding event within the same thread. Other fields in an observed trace are omitted here without affecting the post-prediction analysis. An atomicity violation prediction is based on an atomicity violation pattern $a^1_{i'}$, $a^2_{j'}$, $a^1_{k'}$ involving two threads 1 and 2. The algorithm shown in Algorithm 5.1 analyzes the feasibility of a predicted violation according to Lemmas 5 and 6. Five true returns in the algorithm correspond to the five cases in Lemmas 5 and 6.

5.4 Replay

Post-prediction analysis on predicted violation traces, while making prediction precise and reducing possible missing real bugs, may still miss real bugs due to the lacking of complete data dependencies. Replaying validates a predicted violation execution trace by orchestrating thread scheduling in a concrete execution for a given input, can be used to alone to eliminate infeasible traces, or used after post-prediction analysis to validate the feasibility of uncertain traces.

Algorithm 5.1 Algorithm of post-prediction analysis

Input: $\tau : seq \rightarrow (tid_{seq}, rw_{seq}, var_{seq}, br_{seq})$, and three seq: $...a_i^1...$, $...a_j^2...$, $...a_k^1...$ that contain accesses relevant to a violation pattern $a_{i'}^1$, $a_{j'}^2$, $a_{k'}^1$ in τ'.

Output: Whether a predicted violation maybe infeasible.

if $a_j^2 > a_i^1$ **then**
  $prev \leftarrow max(seq)$ where $tid_{seq} = 2 \wedge var_{seq} = var_{a_j^2} \wedge seq < a_j^2$
  for $r \in (a_i^1, prev] \cup (a_k^1, a_j^2) \wedge rw_r = read \wedge tid_r = 2$ **do**
    $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r$
    if $r \in (a_i^1, prev] \wedge w > a_i^1 \wedge tid_w = 1$ **then**
      return $True$
    end if
    if $r \in (a_k^1, a_j^2) \wedge w > a_k^1 \wedge tid_w = 1$ **then**
      return $True$
    end if
  end for
  for $r \in [a_i^1, a_k^1) \wedge rw_r = read \wedge tid_r = 1$ **do**
    $w = min(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq > r \wedge tid_w = 2$
    if $w \leq prev \wedge \exists seq \,.\, (r < seq < a_k^1) \wedge (tid_{seq} = 1) \wedge br_{seq} > 0$ **then**
      return $True$
    end if
  end for
end if
if $a_j^2 < a_i^1$ **then**
  for $r \in (a_j^2, a_i^1] \wedge rw_r = read \wedge tid_r = 1$ **do**
    $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r$
    if $w \geq a_j^2 \wedge tid_w = 2$ **then**
      return $True$
    end if
  end for
  $next \leftarrow min(seq)$ where $tid_{seq} = 2 \wedge var_{seq} = var_{a_j^2} \wedge seq > a_j^2$
  for $r \in (a_i^1, a_k^1) \wedge rw_r = read \wedge tid_r = 1$ **do**
    $w = max(seq)$ where $rw_{seq} = write \wedge var_{seq} = var_r \wedge seq < r \wedge tid_w = 2$
    if $w > next \wedge \exists seq \,.\, (r < seq < a_k^1) \wedge (tid_{seq} = 1) \wedge br_{seq} > 0$ **then**
      return $True$
    end if
  end for
end if
return $False$

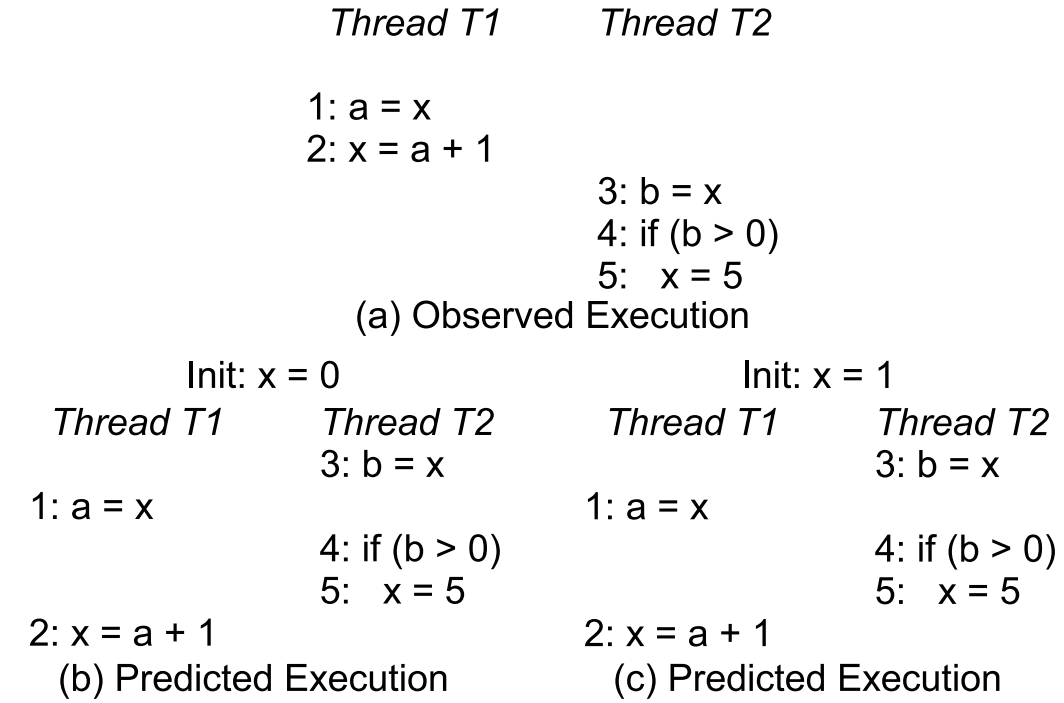


Figure 5.9: An example of replay related to data constraints

Figure 5.9 gives an example of data constraints, in which x is a shared variable. In the figure, (a) is an observed execution in which there are no interleaved accesses between line 1 and line 2; (b) and (c) are violation traces predicted based on over-approximate methods [63]. Both (b) and (c) break the read-after-write relationships between line 2 and line 3, and are classified as uncertain (maybe infeasible) traces by post-prediction analysis. During replay, (b) is recognized as a false positive since line 5 cannot be executed because the branch condition is not satisfied, and (c) is confirmed as violation trace.

Given atomicity violation trace $\tau' = ..., a^1_{i'}, ..., a^2_{j'}, ..., a^1_{k'}, ...$ with atomicity violation pattern $a^1_{i'}, a^2_{j'}, a^1_{k'}$, we insert two signal-wait pairs in the following way: a *signal* after $a^1_{i'}$, a *wait* before $a^2_{j'}$, a *signal* after $a^2_{j'}$ and a *wait* before $a^1_{k'}$. The memory access order will be enforced to let atomicity violation manifest; however, as shown in Figure 5.10(a), there can be a deadlock even for a feasible prediction.

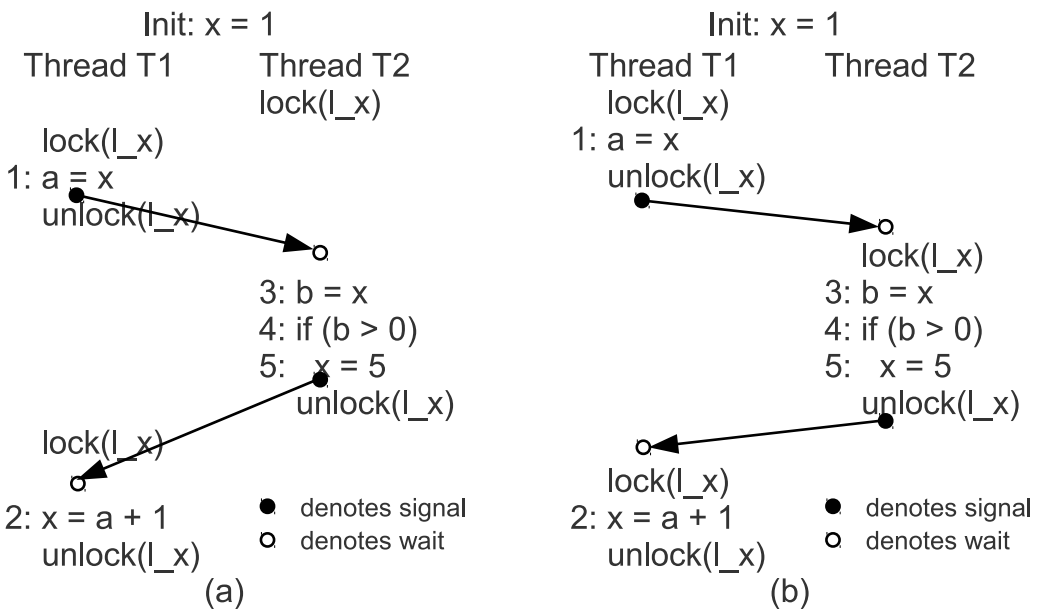


Figure 5.10: Replaying need considering mutex

Thread $T2$ firstly acquires the lock then wait for a signal, while thread $T1$ cannot acquire the lock thus not be able to issue a signal, as a result, the two threads cannot make any progress and run into a deadlock. Therefore, mutex locks need to be taken into consideration when inserting instructions for enforcing the predicted interleaving. Let L_1^1 and L_2^2 be two sets of locks acquired before $a_{i'}^1$ and between $a_{i'}^1$and $a_{j'}^2$ respectively, $L = L_1^1 \cap L_2^2$, and let $FirstLock(L)$ be the instruction for the first lock acquiring in L, $LastUnlock(L)$ be the instruction for the last lock releasing in L. If $L \neq \emptyset$, instead of inserting a signal-wait pair as described earlier, we insert a *signal* after $LastUnlock(L)$ and a *wait* before $FirstLock(L)$. We define and insert another signal-wait pair similarly between $a_{j'}^2$ and $a_{k'}^1$. An example is shown in Figure 5.10(b).

A potential problem with above inserted signal-wait pair is that the modified program may run into deadlocks, livelocks and missing memory accesses, as shown in

Figure 5.11. Figure 5.11(a) shows a deadlock, a signal-wait pair is inserted between line 3 and 2, and there exists an ad-hoc synchronization implemented by a shared variable *done*. $T2$ is waiting on $T1$ to set the flag *done*, while $T1$ is waiting on $T2$ as required by the predicted interleaving, it results a circular wait. Figure 5.11(b) shows a missing memory access, the predicted violation trace expects the memory access in line 2 interleaves between memory accesses in line 3 and line 5. However, line 5 together with inserted instructions are not executed at all as the condition in line 4 is not satisfied, so replaying misses a memory access and is not able to observe the predicted interleaving. Figure 5.11(c) shows a livelock in which $T1$ can not make progress while $T2$ continues as normal, because line 5 together with inserted instructions are not executed.

A simple solution is to use timeout mechanism to detect deadlocks and livelocks caused by inserted signal-wait instructions, and to check signal-wait pairs in observed replaying to detect missing memory accesses. If a deadlock, or a livelock or some missing memory accesses are detected, the predicted interleaving is not feasible and is marked as a false positive; otherwise, replaying continues as usually without a large number of context switches.

5.5 Experiments and Evaluation

We have implemented the proposed algorithm in a prototype tool based on the tool in [63] and conducted several experiments. The tool is automatic such that it only requires running a use case of the target executable as the manual step. During our experiments for the replaying method, predicted traces of known atomicity violations in [63] and Table 5.3 can be validated through replaying successfully, while all other predicted violation traces cannot be replayed due to data constraints. Our ex-

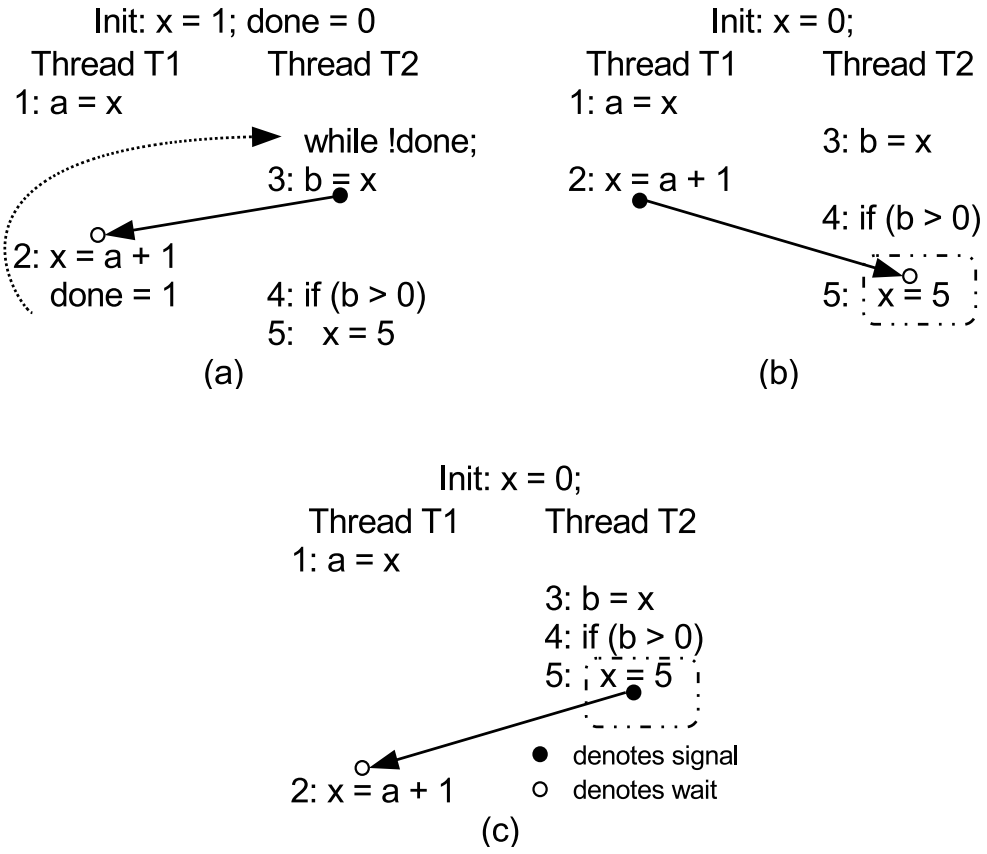


Figure 5.11: False positives pruned out by replaying

Table 5.2: Experimental Results using Apache and FFmpeg

	Program-Size	Events_in_Trace	OA	PPA	PPA-time
Apache	1.5 MB	140532	155	1	12.1 sec
FFmpeg	41 MB	550352	29	0	11.6 sec

periments for post-prediction analysis PPA used the benchmarks in [12], Apache web server and FFmpeg audio/video codec library. The sequel in this section discusses the experiments for PPA.

The results of experiments using Apache and FFmpeg are shown in Table 5.2, and the results of experiments using the benchmarks in [12] are shown in Table 5.3.

In Table 5.2, Apache has a known atomicity violation bug but FFmpeg does not. The first column *Program-Size* gives the size of the executable, the second column *Events_in_Trace* lists the number of events in the trace; the third column *OA* contains the number of prediction by the over-approximate method McPatom; the fourth column *PPA* is the number of prediction by post-prediction analysis PPA; the last column *PPA-time* is the time in seconds to perform post-prediction analysis.

In Table 5.3, Programs *atom001* and *atom002* have atomicity violations that are extracted from a real bug [10]. Their modified versions without atomicity violations are *atom001a* and *atom002a*. Other programs are Linux/Pthreads/C implementation of the parameterized bank example [67], in which program *bank-av-8* has atomicity violations; program *bank-sav-8* adds a condition variable as a partial fix without avoiding all atomicity violations for any shared variable; and program *bank-nav-8* adds a transaction lock to remove all atomicity violations. The first three columns provide the statistics of programs, in which *svars-causing-av* is the number of shared variables that cause atomicity violations. The next three columns provide the statistics of our method, which uses the results of an over-approximate method McPatom [63]. *OA-svars* is the number of shared variables that cause atom-

Table 5.3: Experimental results compared to CTP and UA methods

Program			Our method			CTP[12]			Other
name	threads	svars-causing-av	OA-svars	PPA-svars	PPA-time	hb-pavs	sym-avs	sym-time	UA-avs
atom001	3	1	1	1	0.06	2	1	0.03	0
atom001a	3	0	1	0	0.06	2	0	0.03	0
atom002	3	1	1	0	0.04	34	33	20.4	0
atom002a	3	0	1	0	0.05	34	0	17.6	0
bank-av-8	9	8	8	8	0.73	32	32	2.5	0
bank-sav-8	9	8	8	8	0.79	32	16	4.6	0
bank-nav-8	9	0	8	0	0.83	32	0	140.6	0

icity violations predicted by over-approximate methods, *PPA-svars* is the number of shared variables predicted by post-prediction analysis PPA that cause atomicity violations and *PPA-time* is the running time in seconds. The last three columns are statistics provided in [12], in which *hb-pavs* is the number of predicted atomicity violation traces and *sym-avs* is the number of feasible atomicity violation traces out of *hb-pavs* obtained from using symbolic method CTP. Note that a single shared variable may generate many possible atomicity violations traces, which can often be fixed in a single fix. We count shared variables in *PPA-svars* that have at least one feasible predicted violation trace. The last column *UA-avs* is the number of predicted atomicity violation traces by under-approximate methods that enforce all read-after-write relations.

Lightweight and fast

The running times in Table 5.3 show our method's scalability is promising compared to that of the symbolic method CTP. When the size of programs grows, e.g. *bank-nav-8* contains more code than others, the formulas built in CTP also grow bigger and require more time to be solved. Our method stops as soon as a broken read-after-write relation defined in Lemmas 5 or 6 is detected, incurs insignificant time increase when the size of a program grows, and thus can handle much larger programs.

Our method is also evaluated using the complete Apache web server and FFmpeg audio/video codec library, as shown in Table 5.2, in which the running times show the scalability of our method is promising for large scale software.

Precise predictions and better coverage

The results show that our method reports no false positives while reporting more real bugs than under-approximate methods. Thus our method is precise and improves coverage. One shared variable in *atom002* is missed due to read-after-write relations of accesses to other shared variables. Our method cannot decide whether it is feasible because the value of a shared variable or a local variable depending on the value of a shared variable affects the feasibility. [12] collects and encodes all program information in CTP and thus can detect it.

5.6 Related Works

5.6.1 Post-prediction analysis

The post-prediction analysis method in this chapter achieves precision and improves coverage by reducing the number of missing real bugs compared to other precise methods. Under-approximate models such as [61][58][62] admit only interleavings with the same read-after-write relations as in the observed executions to achieve precision; however, the constraints imposed by read-after-write relations are too strong, thus make the model over restrictive and may miss real bugs. Over-approximate models such as [63][56][64][65][66] admit not only all feasible interleavings but also infeasible interleavings due to data constraints and ad-hoc synchronization, thus make prediction imprecise due to false positives. [60] allows broken read-after-write relations but prohibits the thread with such a read event to continue, hence can be considered as an under-approximate model.

CTP [12] is an analysis tool applicable to the predicted violation traces generated by over-approximate methods, thus is the most relevant work to ours. CTP

achieves precision and complete coverage by using the values of shared variables and local variables in the predicted atomicity violation trace, which requires heavy instrumentation and the static analysis of source code of expressions. Our method explores ways to ensure precision and to improve coverage while avoiding heavy instrumentation and the static analysis of source code.

5.6.2 Replay

Penelope [56] instruments the scheduler to follow a predicted schedule, from which it gets a set of threads and the number of steps that each thread should take before next context switch. Similar to us, the way it counts steps is also based on the events that were monitored during an observed execution. Only after execution reaches the point that the violation pattern is executed, the scheduler releases all threads to execute as they normally do. Thus, before it reaches the point, it has to pay the same overhead as an observed execution, and in addition the overhead of instrumenting scheduler.

Maple [68] memoizes tested interleavings and tries to expose untested interleavings for a given test input to increase interleaving coverage. The predicted untested interleavings are exposed by controlling the thread schedule during execution for the test input. In Maple, the active scheduler takes the test input and forces all threads to run on a single processor, and therefore records the order of the thread schedule.

CHESS [53] is a systematic and deterministic testing tool for concurrent programs. It takes complete control over scheduling of threads. However, its scheduler is non-preemptive , therefore cannot model the behavior of a real scheduler that may preempt a thread at any point in its execution.

Existing works mentioned above need heavy context switches. However, even following exactly the same schedule of a predicted atomicity violation trace using heavy context switches cannot guarantee perfect replaying. Perfect replaying is impossible without capturing all sources of nondeterminism, as demonstrated in [69][70][71]. Our method reduces context switches to the minimal level by allowing nondeterminism while trying to ensure the determinism of events related to predicted atomicity violations. In case of a large number of predicted atomicity violation traces, our method performs post-prediction analysis first to effectively and significantly reduce the number of replays needed.

5.7 Summary

Predictive methods for atomicity violations need to consider the tradeoffs between precision and coverage. This chapter presented a post-prediction analysis method and a replaying method to ensure the precision and improve the coverage of predicted atomicity violation traces generated from over-approximate methods. The post-prediction analysis method covers all ten scenarios in Table 5.1. The replaying method reduces context switches to the minimal level to improve scalability. Figure 5.1 compares our methods with other predictive methods on coverage and precision, in which our post-prediction analysis method PPA improves coverage while ensuring precision, our replaying method further improves coverage and ensures precision as well. Both methods does not rely on the instrumentation of local variables and the analysis of source code. Therefore, our methods are scalable and applicable to large programs.

CHAPTER 6

CONCLUSION

6.1 Summary

This dissertation presents methods and tools for modeling and analyzing concurrent software systems at design and code levels, to improve reliability of concurrent software. At design level, we build a formal specification of Mondex using Petri nets, and provide a way of using model checking to verify the formal specification of Mondex, including the abstract model and concrete model. We also develop methods to mine traces to build Petri nets automatically to aid designing scientific workflows. At code level, we develop methods and tools to predict atomicity violation bugs using binary instrumentation and model checking techniques.

Our method for mining traces to build Petri nets is based on provenance of scientific workflows, and mine both data and control dependency. The mining result can either suggest part of others' workflows for consideration, or make familiar part of workflow easily accessible, thus provide recommendation support for scientific workflows composition, which offers a new approach to build workflows in the context of scientific workflows. Given the fact that provenance captured in any scientific workflow based systems or system level monitoring systems contains information about tasks and their temporal order, the proposed algorithm can give both control and data dependency for recommendation during scientific workflows composition.

Our tool McPatom, using model checking to predict atomicity violation concurrency bugs, is powerful and can explore a vast interleaving space of a multi-threaded program based on a small set of instrumented test runs. McPatom is applicable to large real-world systems. Predictive methods for atomicity violations need to consider the tradeoffs between precision and coverage. Our post-prediction anal-

ysis method and our replaying method are presented to ensure the precision and improve the coverage of predicted atomicity violation traces generated from over-approximate methods. The post-prediction analysis method covers all ten possible scenarios. The replaying method reduces context switches to the minimal level to improve scalability. Comparing to other predictive methods on coverage and precision, our post-prediction analysis method PPA improves coverage while ensuring precision, our replaying method further improves coverage and ensures precision as well. Both methods does not rely on the instrumentation of local variables and the analysis of source code. Therefore, our methods are scalable and applicable to large programs. The experiment result shows the scalability of our methods is promising compared to related works.

6.2 Future Work

In our tool McPatom, although the extracted thread model contains all equivalent interleavings that have the same happen-before relationships as the instrumented interleaved trace, there can be other interleaved traces containing different happen-before relationships involving other pairs of threads due to branching structures in a concurrent program. Thus in order to predict all potential atomicity violations, enough instrumented interleaved traces need to be captured during test runs of the program. In a word, existing works on atomicity violation prediction check one path at a time, however, it is desired for predictive analysis to reason on entire families of paths. Additional methods can be developed in the future work, to improve the branch coverage toward the completeness of predicting atomicity violations.

BIBLIOGRAPHY

[1] (2008) Grand Challenges in Computer Research, United Kingdom Computing Research Committee, http://www.ukcrc.org.uk/grand_challenges/index.cfm. [Online]. Available: http://www.ukcrc.org.uk/grand_challenges/index.cfm

[2] J. Woodcock, "First Steps in the Verified Software Grand Challenge," *Computer*, vol. 39, no. 10, pp. 57–64, 2006.

[3] (2007) Verified Software Repository, http://vsr.sourceforge.net. [Online]. Available: http://vsr.sourceforge.net

[4] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, 2007.

[5] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, vol. 34, no. Web Server issue, pp. 729–732, 2006.

[6] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1039–1065, 2006.

[7] S. P. Callahan, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Towards provenance-enabling paraview," in *Provenance and Annotation of Data and Processes*, ser. Lecture Notes in Computer Science, vol. 5272. Springer Berlin / Heidelberg, 2008, pp. 120–127.

[8] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.

[9] K. Poulsen, "Software bug contributed to blackout," URL: http://www.securityfocus.com/news/8016, 2004, [Online; Accessed: 07/16/2011].

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, USA, 2006, pp. 37–48.

[11] S. Lu, S. Park, and Y. Zhou, "Finding Atomicity-Violation bugs through unserializable interleaving testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 844–860, 2011.

[12] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'10)*, Paphos, Cyprus, 2010, pp. 328–342.

[13] J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob, "The certification of the Mondex electronic purse to ITSEC Level E6," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 5–19, 2007.

[14] S. Stepney, D. Cooper, and J. Woodcock, "An Electronic Purse: Specification, Refinement, and Proof," Oxford University Computing Laboratory, Technical monograph PRG-126, Jul. 2000.

[15] R. Zeng, J. Liu, and X. He, "A Formal Specification of Mondex Using SAM," in *SOSE '08: Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*. Jhongli, Taiwan: IEEE Computer Society, 2008, pp. 97–102.

[16] X. He and Y. Deng, "A Framework for Developing and Analyzing Software Architecture Specifications in SAM." *The Computer Journal, vol.45, no.1*, pp. 111–128, 2002.

[17] X. He and T. Murata, *High-Level Petri Nets - Extensions, Analysis, and Applications*, W.-K. Chen, Ed. Elsevier Academic Press, 2005, vol. The Electrical Engineering Handbook.

[18] S. Katz and O. Grumberg, "A Framework for Translating Models and Specifications," in *Proceedings of the Third International Conference on Integrated Formal Methods*. Springer-Verlag, 2002, pp. 145–164. [Online]. Available: http://portal.acm.org/citation.cfm?id=761039

[19] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[20] L. Freitas and J. Woodcock, "Mechanising Mondex with Z/Eves," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 117–139, 2007.

[21] T. Ramananandro, "Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 21–39, 2007. [Online]. Available: http://www.normalesup.org/~ramanana/work/mondex/

[22] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif, "Verification of Mondex electronic purses with KIV: from transactions to a security protocol," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 41–59, 2007.

[23] M. Butler and D. Yadav, "An incremental development of the Mondex system in Event-B," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 61–77, 2007.

[24] C. George and A. E. Haxthausen, "Specification, proof, and model checking of the Mondex electronic purse using RAISE," *Form. Asp. Comput.*, vol. 20, no. 1, pp. 101–116, 2007.

[25] G. Argote-Garcia, P. J. Clarke, X. He, Y. Fu, and L. Shi, "A Formal Approach for Translating a SAM Architecture to PROMELA," in *SEKE*, 2008, pp. 440–447.

[26] G. C. Gannod and S. Gupta, "An Automated Tool for Analyzing Petri Nets Using SPIN," in *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering.* Washington, DC, USA: IEEE Computer Society, 2001, p. 404.

[27] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim, "Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 22, 2007, p. 1767.

[28] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the kepler scientific workflows system," *In Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, vol. 4145, p. 118, 2006.

[29] B. F. Dongen, A. K. A. D. Medeiros, and L. Wen, "Process mining: Overview and outlook of petri net discovery algorithms," in *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems.* Springer-Verlag, 2009, pp. 225–242.

[30] S. Bowers, T. McPhillips, M. Wu, and B. Ludäscher, "Project histories: managing data provenance across collection-oriented scientific workflow runs," in

Proceedings of the 4th international conference on Data integration in the life sciences. Philadelphia, PA, USA: Springer-Verlag, 2007, pp. 122–138.

[31] B. F. V. Dongen, A. K. de Medeiros, H. M. W. Verbeek, A. Weijters, and W. M. P. V. der Aalst, "The ProM framework: A new era in process mining tool support," *Applications and Theory of Petri Nets 2005*, p. 444–454, 2005.

[32] B. F. van Dongen and W. M. P. van der Aalst, "A meta model for process mining data," in *Proceedings of the CAiSE*, vol. 5, 2005, p. 309–320.

[33] H. M. W. Verbeek, J. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "ProM 6: The process mining toolkit," in *Proceedings of the BPM Demonstration Track*, 2010.

[34] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren, "Provenance: a future history," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. Orlando, Florida, USA: ACM, 2009, pp. 957–964.

[35] C. W. Günther and W. M. P. van der Aalst, "Fuzzy mining: adaptive process simplification based on multi-perspective metrics," in *Proceedings of the 5th international conference on Business process management*, Brisbane, Australia, 2007, pp. 328–343.

[36] A. J. M. M. Weijters and A. K. A. D. Medeiros, "Process mining with the HeuristicsMiner algorithm," *Technische Universiteit Eindhoven, Tech. Rep. WP*, vol. 166, 2006.

[37] D. Leake and J. Kendall-Morwick, "Towards Case-Based support for e-Science workflow generation by mining provenance," in *Proceedings of the 9th European conference on Advances in Case-Based Reasoning*. Trier, Germany: Springer-Verlag, 2008, pp. 269–283.

[38] J. Zhao, C. Goble, R. Stevens, and D. Turi, "Mining taverna's semantic web of provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 463–472, 2008.

[39] D. A. Holland, M. I. Seltzer, U. Braun, and K. Muniswamy-Reddy, "PASSing the provenance challenge," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 531–540, 2008.

[40] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva, "Tackling the provenance challenge one layer at a time," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 473–483, 2008.

[41] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson, "Addressing the provenance challenge using ZOOM," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 497–506, 2008.

[42] B. Cao, B. Plale, G. Subramanian, E. Robertson, and Y. Simmhan, "Provenance information model of karma version 3," in *IEEE Congress on Services - I*, 2009, pp. 348–351.

[43] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, and J. Myers, "The open provenance model core specification (v1.1)," *Future Generation Computer Systems*, 2010.

[44] W. M. P. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, no. 9, p. 1128–1142, 2004.

[45] J. Zhang, Q. Liu, and K. Xu, "FlowRecommender: a workflow recommendation technique for process provenance," in *Proceedings of the 8th Australasian Data Mining Conference (AusDM 2009): Data Mining and Analytics 2009*, Melbourne, Australia., 2009.

[46] W. Tan, J. Zhang, and I. Foster, "Network analysis of scientific workflows: A gateway to reuse," *Computer*, vol. 43, no. 9, p. 54–61, 2010.

[47] T. Li, F. Liang, S. Ma, and W. Peng, "An integrated framework on mining logs files for computing system management," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, p. 776–781.

[48] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *the 2005 ACM Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, 2005, pp. 190–200.

[49] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[50] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems.* Addison-wesley New York, 1987, vol. 5.

[51] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, USA, 2009, pp. 325–336.

[52] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, Madison, WI, USA, 1995, pp. 24–36.

[53] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, 2008, pp. 267–280.

[54] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, AZ, USA, 2008, pp. 293–303.

[55] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, pp. 93–110, 2006.

[56] F. Sorrentino, A. Farzan, and P. Madhusudan, "Penelope: weaving threads to expose atomicity violations," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, Santa Fe, NM, USA, 2010, pp. 37–46.

[57] A. Farzan and P. Madhusudan, "The complexity of predicting atomicity violations," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, York, UK, 2009, pp. 155–169.

[58] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 221–230.

[59] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI'03), San Diego, CA, USA, 2003, pp. 338–349.

[60] T. Serbanuta, F. Chen, and G. Rosu, "Maximal causal models for sequentially consistent systems," in *Proceedings of the 3rd International Conference on Runtime Verification (RV'12)*, Istanbul, Turkey, 2012, pp. 136–150.

[61] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation," in *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign (MEMOCODE'11)*, Cambridge, UK, 2011, pp. 99–108.

[62] K. Sen, G. Rosu, and G. Agha, "Detecting errors in multithreaded programs by generalized predictive analysis of executions," in *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, Athens, Greece, 2005, pp. 211–226.

[63] R. Zeng, Z. Sun, S. Liu, and X. He, "McPatom: a predictive analysis tool for atomicity violation using model checking," in *Proceedings of the 19th international conference on Model Checking Software (SPIN'12)*, Oxford, UK, 2012, pp. 191–207.

[64] V. Kahlon and C. Wang, "Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs," in *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, Edinburgh, United Kingdom, 2010, pp. 434–449.

[65] M. K. Ganai, "Scalable and precise symbolic analysis for atomicity violations," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, Lawrence, KS, USA, 2011, pp. 123–132.

[66] J. Yi, C. Sadowski, and C. Flanagan, "SideTrack: generalizing dynamic atomicity analysis," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'09)*, Chicago, IL, USA, 2009, pp. 8:1–8:10.

[67] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*, 2003.

[68] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *the 27th ACM*

SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'12), Tucson, AZ, USA, Oct. 2012.

[69] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, USA, 2002, pp. 211–224.

[70] R. Konuru, H. Srinivasan, and J.-D. Choi, "Deterministic replay of distributed java applications," in *Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, 2000, pp. 219 –227.

[71] X. Liu, W. Lin, A. Pan, and Z. Zhang, "WiDS checker: combating bugs in distributed systems," in *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*, Cambridge, MA, USA, 2007, pp. 19–19.

VITA

RENG ZENG

1998–2003	B.E., Computer Science University of Science and Technology of China Hefei, China
2003–2004	Software Engineer Lenovo Group Beijing, China
2004–2007	Senior Software Engineer Nortel Networks Guangzhou, China
2007–2013	Doctoral Candidate, Computer Science Florida International University Miami, Florida
	Research Assistant and Teaching Assistant Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

R. Zeng, J. Liu, and X. He. *A formal specification of mondex using SAM.* In Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering (SOSE'08), page 97-102, Jhongli, Taiwan, 2008.

R. Zeng, and X. He. *Analyzing a formal specification of mondex using model checking.* In Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC'10), pages 214-229, Natal, Brazil, 2010.

R. Zeng, X. He, and W.M.P van der Aalst. *Mining Processes from Provenance of Scientific Workflows.* In Eighth Latin American Grid (LA Grid) Summit, 2010.

S. Liu, R. Zeng, and X. He. *Pipe+ - a modeling tool for high level petri nets.* In Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE'11), pages 115-121, Miami, FL, USA, 2011.

R. Zeng, Y. Huang, S. Liu, P.J. Clarke, X. He, G.W. van der Linden, and J.L. Ebert. *Sc-xscript: An embedded script language for scientific computation in embedded systems.* In Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE'11), pages 308-314, Miami, FL, USA, 2011.

R. Zeng, X. He, and W.M.P. van der Aalst. *A method to mine workflows from provenance for assisting scientific workflow composition.* In Proceedings of the 7th IEEE 2011 World Congress on Services (SERVICE'11), pages 169-175, Washington DC, USA, 2011.

R. Zeng, X. He, J. Li, Z. Liu, and W.M.P. van der Aalst. *A method to build and analyze scientific workflows from provenance through process mining.* In Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP'11), Heraklion, Crete, Greece, 2011.

S. Liu, R. Zeng, Z. Sun and X. He. *SAMAT - a tool for software architecture modeling and analysis.* In Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE'12), pages 352-358, San Francisco Bay, USA, 2012.

R. Zeng, Z. Sun, S. Liu, and X. He. *McPatom: A predictive analysis tool for atomicity violation using model checking.* In Proceedings of the 19th International SPIN Workshop on Model Checking of Software (SPIN'12), pages 191-207, Oxford, UK, 2012.

R. Zeng, and X. He. *Specifying and Analyzing Mondex in SAM Framework.* Submited to Formal Aspects of Computing.

R. Zeng, Z. Sun, S. Liu, and X. He. *Methods for Improving the Coverage and Precision of Atomicity Violation Predictions.* Submitted to the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13).