Debugging Support for Message-Passing Based Concurrent Software

Mohamed Medhat Elwakil

Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation

Elwakil, Mohamed Medhat, "Debugging Support for Message-Passing Based Concurrent Software" (2011). Dissertations. 401.
https://scholarworks.wmich.edu/dissertations/401
DEBUGGING SUPPORT FOR MESSAGE-PASSING BASED CONCURRENT SOFTWARE

by

Mohamed Medhat Elwakil

A Dissertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Computer Science
Advisor: Zijiang Yang, Ph.D.

Western Michigan University
Kalamazoo, Michigan
June 2011
DEBUGGING SUPPORT FOR MESSAGE-PASSING BASED CONCURRENT SOFTWARE

Mohamed Medhat Elwakil, Ph.D.
Western Michigan University, 2011

Today, multi-core processors are used in all computing aspects, including embedded systems such as mobile devices. In order for software programs to benefit from the transition to multi-core processors, the programs need to be concurrent.

Recent research shows that programs developed using the message-passing model can scale better than programs developed using the shared-memory model. However, the message-passing model is not widely adopted, partly, because of the lack of debugging tools. Debugging message-passing programs is very intricate due to their inherent nondeterministic behavior; even with the same input, a program may behave differently when executed multiple times. Bugs in message-passing programs are elusive; they are hard to detect and are hard to reproduce when they appear.

The primary goal of this research is to alleviate the burden of debugging message-passing based concurrent programs. In particular, we enhance traditional testing by using trace-based symbolic predictive analysis to detect much more errors that couldn’t be detected before, and we propose efficient techniques for deterministically replaying an execution in case of failure.

In testing enhancement, we apply a novel predictive analysis technique in which a program execution model and a set of safety properties are encoded as a quantifier-free...
first-order logic formula, whose satisfiability determines if there exists an execution in which at least one safety property is violated. Also, we have developed DR-MCAPI, the first tool to introduce the deterministic replay capability to message-passing programs for multicore systems. During a recording phase, an unobstructed execution of an input program is monitored to produce a trace. In case of failure, the stored trace is used to enforce an execution that is logically equivalent to the one observed in the recording phase. We have successfully applied our techniques on multicore programs developed using the new MCAPI specification and Web Services described using the BPEL language.
© 2011 Mohamed Medhat Elwakil
Dedicated to my parents and first teachers
Medhat Elwakil and Magda Abdelkhalek
ACKNOWLEDGMENTS

Even though only my name appears on the cover of this dissertation, many people have contributed to it.

First of all, Dr. Zijiang Yang, my advisor and mentor. He introduced me to automated software engineering and SMT-driven verification and this work would not have been possible without his profound insight, helpful guidance and constant encouragement. Dr. Yang was always there when I needed him.

I would also like to thank my committee members: Dr. Li Yang, Dr. Leszek Lilien and Dr. Liqiang Wang for their interest, time, feedback and encouragement.

I’m very grateful to the CS department at WMU for supporting my graduate studies. Special thanks go to Dr. Mohsen Guizani, Dr. Dionysios Kountanis and Dr. Wuwei Shen.

Also, I can’t forget all the great teachers and friends that I met at Cairo University: Dr. Ehab Hassanein, Dr. Mokhtar Boshra, Dr. Ali El-Bastawissy, Dr. Sherif Mazen, Dr. Khaled Shaalan, Dr. Mohamed El-Sharkawi, Dr. Reem Bahgat, Sherif Sakr, Essam Mansour, Rami Hafez, Mohamed Kamal, and Ahmad Nabil.

Many friends have helped me during my stay in Kalamazoo. Their support and care helped me overcome setbacks and stay focused. I value their friendship. I would mention: Mustafa Sanver, Wasim El-Hajj, Lotfi Ben Othmane, Shaheen Abdel Naby, Ibrahim Sheta, Shannon Rauner, and Dr. Sherif Yehia.
Lastly, but most importantly, I want to express my deep gratitude towards my parents who shouldered much emotional and financial pain for this dissertation to become a reality.

Mohamed Medhat Elwakil
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................ii

LIST OF TABLES ..................................................................................................................ix

LIST OF FIGURES ...............................................................................................................x

CHAPTER

1. INTRODUCTION ............................................................................................................. 1

1.1. Motivation ....................................................................................................................... 1

1.1.1. Concurrent programming ......................................................................................... 1

1.1.2. Shared-memory vs. message-passing ....................................................................... 2

1.1.3. Advancements in message-passing .......................................................................... 3

1.1.4. Challenges of debugging concurrent programs ...................................................... 3

1.1.5. The need for debugging support for message-passing programs.......................... 3

1.2. Contributions ............................................................................................................... 4

1.3. Dissertation Organization ........................................................................................... 5

2. PRELIMINARIES ........................................................................................................... 6

2.1. MCAPI concepts ........................................................................................................... 7

2.2. MCAPI APIs ............................................................................................................... 7

2.3. Non-determinism in MCAPI programs ....................................................................... 13
CHAPTER

2.4. Satisfiability Modulo Theories (SMT) .................................................. 16

3. TRACE-BASED SMT-DRIVEN PREDICTIVE ANALYSIS ............................... 18

3.1. The mzPredictor workflow ................................................................. 20

3.2. The mzInstrumenter ............................................................................. 22

3.2.1. The trace grammar ......................................................................... 23

3.3. State-based encoding ........................................................................... 25

3.3.1. The symbolic variables ................................................................. 26

3.3.2. The constraints ................................................................................ 28

3.3.3. Experimental results ....................................................................... 35

3.4. Order-based encoding .......................................................................... 38

3.4.1. The symbolic variables ................................................................. 38

3.4.2. The constraints ................................................................................ 40

3.4.3. Experimental results ....................................................................... 45

3.5. Reporting a violating scenario .............................................................. 46

3.6. Conclusion ............................................................................................ 48

4. DETERMINISTIC REPLAY FOR MCAPI .................................................. 50

4.1. Introduction .......................................................................................... 50
Table of Contents-Continued

CHAPTER

4.2. DR-MCAPI: Deterministic Replay for MCAPI Programs .......................... 52
   4.2.1. DR-MCAPI workflow .................................................................. 52
   4.2.2. Source instrumentation .............................................................. 53
   4.2.3. Operating modes ...................................................................... 54

4.3. DR-MCAPI data-replay ...................................................................... 56
   4.3.1. The trace structure .................................................................. 56
   4.3.2. The replay mechanism .............................................................. 60

4.4. DR-MCAPI order-replay .................................................................... 64
   4.4.1. Sender-based order-replay ........................................................ 64
   4.4.2. Receiver-based order-replay ...................................................... 72

4.5. Results ............................................................................................. 83
   4.5.1. Methodology ............................................................................ 83
   4.5.2. Log size .................................................................................. 85
   4.5.3. Runtime overhead .................................................................. 86
   4.5.4. Memory usage overhead .......................................................... 89

4.6. Features ............................................................................................ 92
   4.6.1. Usability .................................................................................. 92
   4.6.2. Portability ............................................................................... 93
CHAPTER

6.2. Deterministic replay for MCAPI ....................................................... 120
6.3. Message race detection for Web services ........................................... 122

7. CONCLUSION AND FUTURE WORK ..................................................... 125
7.1. Conclusion ......................................................................................... 125
7.2. Future research directions ................................................................. 126

8. REFERENCES .......................................................................................... 127
LIST OF TABLES

3.1. Trace example ................................................................. 25
3.2. Symbolic variables of the trace in 3.1 ................................ 40
3.3. The solution of the $F_R$ formula ...................................... 47
3.4. mzReporter report ......................................................... 48
4.1. Data-replay trace of the program in Figure 4.5 ..................... 59
4.2. Sender-based order-replay trace of the program in Figure 4.5 .... 66
4.3. Sender-based order-replay example .................................... 70
4.4. Handling dr_msg_send example ....................................... 71
4.5. Receive-based order-replay trace ..................................... 75
4.6. Records structures and sizes in D-replay, S-replay and R-replay .. 86
4.7. Runtime and memory usage overheads of the replay techniques .... 93
4.8. An order of arrival in a recorded execution .......................... 95
4.9. Equivalent but not identical order of arrival ....................... 96
4.10. D-replay vs. S-replay vs. R-replay ................................... 96
5.1. Partial valuation of the stock-trading FOL formula .................. 113
5.2. Partial valuation of the loan-approval FOL formula ................ 115
5.3. Yices statistics .............................................................. 116
LIST OF FIGURES

2.1. A MCAPI program snippet ................................................................. 9
2.2. Pseudocode of a MCAPI program with message races ...................... 14
2.3. Pseudocode of a MCAPI program that uses mcapi_wait_any ............... 15
2.4. Pseudocode of a MCAPI program that uses mcapi_test ...................... 16
3.1. The mzPredictor workflow ................................................................... 20
3.2. Results of the first iteration of experiments on state-based encoding .... 36
3.3. Results of applying state-based encoding on the BT benchmark .......... 38
3.4. The Construct_FOrder() algorithm ...................................................... 41
3.5. Results of applying order-based encoding on the BT benchmark .......... 45
3.6. Results of applying order-based encoding on the CG benchmark ......... 46
3.7. Results of applying order-based encoding on the TN benchmark .......... 46
4.1. The two phases of deterministic replay ................................................ 52
4.2. DR-MCAPI workflow ........................................................................... 53
4.3. An instrumented MCAPI program snippet ......................................... 53
4.4. DR-MCAPI sits between the MCAPI library and the instrumented program 54
4.5. Pseudocode of the program in 2.1 ........................................................ 55
4.6. Handling dr_msg_recv calls ............................................................... 61
4.7. Handling dr_wait calls ....................................................................... 62
4.8. Handling dr_wait_any calls ................................................................. 62
List of Figures-Continued

4.9. Handling dr_test calls ........................................................................................................ 64
4.10. The three steps of sending a message .............................................................................. 68
4.11. The SendCallsScheduler algorithm .................................................................................. 69
4.12. Handling dr_wait_any calls ............................................................................................. 69
4.13. Handling dr_test calls ........................................................................................................ 70
4.14. Handling dr_msg_recv calls ............................................................................................ 77
4.15. Handling dr_wait calls ........................................................................................................ 79
4.16. Handling dr_wait_any calls ............................................................................................. 80
4.17. Handling dr_test calls ........................................................................................................ 82
4.18. Comparing the trace size among replay techniques ........................................................... 85
4.19. The runtime overhead in D-replay .................................................................................... 87
4.20. The runtime overhead in S-replay ..................................................................................... 88
4.21. The runtime overhead in R-replay .................................................................................... 89
4.22. The memory usage overhead in D-replay ......................................................................... 90
4.23. The memory usage overhead in S-replay ......................................................................... 91
4.24. The memory usage overhead in R-replay ......................................................................... 92
4.25. Equivalent replay vs. identical replay ............................................................................... 95
5.1. Messages can arrive at different orderings ......................................................................... 100
5.2. Steps for finding messages races in a Web service ............................................................ 101

xi
List of Figures-Continued

5.3. The stock-trading Web service ................................................................. 112

5.4. The loan-approval Web service ................................................................. 114
CHAPTER 1
INTRODUCTION

1.1. Motivation

In the last three decades, new processors have attained huge performance gains due to improvements in three directions: transistor density, cache capacity and, primarily, raw clock speed [1]. Enhancements in any of these directions delivered predictable and significant increases in the performance of sequential programs. Recently, single-core processors started reaching the physical limits of complexity and speed. Instead of driving raw clock speeds higher, processor manufacturers had instead turned to the multi-core architecture. A multi-core processor is an integrated circuit to which two or more CPUs are attached for improved performance and reduced energy consumption. Today, multi-core processors are used in all computing aspects, including embedded systems such as mobile devices [2].

1.1.1. Concurrent programming

In order for software programs to benefit from the transition to multi-core processors, the programs need to be concurrent. A concurrent program consists of a collection of cooperating computation components that are executed in parallel [3]. There are two major concurrent programming models that differ in how the computation components communicate with each other. In the shared-memory model, components communicate by altering the contents of shared-memory locations. In the message-passing model, components communicate by exchanging messages.
1.1.2. Shared-memory vs. message-passing

Among practitioners and researchers alike, shared-memory is considered the leading concurrency model with message-passing being known as "the other model for concurrent programming" [4]. This is due to the conception that message-passing programs are always slower than their shared-memory counterparts because of the communication time overhead. Hence, using the message-passing model was restricted to situations where there is no shared memory available such as in Beowulf clusters. This conception ignores the time overhead associated with synchronization mechanisms commonly used in shared-memory programs such as locks [5] and barriers [6].

A study [7] that compares the performance of two implementations (a shared-memory implementation and message-passing implementation) of the same algorithm reveals that the message-passing implementation performs up to 52% better than the shared-memory one when four and more processors are used. This study concludes that given fast-enough communication channels, message-passing programs can rival corresponding shared-memory programs. Also, the authors of [8] coin the term "coherency wall" while arguing that the message-passing model is more suitable than the shared-memory model for the multi-core era processors. The coherence wall refers to the overhead imposed by cache coherence protocols needed to maintain the view of memory coherent across the cores. This overhead grows with the number of cores. Using message-passing among cores eliminates the cache coherence protocols overhead, altogether.
1.1.3. Advancements in message-passing

In 2008, the Multicore Association [9], a consortium of major corporations and leading research centers, published the first version of the Multicore Communications API (MCAPI) [10]. MCAPI is a new message-passing API that is intended for systems with multiple cores on a chip and/or chips on a board. MCAPI leverages efficient on-chip interconnects in multi-core processors to minimize communication latency. Another emerging message-passing work is the Intel RCCE library [11]. RCCE offers a message-passing API that is designed for the Single-Chip Cloud Computer (SCC) processor [12]. MCAPI and RCCE are set to encourage programmers to consider message-passing as a viable alternative to shared-memory.

1.1.4. Challenges of debugging concurrent programs

Probably the greatest barrier of developing concurrent programs is that they are very hard to debug. Debugging is the process of analyzing and possibly changing a given program that doesn’t meet its specifications in order to produce a new program that is close to the original program and does satisfy the specifications [13]. Debugging concurrent programs is very intricate due to their inherent nondeterministic behavior; even with the same input, a concurrent program may behave differently when executed multiple times. The bugs of concurrent programs are elusive; they are hard to detect and are hard to reproduce when they appear.

1.1.5. The need for debugging support for message-passing programs

Reducing the performance penalty associated with using the message-passing model is necessary but not sufficient for attracting programmers. Message-passing programs are
still concurrent programs with inherent non-determinism and debugging them is as challenging as debugging shared-memory programs. The long industrial usage and academic interest of the shared-memory model produced a plethora of techniques and tools that help programmers address the specific challenges of debugging shared-memory programs. Relatively, little effort was put into addressing the issues related to debugging message-passing programs [4].

We believe that by developing techniques and tools that support message-passing debugging, programmers will be willing to adopt the message-passing model. Such tools should be easy to use and scale well.

1.2. Contributions

The primary goal of this dissertation is to develop sound techniques and practical tools that ease the burden of debugging message-passing based concurrent programs. The main contributions of this dissertation include the following:

1. To aid detecting bugs, we propose a trace-based SMT-driven predictive analysis technique tailored for enhancing the testing process of MCAPI programs [14], [15], and [16].

2. We have developed a proof-of-concept tool (mzPredictor) that demonstrates the practicality of the trace-based SMT-driven predictive analysis technique.

3. To help finding the cause of a program failure, we propose methods for deterministically replaying a MCAPI program [17] and [18].
4. The DR-MCAPI tool that implements our methods for deterministically replaying a MCAPI program

5. We propose an approach for improving the reliability of Web services by detecting message races using SMT-based analysis [19].

1.3. Dissertation Organization

This dissertation is organized as follows. Chapter 2 gives a background on MCAPI and SMT. Chapter 3 describes the techniques employed by the mzPredictor. Chapter 4 describes DR-MCAPI. Chapter 5 describes our work on detecting message races in Web services applications. Chapter 6 discusses related work. Chapter 7 concludes the dissertation and indicates future directions for this research.
CHAPTER 2

PRELIMINARIES

Processors incorporating multiple cores on a chip or multiple chips on a board are becoming progressively common. The Multicore Association (MCA) [9] has developed the Multicore Communications API (MCAPI) specification to address inter-core communication needs in multi-core programs. A key objective of MCAPI is providing source code compatibility across multiple operating systems, while consuming a minimal footprint. MCAPI is different from the Message Passing Interface (MPI) [20]; a widely used standard for managing coarse-grained concurrency on distributed computers. While MPI is intended for inter-computer communication and needs to be installed on top of an operating system, MCAPI is intended for inter-core communication and can be installed on top of an operating system or an extremely thin run-time environment such as a hypervisor. A major design goal of MCAPI is to function as a low-latency interface benefiting from efficient on-chip interconnects in a multi-core chip. Thus, MCAPI is a light weight API that delivers high performance and needs a tiny memory footprint that is significantly lower than that of MPI [10]. Currently, there are two implementations of the MCAPI specification; the standard implementation provided by the MCA and a relatively newer implementation provided by Mentor Graphics [21]. Our work is applicable to both implementations.
2.1. MCAPI concepts

A typical MCAPI program consists of a fixed set of MCAPI nodes running in parallel and communicating via messages. The MCAPI specification states that a node is a logical abstraction that can be a process, a thread, or a processor core. A node can be thought of as a stream of code execution. In the MCA implementation, a node is mapped to a thread. A node is identified by a node identifier that ranges from 1 ... n for n nodes. Communication between nodes occurs through endpoints. An endpoint is a destination to which messages may be sent; much similar to a socket in networking. A node may have one or more endpoints and an endpoint is uniquely defined by a pair of a node identifier and a port number. A message is a chunk of data sent from one endpoint to another. No handshaking protocols are needed to send a message.

2.2. MCAPI APIs

The MCAPI specification supplies APIs for initializing and tearing down nodes, creating and deleting endpoints, obtaining addresses of remote endpoints, and sending and receiving messages. Figure 2.1 shows a snippet of a MCAPI program with 5 nodes. Both of nodes 1 and 3 send three messages nodes 2, 4 and 5. Node 2 uses the data in the two received messages to calculate a new value and sends it to nodes 4 and 5. Hence, nodes 4 and 5 are expecting to receive 3 messages, each.
#include <stdio.h>
#include <mcapi.h>

void N0DE1 () {
    mcapi_status_t Status;
    mcapi_version_t Version;
    int X=1;
    mcapi_initialize(1,&Version,&Status);
    LocalEP = mcapi_create_endpoint (1,&Status);
    N2EP = mcapi_get_endpoint (2,1,&Status);
    N4EP = mcapi_get_endpoint (4,1,&Status);
    N5EP = mcapi_get_endpoint (5,1,&Status);
    mcapi_msg_send(LocalEP, N2EP, &X, sizeof (X), 1, &Status);
    mcapi_msg_send(LocalEP, N4EP, &X, sizeof (X), 1, &Status);
    mcapi_msg_send(LocalEP, N5EP, &X, sizeof (X), 1, &Status);
    mcapi_delete_endpoint(LocalEP,&Status);
}

void N0DE2 () {
    mcapi_status_t Status;
    mcapi_version_t Version;
    mcapi_endpoint_t LocalEP,N4EP,N5EP;
    int Request_t* Requests[2];
    int A,B,C,
    int Req;
    mcapi_initialize(2,2,Version,&Status);
    LocalEP = mcapi_create_endpoint (1,&Status);
    N4EP = mcapi_get_endpoint (4,1,&Status);
    N5EP = mcapi_get_endpoint (5,1,&Status);
    mcapi_msg_recv_i(LocalEP, &A, sizeof (A),Requests[0],&Status);
    mcapi_msg_recv_i(LocalEP, &B, sizeof (B),Requests[1],&Status);
    Req=mcapi_wait_any (2,Requests,&RecvSize,&Status,MCAPI_INFINITE);
    if (Req==0)
        mcapi_wait(Requests[1], &RecvSize,&Status,MCAPI_INFINITE);
    else
        mcapi_wait(Requests[0], &RecvSize,&Status,MCAPI_INFINITE);
        C=Func(A,B);
        mcapi_msg_send(LocalEP, N4EP, &C, sizeof (C), 1, &Status);
        mcapi_msg_send(LocalEP, N5EP, &C, sizeof (C), 1, &Status);
        mcapi_delete_endpoint(LocalEP,&Status);
}

void N0DE3 () {
    mcapi_status_t Status;
    mcapi_version_t Version;
    int Y=1;

}
Figure 2.1. A MCAPI program snippet
Before invoking any other MCAPI functions, the `mcapi_initialize` (e.g. line 10) function must be called to initialize the MCAPI environment on a given node. `mcapi_initialize` has one input argument which is the node identifier. A node finishes execution by calling `mcapi_finalize` (e.g. line 19). An endpoint is created by invoking `mcapi_create_endpoint` (e.g. line 11). `mcapi_create_endpoint` has one input argument which is the endpoint port number. The function `mcapi_get_endpoint` retrieves a handle to a remote endpoint given its node identifier and port number (e.g. line 12).

The functions used for sending and receiving messages are: `mcapi_msg_send`, `mcapi_msg_send_i`, `mcapi_msg_recv`, and `mcapi_msg_recv_i`. `mcapi_msg_send` has the following prototype:

```c
void mcapi_msg_send (mcapi_endpoint_t send_endpoint, mcapi_endpoint_t receive_endpoint, void* buffer, size_t buffer_size, mcapi_priority_t priority, mcapi_status_t* status);
```

`mcapi_msg_send` sends a message from a source endpoint (`send_endpoint`) to a destination endpoint (`receive_endpoint`). `buffer` is the starting address of the data buffer whose contents need to be transferred to the destination side. `buffer_size` is the size of the buffer in bytes, `priority` is the priority of this message, and `status` returns an error code.

`mcapi_msg_send` is **blocking**, since it doesn’t return till the contents of the program buffer (`buffer`) have been copied to the MCAPI runtime buffers. For example, the `mcapi_msg_send` call in line 15 sends the contents of `x` from endpoint `LocalEP` to endpoint `N2EP`. 
mcapi_msg_send_i is the non-blocking version of mcapi_msg_send as it returns immediately and before the contents of the program buffer have been copied to the MCAPI runtime buffers. mcapi_msg_send_i has the following prototype:

```c
void mcapi_msg_send_i (mcapi_endpoint_t send_endpoint, mcapi_endpoint_t receive_endpoint, void* buffer, size_t buffer_size, mcapi_priority_t priority, mcapi_request_t* request, mcapi_status_t* status);
```

When a mcapi_msg_send_i function is called, it initializes the request argument and the state of the mcapi_msg_send_i call is said to be *Pending*. When the data in the program buffer (buffer) has been completely copied to the MCAPI runtime buffers, the state of the mcapi_msg_send_i call becomes *Complete*. The program buffer shouldn’t be accessed by the program, till the state of the mcapi_msg_send_i call becomes *Complete*.

mcapi_msg_recv is a blocking receive with the following prototype:

```c
void mcapi_msg_recv (mcapi_endpoint_t receive_endpoint, void* buffer, size_t buffer_size, size_t* received_size, mcapi_status_t* mcapi_status);
```

mcapi_msg_recv retrieves a message from the MCAPI runtime buffers at a destination endpoint (receive_endpoint). buffer is the starting address of the memory allocated by the program where the data will be received, buffer_size is the size of the buffer in bytes, received_size is the size of the received data in bytes, and status returns an error code. For example, the mcapi_msg_recv call in line 94 retrieves a message from the MCAPI runtime buffers of endpoint LocalEP and places the message payload in G.
mcapi_msg_recv_i is the non-blocking version of mcapi_msg_recv as it returns immediately and before a message has been retrieved from the MCAPI runtime buffers.

mcapi_msg_send_i has the following prototype:

```c
void mcapi_msg_recv_i (mcapi_endpoint_t receive_endpoint, void* buffer,
size_t buffer_size, mcapi_request_t* request, mcapi_status_t* mcapi_status );
```

The MCAPI specifications provide three routines for tracking the completion state of non-blocking calls: `mcapi_test`, `mcapi_wait`, and `mcapi_wait_any`.

`mcapi_test` is used to determine whether a non-blocking call has completed or not. `mcapi_test` itself is non-blocking and returns immediately. `mcapi_test` has the following prototype:

```c
mcapi_boolean_t mcapi_test (mcapi_request_t* request, size_t* size,
mcapi_status_t* mcapi_status);
```

A `mcapi_test` call succeeds (i.e. returns `true`), if the non-blocking call that initialized the request variable (`request`) has completed and fails (i.e. returns `false`) otherwise. If a `mcapi_test` call succeeds, then the `size` argument is set to the number of bytes that was either sent or received by the non-blocking call. For example, the loop in line 79 will not terminate till the `mcapi_msg_recv_i` call in line 77 completes and a message is retrieved from the runtime buffers.

`mcapi_wait` is a blocking function and is used to determine the completion of a non-blocking call. It has the following prototype:
mcapi_boolean_t mcapi_wait (mcapi_request_t* request, size_t* size,
mcapi_status_t* mcapi_status, mcapi_timeout_t timeout);

A mcapi_wait call returns when the non-blocking call that initialized the request variable (request) has completed. The size argument is set to the number of bytes that was either sent or received by the non-blocking call. For example, the mcapi_wait call in line 80 will not return till the mcapi_msg_recv_i call in line 78 completes (i.e. a message has been retrieved from the MCAPI runtime buffers and copied to the variable d).

mcapi_wait_any is a blocking function and is used to determine the completion of one of a set of non-blocking calls. It has the following prototype:

mcapi_int_t mcapi_wait_any (size_t number, mcapi_request_t** requests, size_t* size, mcapi_status_t* mcapi_status, mcapi_timeout_t timeout);

where requests is an array of request variables and number is the size of this array. mcapi_wait_any returns the index of the request variable whose non-blocking call has completed. The size argument is set to the number of bytes there were either sent or received by the non-blocking call. For example, the mcapi_wait_any call in line 36 will return when either one of the mcapi_msg_recv_i calls in lines 34 and 35 completes.

2.3. Non-determinism in MCAPI programs

There are two rules that govern the order of messages arrivals at a destination endpoint: 1) messages sent from the same source endpoint to the same destination endpoint are guaranteed to arrive at their destination according to their transmission order and 2) messages sent from different source endpoints will arrive at their destination in any order,
even if these source endpoints belong to the same node. The second rule combined with the fact that \texttt{mcapi\_msg\_recv} and \texttt{mcapi\_msg\_recv\_i} calls don't specify the source endpoint, make it possible for \textit{message races} to take place. Two or more messages are said to be \textit{racing} if their order of arrival at a destination (i.e. an endpoint) is non-deterministic [22]. MCAPI receive calls are called \textit{promiscuous} receives as they permit receiving messages from any source endpoint.

Figure 2.2 shows the pseudocode of a MCAPI program that suffers from message races. In that program, a node creates a single endpoint and sends messages to all other nodes (lines 3 to 7) and is expecting to receive a message from all other nodes (lines 8 to 9). Assuming there are N nodes, any node should receive N-1 messages that are racing with each other. The orders of messages arrival can change across different executions of the program (i.e. the final values in the \texttt{Buffer} array will be different with different executions); leading to the irreproducibility effect.

```
N=NodesCount();
mcai\_init\_node(ThisNode);
LocalEP=mcapi\_create\_ep(ThisNode,1);
for (Index=0;Index<N;Index++){
  if (Index==ThisNode) continue;
  RemoteEP=mcapi\_get\_ep(Index,1);
  mcapi\_msg\_send(LocalEP,RemoteEP,&Data[Index]);
}
for (Index=0;Index<N-1; Index++)
  mcapi\_msg\_recv(LocalEP,&Buffer[Index]);
mcapi\_delete\_ep(LocalEP);
mcapi\_finalize\_node(ThisNode);
```

\textbf{Figure 2.2. Pseudocode of a MCAPI program with message races}

Another source of non-determinism in MCAPI programs is the \texttt{mcapi\_wait\_any} call. The node in Figure 2.3 has two endpoints and is expecting to receive a message at each
endpoint (lines 4 and 5). mcapi_wait_any blocks execution till one of the two messages is received. Although the two messages are not racing with each other as they have different destination endpoints, mcapi_wait_any will let the order of arrival of these messages affect the proceeding parts of the program in a non-deterministic manner. Specifically, the value of ReqIndex may be different across different executions of the program resulting in different branches of the switch statement being selected in consecutive executions of the program.

```c
1 mcapi_initialize(ThisNode);
2 LocalEPl=mcapi_create_ep(ThisNode,1);
3 LocalEP2=mcapi_create_ep(ThisNode,2);
4 mcapi_msg_recv_i(LocalEPl,&Buffer1,Requests[0]);
5 mcapi_msg_recv_i(LocalEP2,&Buffer2,Requests[1]);
6 ReqIndex=mcapi_wait_any(Requests);
7 switch (ReqIndex) {
 8   case 0: ...
9   case 1: ...
10 }
11 mcapi_delete_ep(LocalEPl);
12 mcapi_delete_ep(LocalEP2);
13 mcapi_finalize_node(ThisNode);
```

**Figure 2.3. Pseudocode of a MCAPI program that uses mcapi_wait_any**

The non-blocking mcapi_test call can be used to maximize the overlap between computation and communication. Since mcapi_test calls are non-blocking, they can be used in polling loops. In Figure 2.4, a node is expecting to receive a message at a local endpoint. The function mcapi_test is used to determine whether the expected message has arrived. The number of times mcapi_test will return false, and consequently the value of variable A at line 6, is dependent on uncontrollable factors such as the current core workload and the inter-core communication latency. Although not many programs will base their actions on the number of failed tests, some could do so to implement a kind of time-out.
In summary, MCAPI programs have three intrinsic sources of non-determinism 1) `mcapi_msg_recv` and `mcapi_msg_recv_i` calls, 2) `mcapi_wait_any` calls and 3) `mcapi_test` calls. Such inherent non-deterministic behavior doesn’t permit repeated execution as a mean of debugging MCAPI programs. Hence, introducing the ability to replay an observed MCAPI program execution can significantly help MCAPI programs developers.

2.4. Satisfiability Modulo Theories (SMT)

The *Boolean satisfiability* problem (SAT) is the problem of deciding whether a given propositional logic formula has a solution. A solution of a SAT problem is a truth assignment to the formula boolean variables such that the formula evaluates to true. For example, a solution for the formula: \((p \lor q) \land (p \lor r)\) could be the truth assignment: \(p = True, q = False, r = False\).

A *SAT Solver* is a program that automatically decides whether an input formula has a solution, and if so, outputs the truth assignment of this solution. Available SAT solvers include Chaff [23], MiniSAT [24] and GRASP [25]. SAT solvers have many practical applications where a problem is expressed as a propositional logic formula and then decided using a SAT solver. Such applications include model-checking of finite-state
systems [26], AI planning [27], checking of pedigree consistency [26], and logic synthesis [28].

Some problems are better described using more expressive logics such as first-order logic in which logical connectives, quantifiers, function and predicate symbols are used. The solution of a first-order formula consists of an interpretation for the variable, function and predicate symbols [29]. Deciding the general first-order logic satisfiability is very time consuming and impractical as many applications require satisfiability with respect to some background theories which fixes the interpretations of specific predicate and function symbols [30]. The research field Satisfiability Modulo Theories (SMT) focuses on the satisfiability of formulas with respect to some background theories.

An SMT solver is a program that decides whether a first-order logic formula has a solution with respect to a combination of certain background theories. Existing SMT solvers include Yices [31], Z3 [32] and CVC3 [33]. Yices supports many background theories such as uninterpreted functions, linear arithmetic, scalar types, recursive data-types, tuples, records, extensional arrays, fixed-size bit vectors, \(\lambda\)-expressions, and quantifiers [34].
CHAPTER 3

TRACE-BASED SMT-DRIVEN PREDICTIVE ANALYSIS

In this chapter we describe our trace-based SMT-driven predictive analysis technique tailored for enhancing the testing process of programs developed using the MCAPI specification.

Given a runtime model of a program and a set of safety properties, predictive analysis is used to determine whether there exists a feasible interleaving of the events in the runtime model that violates any of the functional correctness properties of the program. Predictive analysis combines features from testing and model-checking. Similar to testing, the input program is run and its behavior is observed. The execution trace of a program could be viewed as an abstract model of the input program that is analyzed exhaustively to detect potential errors, akin to model-checking [35]. Predictive analysis is not as comprehensive as model-checking, but it is efficient and scalable. Predictive analysis has been used successfully to detect concurrency errors in multi-threaded programs [36], [37], and [38].

Applying predictive analysis involves four steps: 1) obtain a runtime model of an input program, 2) enumerate all feasible execution interleavings that can be derived from the runtime model, 3) check every interleaving to determine if it violates any functional correctness property and 4) inspect violating interleavings to verify that they can occur in
the input program. Explicitly enumerating all feasible execution scenarios is a bottleneck that hinders the performance of predictive analysis.

We have developed a trace-based SMT-driven predictive analysis technique that reduces the process of explicitly enumeration and checking all interleavings to constraint solving that employs off-the-shelf SMT solvers. First, we instrument a program such that it produces a trace when executed. Given the program trace, we construct a quantifier-free first-order logic (QF-FOL) formula that captures not only the given trace, but also all possible interleavings of the events of this trace and the functional correctness properties of the program. The formula is input to an SMT solver; if the formula is satisfiable, then there exists an execution scenario (i.e. a valid permutation of the events in the trace) that violates at least one correctness property. In this case, the SMT solver produces an abstract counter example that embodies the erroneous execution. Otherwise, we can conclude that for all possible execution scenarios involving the same events in the input trace, no violation of the correctness properties is possible.

The set of rules that govern mapping the constructs of a trace to the QF-FOL formula are called an encoding. We have devised two different encodings for MCAPI programs traces: state-based encoding and events-based encoding. Devising the rules that map messaging and other programming constructs (i.e. the trace constructs) to SMT constructs is a daunting research venture because of two reasons: 1) there is a huge semantic gap between the trace constructs and the SMT constructs. Even with such a gap, the logic formula should accurately mimic the trace constructs. Otherwise, the formula will be
incorrect as it will allow execution scenarios that can never happen in reality. 2) at the same time, replicating the tiny details of the trace constructs will produce an inefficient formula that doesn’t scale well with the size of the input trace. Thus, the mapping rules must strike a balance between accuracy and efficiency.

Also, we have developed the mzPredictor tool that implements our trace-based SMT-driven predictive analysis technique. mzPredictor is a push-button solution that takes as input a MCAPI program source code and produces as output a report that describes a specific execution scenario that violates the functional correctness of the input program. In the next section, we describe the workflow of the mzPredictor and its components.

3.1. The mzPredictor workflow

The mzPredictor is a push-button solution that takes as input a MCAPI program source code and produces as output a report that describes a specific execution scenario that violates the functional correctness of the input program. Figure 3.1 shows the mzPredictor workflow and highlights its three components: mzInstrumenter, mzEncoder and mzReporter.

Figure 3.1. The mzPredictor workflow
The mzInstruments instruments a program source code by adding extra code that monitors the program execution and emits events during runtime. An event indicates the execution of a particular program statement that is being monitored. The amalgamation of captured events constitutes a trace of the input program. The resulting trace is accumulated in memory and dumped to a file prior to the termination of the instrumented program. The monitoring is light-weight to minimize probe-effect, but broad enough to allow generating an accurate symbolic representation of the trace. Also, the trace maintains the path condition of each event. An event path condition is the conjunction of all conditions necessary for executing the corresponding statement. Section 3.2 details the instrumentation process and describes the trace grammar.

The mzEncoder translates the captured trace to a QF-FOL formula that consists of symbolic variables and constraints over the values of the symbolic variables. The translation is performed according to a set of rules (i.e. an encoding) that maps the trace constructs to SMT constructs that are restricted to the background theories supported by the target SMT solver. We have devised two encodings. We call the first one the state-based encoding. Preliminary experiments with the state-based encoding was encouraging, however, more thorough experiments revealed that it doesn’t scale well when the number of potential races increases. The state-based encoding uses arrays to mimic MCAPI communication channels. Solving formulas with arrays is very time-consuming using current SMT solvers. Hence, we developed another encoding that doesn’t use arrays and we call it the order-based encoding. The state-based encoding is described in section 3.3 and the order-based encoding described in section 3.4.
We use Yices [31] to solve the formula generated by the mzEncoder. If the formula is solvable (i.e. satisfiable), then Yices outputs a solution that assigns values to the symbolic variables.

The solution produced by Yices embodies an execution scenario that violates the program functional correctness. Finally, the mzReporter (described in section 3.5) transforms this solution into a report that describes a concrete execution scenario showing how functional correctness of the program is violated.

3.2. The mzInstrumenter

The mzInstrumenter uses Eclipse CDT [39] to instrument an input program by inserting logging functions calls at certain locations in the code. These functions communicate with a monitoring engine (the mzLib library in Figure 3.1). The mzLib library maintains the in-memory trace and saves it to the disk when the program execution ends.

The mzInstrumenter performs the following tasks: 1) add auxiliary code before node starting/exiting points to initialize/tear-down the monitoring engine, 2) add function calls that register program variables, including for loops iteration variables 3) since SMT solvers can’t handle prefix/postfix operators, prefix and postfix expressions are replaced with equivalent expressions that do not contain prefix or postfix operators, 4) add code that logs assignment statements including for loops update statements, 5) replace an assert statement with code that logs the assert invariant, 6) replace MCAPI routines calls
with other calls that generate corresponding events and invoke the original MCAPI calls, 7) extend the parameters list of each added logging function to include the path condition and the line number of the corresponding program statement.

The path condition of a program statement is the conjunction of all conditions in the program path leading to the statement. For example, if the statement is in the else-part of an if statement, which in turn is inside a while loop, then the path condition will be the conjunction of the negation of the if statement condition and the while loop condition.

After the instrumentation is complete, the instrumented program is compiled and run. When the instrumented program terminates, it dumps a file that contains an execution trace (i.e. the logged events).

3.2.1. The trace grammar

The execution trace of an MCAPI program running on $N$ nodes will have $N$ sub-traces; a sub-trace for each node. Let $R = \{ T_1, \ldots, T_N \}$ be the trace of an MCAPI program with $N$ nodes. $T_n$ is the sub-trace produced by node $N_n$ and it consists of a sequence of events $T_n = T_{n,1} \ldots T_{n,|T_n|}$. Each sub-trace $T_n$ is associated with a set of local variables: $L_n = \{L_{n,1}, \ldots, L_{n,|L_n|}\}$, and a set of endpoints used in this node: $E\mathcal{P}_n = \{EP_{n,1}, \ldots, EP_{n,|E\mathcal{P}_n|}\}$.

An event $T_{n,x} \in T_n$ is a tuple $< y, Guard, Action >$, such that $n$ is a node identifier, $x$ is the order of $T_{n,x}$ appearance in $T_n$, $y$ is the order of proceeding event in $T_n$, $Guard$ is a condition that must be true for this event to take place, and $Action$ is an atomic
computation that corresponds to an executed statement in the program. If $T_{n,x}$ is the last event in $T_n$, then $y$ is set to the special marker value $\bot$. Guard is the conjunction of all conditions in the program path leading to the statement that produced the event. For example, if the statement that produces $T_{n,x}$ is in the then-part of an if statement, which in turn is inside a while loop, then Guard will be the conjunction of the if statement condition and the while loop condition. Action can be any of the following types:

- **Assign**($v$, $exp$) corresponds to an assignment statement that assigns $exp$ to $v$. $v \in L_n$ is a variable and $exp$ is an expression over $L_n$.

- **Send**($src$, $dest$, $exp$) corresponds to a `mcapi_msg_send` statement that sends a message from $src$ to $dest$, which contains $exp$. $src \in E_P_n$ and $dest \in E_P_m$ are the source and destination endpoints. $exp$ is an expression over $L_n$.

- Similarly, **Send_i**($src$, $dest$, $exp$, $req$) corresponds to a `mcapi_msg_send_i` statement where $req \in L_n$ is a request variable.

- **Recv**($recv$, $v$) corresponds to a `mcapi_msg_recv` statement that receives a message at the receiving endpoint $recv \in E_P_n$. The message contents are assigned to variable $v \in L_n$.

- Similarly, **Recv_i**($recv$, $v$, $req$) corresponds to a `mcapi_msg_recv_i` statement where $req \in L_n$ is a request variable.

- **Wait**($req$) corresponds to a `mcapi_wait` statement that waits for the completion of a non-blocking action whose status is tracked with request variable $req \in L_n$.

- **Assert**($exp$) corresponds to an assert statement with the expression $exp$. $exp$ is a boolean expression over $L_n$. 

24
Table 3.1 shows the events of a trace of an MCAPI program with four nodes. Nodes 1 and 3 send two numbers to nodes 2 and 4, each. Node 2 calculates the difference between the two received numbers and sends that difference to node 4. Node 4 is expecting to receive three numbers, with the first number being greater than zero. We will be using this trace as an ongoing example.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{1,1}$</td>
<td>Assign(Msg, 1)</td>
</tr>
<tr>
<td>$T_{1,2}$</td>
<td>Send_i(Ep1, Ep2, Msg, r0)</td>
</tr>
<tr>
<td>$T_{1,3}$</td>
<td>Send_i(Ep1, Ep4, Msg, r1)</td>
</tr>
<tr>
<td>$T_{1,4}$</td>
<td>Wait(r0)</td>
</tr>
<tr>
<td>$T_{1,5}$</td>
<td>Wait(r1)</td>
</tr>
<tr>
<td>$T_{2,1}$</td>
<td>Assign(X, 0)</td>
</tr>
<tr>
<td>$T_{2,2}$</td>
<td>Assign(Y, 0)</td>
</tr>
<tr>
<td>$T_{2,3}$</td>
<td>Assign(Z, 0)</td>
</tr>
<tr>
<td>$T_{2,4}$</td>
<td>Recv_i(Ep2, X, r2)</td>
</tr>
<tr>
<td>$T_{2,5}$</td>
<td>Recv_i(Ep2, Y, r3)</td>
</tr>
<tr>
<td>$T_{2,6}$</td>
<td>Wait(r2)</td>
</tr>
<tr>
<td>$T_{2,7}$</td>
<td>Wait(r3)</td>
</tr>
<tr>
<td>$T_{2,8}$</td>
<td>Assign(Z, X - Y)</td>
</tr>
<tr>
<td>$T_{2,9}$</td>
<td>Send(Ep2, Ep4, Z)</td>
</tr>
</tbody>
</table>

### 3.3. State-based encoding

A symbolic program execution is a finite sequence of program states, with each state is a mapping from symbolic variables to values. The first state of an execution maps all variables to initial values. Each consecutive state is derived by carrying out a single instruction. In the case of concurrent programs consisting of simultaneously running components, instructions are chosen from each component instructions sequence, one at a
time and are interleaved in some total order. The last state of a symbolic execution is that immediately following the execution of the last available instruction from the sequence of any component.

The state-based encoding produces a symbolic replica of the events in the trace and solving the formula produced by this encoding is equivalent to symbolically executing the trace with different orderings of the trace events with the goal of finding an ordering that violates any of the functional correctness properties of the program. In other words, when attempting solving the formula produced by the state-based encoding, the SMT solver generates and simulates the execution of all possible valid interleavings of a symbolic model of the original program. The formula consists of two parts: symbolic variables and constraints.

3.3.1. The symbolic variables

The symbolic variables correspond to the variables that appear in the program in addition to auxiliary variables. For a trace with $B$ events, there will be $B + 1$ symbolic states ($s^0$, $s^1$, ..., $s^j$, ..., $s^B$), such that $s^0$ is the state before carrying out any event’s action, and $s^j$ is the state at the $i$th time instant; after carrying out the $i$th event’s action. A state $s^j$ is a valuation of all symbolic variables at time instant $i$. To capture the $B + 1$ states, we create $B + 1$ copies for the variables in the trace. For example, $L_{n,x}^j$ denotes the copy of variable $L_{n,x}$ at the $i$th time instant.
At any instant of time, one action, called the *pending action*, at one node, called the *active node*, will be symbolically carried out. The node selector variable $NS^i$ identifies the node that will be active at time instant $i$. At any time instant $i$, the value of $NS^i$ is selected by the SMT solver. The selection of $NS^i$ value is not totally random, but is governed by *scheduling constraints*. The pending action in a node $N_n$ is identified using the node counter variable $NC_n$. The domain of a $NC_n$ is $\{1...|T_n|, \bot\}$. $NC_n=x$ indicates that the pending action in the node $N_n$ is $T_n=x$. $NC_n=\bot$ means that all actions in node $N_n$, has been symbolically executed.

The MCAPI runtime buffers associated with endpoints are modeled as queues. For a receiving endpoint $EP_{n,x}$ that receives a message or more, there will be a corresponding queue $Q_{n,x}$. $Q_n$ is the set of all queues needed for the receiving endpoints at node $N_n$. A queue $Q_{n,x}$ is encoded as an array and two variables $head_{n,x}$ and $tail_{n,x}$ that indicate the head and tail positions in the array. The MCAPI standard provides non-blocking send and receive calls: `mcapi_msg_send_i` and `mcapi_msg_recv_i`, respectively. MCAPI runtime uses request objects to track the status of a non-blocking call. A non-blocking call initiates an operation (i.e. a send or a receive operation), sets a request object to pending, and returns immediately. The completion of a non-blocking call could be checked by issuing the blocking call `wait`, and passing to it the request object associated with the non-blocking call. The `wait` call will return when the non-blocking call has completed. A non-blocking send is completed when the message has been delivered to the MCAPI runtime. A non-blocking receive is completed when a message has been retrieved from
the MCAPI runtime buffers. A request object will be encoded as a symbolic variable with three possible values: NLL, PND, and CMP.

3.3.2. The constraints

The state-based encoding is composed of four constraints: the initial constraint \( F_{\text{init}} \), the actions constraint \( F_{\text{acts}} \), the scheduling constraint \( F_{\text{sched}} \), and the property constraint \( F_{\text{prp}} \).

3.3.2.1. The initial constraint \( F_{\text{init}} \)

The initial constraint \( F_{\text{init}} \) assigns the values of the symbolic variables at time instant 0. All node counters are initialized to number 1. \( F_{\text{init}} \) is expressed as:

\[
\Lambda_{n=1}^{\mid R \mid} \left( (N C_n^0 = 1) \land \left( \Lambda_{p=1}^{\mid L_n \mid} L_{n,p}^0 = i v_{n,p} \right) \land \left( \Lambda_{q=1}^{\mid Q_n \mid} head_{n,q}^0 = tail_{n,q}^0 = 0 \right) \right) \tag{1}
\]

Where \( iv_{n,p} \) is the initial value for the variable \( L_{n,p} \). Note that the request variables used in a node \( N_n \), are among the node local variables \( (L_n) \), and that they are initialized to NLL.

3.3.2.2. The actions constraint \( F_{\text{acts}} \)

The actions constraint \( F_{\text{acts}} \) mimics the effect of carrying out a pending action. It is a conjunction of B constraints \( F_{\text{acts}} = \Lambda_{i=1}^{B} F_{\text{act}^i} \), such that \( F_{\text{act}^i} \) corresponds to the action chosen to be carried out at time instant \( i \). The \( F_{\text{act}^i} \) constraint is dependent on the action type as described below:
For an event \( T_{nx} = <y, \text{Guard}, \text{Action}> \) whose \( \text{Action} = \text{Assign}(v, \text{exp}) \), the corresponding constraint formula is:

\[
(NS^i = n \land NC^i_n = x \land \text{Guard}) \rightarrow (NC^{i+1}_n = y \land v^{i+1} = \text{exp}^i \land \delta(\{v^i\})) \tag{2}
\]

Formula 2 states that, at time instant \( i \), if node \( N_n \) is the active node \( (NS^i = n) \), node \( N_n \)'s node counter is equal to \( x \) \( (NC^i_n = x) \), and the \( \text{Guard} \) holds true, then the node counter in the time instant \( i + 1 \) is set to the order of the next action \( (NC^{i+1}_n = y) \), the value of variable \( v \) in the time instant \( i + 1 \) is set to the expression \( \text{Expr} \) at time instant \( i \) \( (v^{i+1} = \text{exp}^i) \), and that all local variables but \( v \) and all queues heads and tails should have in time instant \( i + 1 \), the same values they had in time instant \( i \) \( (\delta(\{v^i\})) \).

For an event \( T_{nx} = <y, \text{Guard}, \text{Action}> \) whose \( \text{Action} = \text{Send}(\text{src}, \text{dest}, \text{exp}) \), the corresponding constraint formula is:

\[
(NS^i = n \land NC^i_n = x \land \text{Guard}) \rightarrow (NC^{i+1}_{nid} = y \land Q_{\text{dest}}[\text{tail}^i_{\text{dest}}] = \text{exp}^i \land \text{tail}^{i+1}_{\text{dest}} = \text{tail}^i_{\text{dest}} + 1 \land \delta(\{\text{tail}^i_{\text{dest}}\}))
\tag{3}
\]

Formula 3 states that, at time instant \( i \), if node \( N_n \) is the active node \( (NS^i = n) \), node \( N_n \)'s node counter is equal to \( x \) \( (NC^i_n = x) \), and \( \text{Guard} \) holds true, then the node counter in the time instant \( i + 1 \) is set to \( y \) \( (NC^{i+1}_{nid} = y) \), the sent expression is enqueued to the destination endpoint queue \( Q_{\text{dest}}[\text{tail}^i_{\text{dest}}] = \text{exp}^i \land \text{tail}^{i+1}_{\text{dest}} = \text{tail}^i_{\text{dest}} + 1 \), and all local variables and all queues heads and tails but \( \text{tail}^i_{\text{dest}} \) should have in time instant \( i + 1 \), the same values they had in time instant \( i \) \( (\delta(\{\text{tail}^i_{\text{dest}}\})) \).
For an action $T_{n,x} = \langle y, Guard, Action \rangle$ whose $Action = \text{Recv}(recv, v)$, the corresponding constraint formula is:

$$(NS^i = n \land NC^i_n = x \land Guard) \rightarrow$$

$$\left( NC^{i+1}_n = y \land v^{i+1} = Q_{recv}[head^{i}_{recv}] \land head^{i+1}_{recv} = head^{i}_{recv} + 1 \land \delta(\{v^i, head^{i}_{recv}\}) \right)$$

Formula 4 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$’s node counter is equal to $x$ ($NC^i_n = x$), and the $Guard$ holds true, then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the received value is dequeued from the receiving endpoint queue ($v^{i+1} = Q_{recv}[head^i_{recv}] \land head^{i+1}_{recv} = head^i_{recv} + 1$) and all local variables but $v^i$ and all queues heads and tails but $= head^{i}_{recv}$ should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta(\{v^i, head^{i}_{recv}\})$).

For an event $T_{n,x} = \langle y, Guard, Action \rangle$ whose $Action = \text{Send}_i(src, dest, exp, req)$, the corresponding constraint formula is:

$$(NS^i = n \land NC^i_n = x \land Guard) \rightarrow$$

$$\left( NC^{i+1}_n = y \land Q_{dest}[tail^i_{dest}] = exp^i \land tail^{i+1}_{dest} = tail^i_{dest} + 1 \land req^{i+1} = PND \land \delta(\{tail^i_{dest}, req^i\}) \right)$$

(5)
Formula 5 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$’s node counter is equal to $x$ ($NC^i_n = x$), and $Guard$ holds true, then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the sent expression is enqueued to the destination endpoint queue ($Q_{dest}[tail_{dest}^i] = exp^i \land tail_{dest}^{i+1} = tail_{dest}^i + 1$), the value of the request variable is set to pending ($req^{i+1} = PND$), and all local variables but $req^i$ and all queues heads and tails but $tail_{dest}^i$ should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta([tail_{dest}^i, req^i])$).

For an event $T_{n,x} = < y, Guard, Action >$ whose $Action = Recv_i(recv, v, req)$, the corresponding constraint formula is the conjunction of F6 and F7:

$$\begin{align*}
(NS^i = n \land NC^i_n = x \land Guard \land head_{recv}^i = tail_{recv}^i) & \to \\
(NC^{i+1}_n = y \land req^{i+1} = PND \land \delta([req^i]))
\end{align*}$$

$$\begin{align*}
(NS^i = n \land NC^i_n = x \land Guard \land head_{recv}^i \neq tail_{recv}^i) & \to \\
(NC^{i+1}_n = y \land v^{i+1} = Q_{recv}[head_{recv}^i] \land head_{recv}^{i+1} = head_{recv}^i + 1 \land req^{i+1} \\
& = CMP \land \delta([req^i, head_{recv}^i, v^i]))
\end{align*}$$

Formula 6 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$’s node counter is equal to $x$ ($NC^i_n = x$), the $Guard$ holds true, and the receiving endpoint queue is empty ($head_{recv}^i = tail_{recv}^i$), then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the request variable is set to pending ($req^{i+1} = PND$), and
all local variables but $req^i$ and all queues heads and tails should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta([req^i])$).

Formula 7 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$'s node counter is equal to $x$ ($NC^i_n = x$), the Guard holds true, and the receiving endpoint queue is not empty ($head^i_{recv} \neq tail^i_{recv}$), then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the request variable is set to complete ($req^{i+1} = CMP$), the received value is dequeued from the receiving endpoint queue ($v^{i+1} = Q_{recv}[head^i_{recv}] \land head^{i+1}_{recv} = head^i_{recv} + 1$) and all local variables but $v^i$ and $req^i$ and all queues heads and tails but $head^i_{recv}$ should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta([v^i, head^i_{recv}, req^i])$).

For an event $T_{nx} =< y, Guard, Action >$ whose $Action = Wait(req)$, such that $req$ is associated with a non-blocking send, the corresponding constraint formula is:

$$ (NS^i = n \land NC^i_n = x \land Guard) \rightarrow \left( NC^{i+1}_n = y \land req^{i+1} = CMP \land \delta([req^i]) \right) \ (8) $$

Formula 8 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$'s node counter is equal to $x$ ($NC^i_n = x$), and the Guard holds true, then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the request variable is set to complete ($req^{i+1} = CMP$), and all local variables but $req^i$ and all queues heads and tails should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta([req^i])$).
For an event $T_{n,x} = < y, \text{Guard}, \text{Action} >$ whose Action = Wait($req$), such that $req$ is associated with a non-blocking receive action with Action = Recv$_i$($recv, v, req$), the corresponding constraint formula is the conjunction of F9 and F10:

\[
(NS^i = n \land NC^i_n = x \land \text{Guard} \land req^i = PND) \implies \\
\left( NC^{i+1}_n = y \land v^{i+1} = Q_{recv}[head^{i}_{recv}] \land head^{i+1}_{recv} = head^{i}_{recv} + 1 \land req^{i+1} = CMP \right)
\]

\[
= CMP \land \delta\left(\{req^i, head^i_{recv}, v^i\}\right)
\]

\[
(NS^i = n \land NC^i_n = x \land \text{Guard} \land req^i = CMP) \implies \\
\left( NC^{i+1}_n = y \land \delta(\phi) \right)
\]

Formula 9 states that, at time instant $i$, if node $N_n$ is the active node ($NS^i = n$), node $N_n$'s node counter is equal to $x$ ($NC^i_n = x$), the Guard holds true, and the request is pending ($req^i = PND$), then the node counter in the time instant $i + 1$ is set to $y$ ($NC^{i+1}_n = y$), the request variable is set to complete ($req^{i+1} = CMP$), the received value is dequeued from the receiving endpoint queue ($v^{i+1} = Q_{recv}[head^i_{recv}] \land head^{i+1}_{recv} = head^i_{recv} + 1$) and all local variables but $v^i$ and $req^i$ and all queues heads and tails but $head^i_{recv}$ should have in time instant $i + 1$, the same values they had in time instant $i$ ($\delta(\{v^i, head^i_{recv}, req^i\})$).
Formula 10 states that, at time instant \( i \), if node \( N_n \) is the active node \((NS^i_n = n)\), node \( N_n \)'s node counter is equal to \( x \) \((NC^i_n = x)\), the Guard holds true, and the request is complete \((req^i = CMP)\), then the node counter in the time instant \( i + 1 \) is set to the order of the next action \((NC^{i+1}_n = y)\) and all local variables and all queues heads and tails should have in time instant \( i + 1 \), the same values they had in time instant \( i \) \((\delta(\phi))\).

3.3.2.3. The scheduling constraint \((F_{sched})\)

Like the actions constraint, the scheduling constraint \((F_{sched})\) is the conjunction of \( B \) constraints \((F_{sched} = \bigwedge_{i=1}^{B} F_{sched}^i)\). Each \( F_{sched}^i \) constraint consists of four parts that ensure that:

1. A node that is done carrying out all its actions, will not be an active node:

\[
\bigwedge_{n=1}^{\|\mathcal{R}\|} [(NC^i_n = \bot) \rightarrow (NS^i \neq n)] \tag{11}
\]

2. The variables of an inactive node will not change:

\[
\bigwedge_{n=1}^{\|\mathcal{R}\|} [(NS^i \neq n) \rightarrow \left( \bigwedge_{v=1}^{\|\mathcal{L}_n\|} L_{n,v}^{i+1} = L_{n,v}^i \right) \land (NC^{i+1}_n = NC^i_n)] \tag{12}
\]

3. A node with a pending blocking receive action whose destination queue is empty, will not be an active node:

\[
\bigwedge_{n=1}^{\|\mathcal{R}\|} [(NC^i_n = x \land Action_{n,x} = \text{Recv}(recv, v) \land head_{recv}^i = \text{tail}_{recv}^i) \rightarrow (NS^i \neq n)] \tag{13}
\]
4. A node with a pending wait action of a non-blocking receive whose destination queue is empty, will not be an active node:

\[
\bigwedge_{n=1}^{|\mathcal{R}|} [NC_n^i = x \land Action_{n,x}^i = Wait(req) \land \exists Recv_i(recv, v, req) \land head_{recv}^i = tail_{recv}^i] \quad (14)
\]

\[\rightarrow (NS^i \neq n)]\]

3.3.2.4. The property constraint \((\mathcal{F}_{prp})\)

The property constraint \((\mathcal{F}_{prp})\) captures the functional correctness properties of the program and is derived from events whose \(Action = Assert(exp)\) as follows:

\[
\bigwedge_{n=1}^{|\mathcal{R}|} [NC_n^i = x \land Action_{n,x}^i = Assert(exp) \land Guard \rightarrow Expr^i] \quad (15)
\]

The overall formula that is passed to the SMT solver, is constructed as follows:

\[
\mathcal{F}_\mathcal{R} = \mathcal{F}_{init} \land \mathcal{F}_{acts} \land \mathcal{F}_{sched} \land \neg \mathcal{F}_{prp} \quad (16)
\]

If the formula 16 is satisfiable, then its solution represents an execution scenario that violates one or more of the functional correctness constraints encoded by \(\mathcal{F}_{prp}\).

3.3.3. Experimental results

Due to the lack of publically available benchmarks for MCAPI programs, we conducted our experiments using programs and benchmarks developed by ourselves. Our
experiments were conducted on a machine with Core 2 Duo 1.4 GHz CPU and 4 GB RAM.

In the first iteration of experiments, we applied the state-based encoding on the traces of 9 MCAPI applications with varying features. The generated formulas are then input to Yices. We assess the applicability of the state-based encoding by measuring the time and memory needed by Yices to solve the formula generated by the encoding. The runtime and memory consumed by Yices to solve the formulas are depicted in Figure 3.2. The x-axis reports the number of the events in the input trace. These results looked promising as all the formulas in the experiment could be solved in a fraction of a second and required a small amount of memory.

![Graph A](image1.png) ![Graph B](image2.png)

**Figure 3.2. Results of the first iteration of experiments on state-based encoding**

However, applying the state-based encoding on the traces of more complicated programs revealed that it doesn’t scale well. In the second iteration of experiments, we used these benchmarks:
1) Binary Tree benchmark (BT): This is a set of 10 programs that create networks of nodes with sizes from 3 nodes to 21 nodes. Each two nodes send a message to the same parent node forming a binary tree in which messages travel from the leaves to the root node. The smallest tree has 3 nodes and exchanges 4 messages. The largest one has 21 nodes and exchanges 31 messages. This benchmark has a master/slave communication pattern.

2) Complete Graph benchmark (CG): This is a set of 10 programs that create networks of nodes with increasing sizes from 2 nodes to 11 nodes. All nodes send and receive messages to/from each other forming a complete graph. The number of exchanged messages is between 2 message (for a 2 nodes graph) and 110 messages (for a 11 nodes graph). This benchmark has an all-to-all communication pattern.

3) 10-nodes benchmark (TN): This is a set of 10 programs that create networks of nodes with a fixed size of 10 nodes. However, the number of messages exchanged among the nodes increases monotonically. The number of messages exchanged is between 10 and 100.

Figure 3.3 depicts the results of applying the state-based encoding on the traces of the BT benchmark programs. Only the results of the first six programs are available. Starting from the seventh program, Yices exceeded a time-out of 5 minutes. The programs of the other benchmarks started timing-out at the third program for the CG benchmark and at the second program for the TN benchmark.
The reason behind the poor performance of the state-based encoding is rooted in using arrays to encode the MCAPI runtime buffers. Available SMT solvers can't handle formulas with arrays efficiently. In the next section, we present the order-based encoding which is array-free.

3.4. Order-based encoding

Given a trace $\mathcal{R}$, we create a quantifier-free first-order logic formula $F_{\mathcal{R}}$ that is satisfiable iff there exists a feasible permutation $P_{\mathcal{R}}$ of the events in $\mathcal{R}$ that leads to an error state (e.g. a violation of the functional correctness). A feasible permutation is a strict total order of all the events in $\mathcal{R}$, such that this order can occur in a real execution of the original program. The symbolic variables and constraints of $F_{\mathcal{R}}$ are presented in sections 3.4.1 and 3.4.2, respectively.

3.4.1. The symbolic variables

In this encoding, there are two types of symbolic variables: 1) for every event $T \in \mathcal{R}$, there is a symbolic variable $O_T$ that reflects the order of carrying out $T$ in $P_{\mathcal{R}}$. 2) for every action that assigns a new value to a local variable $L$, we create a new symbolic
variable for $L$ (e.g. $L_i$ corresponds to the value of the variable $L$ after the $i$th assignment).

The values of these symbolic variables record the history of the values of $L$. This is similar to the SSA form [40]. While the SSA form requires $\phi$-functions to handle the effect of branches, we needn't have $\phi$-functions because in a trace all branching decisions have already been made. We add two dummy variables $O_{\text{First}}$ and $O_{\text{Last}}$, such that $O_{\text{First}}$ is the first event in $P_R$ and $O_{\text{Last}}$ is the last event in $P_R$. The values assigned to these symbolic variables are governed by constraints that are crafted to ensure that $P_R$ is a feasible permutation. A symbolic variable $O_{i,j}$ represents the order of the event $T_{i,j}$.

Table 3.2 shows the symbolic variables that are needed for encoding the trace in Table 3.1. A symbolic variable $O_{i,j}$ represents the order of the event $T_{i,j}$. A symbolic variable $T_{i,V_j}$ corresponds to the value of the variable $V$ at sub-trace $T_i$ after being assigned a new value for the $j$th time.
Table 3.2. Symbolic variables of the trace in Table 3.1

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_{1.1}$</td>
<td>$O_{2.1}$</td>
<td>$O_{3.1}$</td>
<td>$O_{4.1}$</td>
</tr>
<tr>
<td>$O_{1.2}$</td>
<td>$O_{2.2}$</td>
<td>$O_{3.2}$</td>
<td>$O_{4.2}$</td>
</tr>
<tr>
<td>$O_{1.3}$</td>
<td>$O_{2.3}$</td>
<td>$O_{3.3}$</td>
<td>$O_{4.3}$</td>
</tr>
<tr>
<td>$O_{1.4}$</td>
<td>$O_{2.4}$</td>
<td>$T_3Msg_1$</td>
<td>$O_{4.4}$</td>
</tr>
<tr>
<td>$O_{1.5}$</td>
<td>$O_{2.5}$</td>
<td></td>
<td>$O_{4.5}$</td>
</tr>
<tr>
<td>$T_1Msg_1$</td>
<td></td>
<td></td>
<td>$O_{4.6}$</td>
</tr>
<tr>
<td></td>
<td>$O_{2.6}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$O_{2.7}$</td>
<td></td>
<td>$O_{4.7}$</td>
</tr>
<tr>
<td></td>
<td>$O_{2.8}$</td>
<td></td>
<td>$T_4W_1$</td>
</tr>
<tr>
<td></td>
<td>$T_2X_1$</td>
<td></td>
<td>$T_4O_1$</td>
</tr>
<tr>
<td></td>
<td>$T_2Y_1$</td>
<td></td>
<td>$T_4U_1$</td>
</tr>
<tr>
<td></td>
<td>$T_2Z_1$</td>
<td></td>
<td>$T_4W_2$</td>
</tr>
<tr>
<td></td>
<td>$T_2X_2$</td>
<td></td>
<td>$T_4O_2$</td>
</tr>
<tr>
<td></td>
<td>$T_2Y_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_2Z_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.4.2. The constraints

The order-based formula has four constraints: the order constraint ($F_{\text{order}}$), the assignment constraint ($F_{\text{asgn}}$), the receive constraint ($F_{\text{recv}}$), and the property constraint ($F_{\text{prp}}$). The $F_R$ formula is the conjunction of these four constraints:

$$F_R := F_{\text{order}} \land F_{\text{asgn}} \land F_{\text{recv}} \land \neg F_{\text{prp}} \quad (17)$$

3.4.2.1. The order constraint ($F_{\text{order}}$)

$F_{\text{order}}$ ensures that in $P_R$, no two events are assigned the same ordering and that every two events $T_{i_x}$ and $T_{i_y}$, such that $x < y$ (i.e. event $T_{i_x}$ appears in the trace before event $T_{i_y}$) will be assigned orderings $O_{i_x}$ and $O_{i_y}$, such that $O_{i_x} < O_{i_y}$. $F_{\text{order}}$ is constructed using the algorithm Construct_FOrder in Figure 3.4.
1 \( F_{\text{order}} := \text{true} \)
2 \( \text{for } i = 1 \text{ to } n \)
3 \( F_{\text{order}} := F_{\text{order}} \land (O_{\text{First}} < O_{T_{i,1}}) \)
4 \( \text{for } j = 1 \text{ to } |T_i| \)
5 \( \text{if } (j < |T_i|) \text{ then } F_{\text{order}} := F_{\text{order}} \land (O_{T_{i,j}} < O_{T_{i,j+1}}) \)
6 \( \text{for } k = i + 1 \text{ to } n \)
7 \( \text{for } l = 1 \text{ to } |T_k| \)
8 \( F_{\text{order}} := F_{\text{order}} \land (O_{T_{i,j}} \neq O_{T_{k,l}}) \)
9 \( \text{end-for} \)
10 \( \text{end-for} \)
11 \( F_{\text{order}} := F_{\text{order}} \land (O_{\text{Last}} > O_{T_{i,j}}) \)
12 \( \text{end-for} \)

Figure 3.4. The Construct\_FOrder() algorithm

3.4.2.2. The assignment constraint (\( F_{\text{asgn}} \))

\( F_{\text{asgn}} \) encodes events with assignment actions. \( F_{\text{asgn}} \) is initially set to true. For every event \( T_{i,x} \) whose action is \( \text{Assign}(v, \exp) \):

\[ F_{\text{asgn}} := F_{\text{asgn}} \land (S(v) = S(\exp) \land S(\text{Guard})) \] (18)

Where \( S(v), S(\exp) \) and \( S(\text{Guard}) \) replace the program variables with the corresponding symbolic ones.

3.4.2.3. The receive constraint (\( F_{\text{recv}} \))

\( F_{\text{recv}} \) encodes the events with an action that is either a blocking receive, or a wait of a non-blocking receive. To facilitate describing the \( F_{\text{recv}} \) constraint, we use the following notations:

For every event \( T_{i,x} \) whose Action is either \( \text{Send}(\text{src}, \text{dest}, \exp) \) or \( \text{Send}_{i}(\text{src}, \text{dest}, \exp, \text{req}) \):
• \( \text{DestEP}(T_{ix}) = \text{dest} \)

• \( \text{Exp}(T_{ix}) = \text{exp} \)

• \( \text{SOrder}(T_{ix}) \) is the order of \( T_{ix} \) with respect to other events in \( T_i \) whose actions are either \( \text{Send}(\text{src}, \text{dest}, \text{exp}) \) or \( \text{Send}_i(\text{src}, \text{dest}, \text{exp}, \text{req}) \) and have the same destination endpoint as \( T_{ix} \).

For every event \( T_{ix} \) whose \( \text{Action} \) is either \( \text{Recv}(\text{recv}, \text{v}) \) or \( \text{Wait}(\text{req}) \) such that \( \text{Wait}(\text{req}) \) is associated with a non-blocking receive action \( \text{Recv}_i(\text{recv}, \text{v}, \text{req}) \):

• \( \text{RecvEP}(T_{ix}) = \text{recv} \)

• \( \text{Var}(T_{ix}) = \text{v} \)

• \( \text{ROrder}(T_{ix}) \) is the order of \( T_{ix} \) with respect to other events in \( T_i \) whose actions are either \( \text{Recv}(\text{recv}, \text{v}) \) or \( \text{Wait}(\text{req}) \) such that \( \text{Wait}(\text{req}) \) is associated with a non-blocking receive action \( \text{Recv}_i(\text{recv}, \text{v}, \text{req}) \) and have the same receiving endpoint as \( T_{ix} \).

• \( \mathcal{S}_{ix} \) is the set of events whose actions are either \( \text{Send}(\text{src}, \text{dest}, \text{exp}) \) or \( \text{Send}_i(\text{src}, \text{dest}, \text{exp}, \text{req}) \) and can potentially match with the receive action of \( T_{ix} \). \( \mathcal{S}_{ix} \) is defined as:

\[
\mathcal{S}_{ix} = \{ T_{ij} | \text{DestEP}(T_{ij}) = \text{RecvEP}(T_{ix}) \land \text{ROrder}(T_{ix}) \geq \text{SOrder}(T_{ij}) \}.
\]

We call \( \mathcal{S}_{ix} \), the set of potential sender events of \( T_{ix} \).

• \( \mathcal{P}_{ix} \) is the set of events whose actions are 1) either \( \text{Recv}(\text{recv}, \text{v}) \) or \( \text{Wait}(\text{req}) \) such that \( \text{Wait}(\text{req}) \) is associated with a non-blocking receive \( \text{Recv}_i(\text{recv}, \text{v}, \text{req}) \) 2) precede \( T_{ix} \) in \( T_i \), and 3) have the same receiving
endpoint as \( T_{i,x} \). \( \mathcal{P}_{i,x} \) is defined as \( \mathcal{P}_{i,x} = \{ T_{i,y} \mid \text{ROrder}(T_{i,y}) < \text{ROrder}(T_{i,x}) \} \).

We call \( \mathcal{P}_{i,x} \), the set of related preceding receiving events of \( T_{i,x} \).

\( F_{\text{recv}} \) is initially set to true. For an event \( T_{i,x} \) whose action is either \( \text{Recv} (\text{recv}, v) \) or \( \text{Wait} (\text{req}) \) such that \( \text{Wait} (\text{req}) \) is associated with a non-blocking receive \( \text{Recv}_i (\text{recv}, v, \text{req}) \):

\[
F_{\text{recv}} := F_{\text{recv}} \land \forall s \in S_{i,x} (S(Va r(T_{i,x})) = S(\text{Exp}(s)) \land S(\text{Guard}) \land \text{CON}_{T_{i,x}}^s \land \forall p \in \mathcal{P}_{i,x} \neg \text{CON}_{p}^s) 
\]

(19)

\[
\text{CON}_{p}^s = (O_s < O_r) \land \forall n \in S_r \land n \neq s \ (O_n < O_s) \lor (O_r < O_n))
\]

(20)

\( \text{CON}_{p}^s \) encodes the conditions needed for matching an event \( s \) with a send action to an event \( r \) with a receive action. These conditions are 1) \( s \) must precede \( r \) \((O_s < O_r)\), and 2) for every event \( n \), such that \( n \in S_r \land n \neq s \), then either \( n \) is before \( s \) or \( r \) is before \( n \) \((\forall n \in S_r \land n \neq s ((O_n < O_s) \lor (O_r < O_n)))\).

Formula 19 states that the receive action of \( T_{i,x} \) will be matched with the event \( s \), when the conditions for this matching are satisfied \((\text{CON}_{T_{i,x}}^s)\), and when all the conditions needed for matching \( s \) with any event in \( \mathcal{P}_{i,x} \), are not satisfiable \((\forall p \in \mathcal{P}_{i,x} \neg \text{CON}_{p}^s)\).

For example, the part of \( F_{\text{recv}} \) that corresponds to the event \( T_{4,4} \) is the disjunction of the formulas 21, 22 and 23. Formulas 21, 22, and 23 match the receive action at event \( T_{4,4} \)
with the send action at events $T_{1,3}$, $T_{2,9}$ and $T_{3,3}$ respectively and encodes the necessary conditions. Only one formula of these three formulas will be satisfied.

\begin{align}
 (T_{4}U_{2}=T_{1}Msg_{1} \land (O_{1,3} < O_{4,4} \land (((O_{3,3}<O_{1,3}) \lor (O_{4,4}<O_{3,3})) \land ((O_{2,9}<O_{1,3}) \lor (O_{4,4}<O_{2,9})))))) \tag{21}
\end{align}

\begin{align}
 (T_{4}U_{2}=T_{2}Z_{1} \land (O_{2,9} < O_{4,4} \land (((O_{3,3}<O_{2,9}) \lor (O_{4,4}<O_{3,3})) \land (O_{1,3}<O_{2,9}) \lor (O_{4,4}<O_{2,9})))))) \tag{22}
\end{align}

\begin{align}
 (T_{4}U_{2}=T_{3}Msg_{1} \land (O_{3,3} < O_{4,4} \land (((O_{2,9}<O_{3,3}) \lor (O_{4,4}<O_{2,9})) \land (O_{1,3}<O_{3,3}) \lor (O_{4,4}<O_{3,3})))))) \tag{23}
\end{align}

Intuitively, $F_{recv}$ matches an event $s \in S_{r}$ with one event $r$, provided that $s$ has not been matched with any event $p \in P_{r}$, and $s$ can occur before $r$. The effect of a matching is assigning the valuation of the expression sent by $s$ to the variable of $r$.

3.4.2.4. The property constraint ($F_{prp}$)

The property constraint ($F_{prp}$) captures the functional correctness properties of the program and is derived from events with $Assert(exp)$ actions as follows:

\[ F_{prp} \text{ is initially set to true. For every event } T_{ix} \text{ whose action as } Assert(exp): \]
\[ F_{prp}': = F_{prp} \land (S(exp) \land S(Guard)) \tag{24} \]
After the formula $F_R$ has been constructed, it is passed to an SMT solver. If $F_R$ is satisfiable, then the SMT solver will produce a solution that assigns a value for every $O_T$ variable that indicates the order of carrying out the event $T$ in the permutation $P_R$.

3.4.3. Experimental results

Figure 3.5 depicts the results of applying the order-based encoding on the traces of the BT benchmark programs.

![Figure 3.5](image)

(a) ![Figure 3.5](image) (b)

**Figure 3.5. Results of applying order-based encoding on the BT benchmark**

Comparing Figure 3.3 and Figure 3.5 shows clearly that the order-based encoding scales better than the state-based encoding.

Figure 3.6 and Figure 3.7 depict the results of applying the order-based encoding on the traces of the CG and TN benchmarks, respectively.
The graphs in figures Figure 3.5, Figure 3.6 and Figure 3.7 show that the order-based encoding exhibits better scalability than the step-based encoding in terms of time and memory usage.

3.5. Reporting a violating scenario

Table 3.3 shows the solution produced by Yices for the formula that corresponds to the trace in Table 3.1 for the order-based encoding.
According to the trace in Table 3.1, any of the events \((T_{1,3}, T_{3,3}, \text{and } T_{2,9})\) whose actions are send actions can match with the event \(T_{4,4}\) whose action is a receive action. In Table 3.3 \(O_{2,9}<O_{4,4}, O_{1,3}>O_{4,4}\) and \(O_{3,3}>O_{4,4}\) indicating that \(T_{2,9}\) is the event that will be matched with \(T_{4,4}\).

The mzReporter receives a Yices solution as an input and translates it to user-friendly report. Table 3.4 shows the output of the mzReporter that corresponds to the solution in Table 3.3.
The first column shows the order of carrying the actions which are listed in the third column. When the action is a msg_recv or a wait of a recv_i, the fourth column shows the change in the variable that is receiving the arriving message. This report describes a concrete execution scenario that leads to an assertion failure. The actual report includes a fifth column showing the line number of each action in the input source code.

### 3.6. Conclusion

We have presented a methodology and a proof-of-concept implementation for predicting runtime failures in MCAPI programs by symbolically enumerating and examining all permutations of the events in an execution trace. We have developed two different
encodings that are used to translate an execution trace to a quantifier-free first-order logic formula.

Our approach is practical, scalable and sound. It is practical, because it is fully automated and does not require manual annotating of the source code. Our experimental results show that our approach is scalable; particularly when using the second encoding. It is sound as no spurious execution scenarios are reported.

Constructing the symbolic formula from the trace, instead of the program leads to two limitations: 1) our approach lacks completeness as some bugs in the input program may escape detection due to the fact that we are considering a trace of the program, rather than the program itself, 2) also, our approach can’t handle non-determinism arising from routines such as mcapi_wait_any, mcapi_test and random numbers generation.

These limitations could be overcome by combining elements from the input program with the trace. Another solution is to repeatedly execute the program to obtain a different trace and hence increasing the analysis coverage, however, there are no guarantees that different executions will produce different traces. Also, MCAPI programs are vulnerable to deadlocks [41]. We plan exploring how to encode a MCAPI program trace such that it is possible to predict deadlocks.
CHAPTER 4

DETERMINISTIC REPLAY FOR MCAPI

In this chapter, we present DR-MCAPI, the first tool for deterministically replaying MCAPI programs executions. DR-MCAPI works by monitoring a program execution to generate a trace. If the program fails, the trace can be used to produce an execution that is logically equivalent to the one that had failed. Since MCAPI programs executions are inherently irreproducible, providing a deterministic replay capability allows developers to find the failure source. DR-MCPAI supports two replay approaches: data-replay and order-replay. Each approach has its own particular strengths and weaknesses. In section 4.1, we introduce deterministic replay and its applications. Section 4.2 describes the workflow of DR-MCAPI. We describe the data-replay and the order-replay approaches in sections 4.3 and 4.4, respectively. Section 4.5 shows the experimental results. Section 4.6 discusses the features of DR-MCAPI. In section 4.7, we present future work.

4.1. Introduction

If two executions of a program exhibit the same set of instructions with each instruction computing the same results and producing the same final values in memory, then these two executions are said to be logically equivalent [42]. A deterministic replay of a program is a controlled execution that is logically equivalent to a previous execution of interest. Deterministic replay has various applications such as cyclic debugging, fault tolerance and intrusion analysis [43].
In cyclic debugging, a program is repeatedly executed under the control of a debugger to allow the user to obtain more information about the program states and intermediate results [44]. Cyclic debugging assumes that different executions of the same program with the same input will be equivalent. Different executions of a concurrent program are not guaranteed to be equivalent as concurrent programs suffer from the irreproducibility effect [45] due to their intrinsic non-determinism. The fact that two subsequent runs of the same program with the same input are not guaranteed to behave the same or produce the same output; makes cyclic debugging of concurrent programs a challenging task. Cyclic debugging is the most prominent application of deterministic replay and is called Deterministic Replay Debugging (DRD) [42].

Within the context of fault-tolerance, deterministic replay has been used to detect hardware design faults by scrutinizing the variances between a replayed execution on a machine and an original execution on another machine [46]. Also, in the case of a program failure, a replayed execution can be used to reconstruct the most recent program state [47]. The ReVirt system [48] shows that deterministic replay is useful for intrusion analysis. ReVirt allows replaying the execution of a whole computer system before, during, and after a system has been compromised facilitating post-attack analysis.

As depicted in Figure 4.1, the deterministic replay process consists of two phases: recording and replay. During a recording phase, a program execution is monitored by a recording environment to record information about the execution in a trace file. When a replay is needed, the data in the trace file is used to replay the program within a replay
environment such that the behavior of the program during the replay phase is logically equivalent to the behavior observed in the recording phase.

![Diagram of deterministic replay phases](image)

**Figure 4.1. The two phases of deterministic replay**

4.2. DR-MCAPI: Deterministic Replay for MCAPI Programs

In this section, we describe DR-MCAPI and its different replay techniques.

4.2.1. DR-MCAPI workflow

Figure 4.2 depicts the workflow of our tool for deterministic replay of MCAPI programs. DR-MCAPI consists of two parts: a source code instrumenter and a MCAPI library wrapper (DR-MCAPI Library).
4.2.2. Source instrumentation

An input MCAPI program is instrumented by replacing all calls to the MCAPI library routines with calls to the DR-MCAPI library routines and by adding extra calls for initializing and finalizing the recording/replay process. Figure 4.3 shows the result of instrumenting a portion of the program in Figure 2.1. We use the ROSE compiler [49] to automate the instrumentation process.

```c
04 void NODE1 () {
05    mcapi_status_t Status;
06    mcapi_version_t Version;
08    int X=1;
09
10   dr_initialize(1,&Version,&Status);
11   LocalEP = dr_create_endpoint (1,&Status);
12   N2EP = dr_get_endpoint (2,1,&Status);
13   N4EP = dr_get_endpoint (4,1,&Status);
14   N5EP = dr_get_endpoint (5,1,&Status);
15   dr_msg_send(LocalEP, N2EP, &X, sizeof (X), 1, &Status);
16   dr_msg_send(LocalEP, N4EP, &X, sizeof (X), 1, &Status);
17   dr_msg_send(LocalEP, N5EP, &X, sizeof (X), 1, &Status);
18   dr_delete_endpoint(LocalEP,&Status);
19   dr_finalize(&Status);
20 }
```

Figure 4.3. An instrumented MCAPI program snippet
The DR-MCAPI library acts as a layer between the program and the MCAPI library as shown in Figure 4.4. When an instrumented program is run, the program invokes the DR-MCAPI routines which will carry out some processing and call the original MCAPI routine. For example, a call to dr_create_ep will add a new endpoint to a list of endpoints maintained for every node by DR-MCAPI and then mcapi_create_ep will be invoked.

![Diagram](image)

**Figure 4.4. DR-MCAPI sits between the MCAPI library and the instrumented program**

### 4.2.3. Operating modes

An instrumented program can run in one of two possible operating modes: recording mode or replay mode. While a program is running in the recording mode, calls to the DR-MCAPI library routines record certain information in addition to invoking MCAPI library routines. When the program execution ends (either normally or due to a failure), the recorded information is stored to the disk as a trace. During the recording mode, DR-MCAPI doesn’t affect the outcomes of non-deterministic operations. When run in the replay mode, the trace information are loaded into memory and are used by DR-MCAPI
library to force an execution that is equivalent to the one observed when the program was
running in the recording mode.

Figure 4.5 depicts the pseudocode of the program in Figure 2.1. We will be using this
pseudocode as an ongoing example.

```
Node 1
1  dr_initialize(1);
2  X=1;
3  EP=dr_create_endpoint(1);
4  N2EP=dr_get_endpoint(2,1);
5  N4EP=dr_get_endpoint(4,1);
6  N5EP=dr_get_endpoint(5,1);
7  dr_msg_send(EP,N2EP,X);
8  dr_msg_send(EP,N4EP,X);
9  dr_msg_send(EP,N5EP,X);
10 dr_delete_endpoint(EP);
11 dr_finalize();

Node 2
23 dr_initialize(2);
24 EP=dr_create_endpoint(1);
25 N4EP=dr_get_endpoint(4,1);
26 N5EP=dr_get_endpoint(5,1);
27 dr_msg_recv_i(EP,&A,Rs[0]);
28 dr_msg_recv_i(EP,&B,Rs[1]);
29 if(dr_wait_any(2,Rs)==0)
30   dr_wait(Rs[1]);
31 else
32   dr_wait(Rs[0]);
33   C=Func(A,B);
34 dr_msg_send(EP,N4EP,C);
35 dr_msg_send(EP,N5EP,C);
36 dr_delete_endpoint(EP);
37 dr_finalize();

Node 3
12 dr_initialize(3);
13 Y=1;
14 EP=dr_create_endpoint(1);
15 N2EP=dr_get_endpoint(2,1);
16 N4EP=dr_get_endpoint(4,1);
17 N5EP=dr_get_endpoint(5,1);
18 dr_msg_send(EP,N2EP,Y);
19 dr_msg_send(EP,N4EP,Y);
20 dr_msg_send(EP,N5EP,Y);
21 dr_delete_endpoint(EP);
22 dr_finalize();

Node 4
38 dr_initialize(4);
39 EP=dr_create_endpoint(1);
40 dr_msg_recv_i(EP,&D,&R1);
41 dr_msg_recv_i(EP,&E,&R2);
42 dr_msg_recv(EP,&F);
43 while(!dr_test(R1));
44 dr_wait(R2);
45 dr_delete_endpoint(EP);
46 dr_finalize();

Node 5
47 dr_initialize(5);
48 EP=dr_create_endpoint(1);
49 dr_msg_recv(EP,&G);
50 dr_msg_recv(EP,&H);
51 dr_msg_recv(EP,&I);
51 dr_delete_endpoint(EP);
52 dr_finalize();
```

Figure 4.5. Pseudocode of the program in Figure 2.1
Replay tools for message-passing programs typically fall into two categories: data-replay and order-replay. DR-MCAPI supports both data-replay and order-replay. Section 4.3 describes how DR-MCAPI implements data-replay.

4.3. DR-MCAPI data-replay

During a recording execution, the contents of all received messages at all nodes are stored. During a replay execution, some processes are run while others are simulated. The messages sent by the simulated processes originate from the trace and not from the program. The data-replay approach generates a huge trace. However, it allows replaying one or more specific nodes. First, we describe the data-replay trace structure in section 4.3.1 and then the data-replay replay mechanism in section 4.3.2.

4.3.1. The trace structure

When an instrumented program $P$ is run in the record mode, a separate trace is generated for each MCAPI node:

$$Trace_P = \{Trace^1, ..., Trace^N\}, \text{ where } N \text{ is the number of nodes in program } P.$$ 

A node’s trace contains a list of records:

$$Trace^n = \{Record^1, ..., Record^{|Trace^n|}\}, \text{ where } n \text{ is a node identifier.}$$ 

A trace record may be any of six types:

$$Record \in \{\text{Recv} \cup \text{Wait} \cup \text{RecvWany} \cup \text{NonRecvWany} \cup \text{ArrivalTest} \cup \text{NonArrivalTest} \cup \text{Rand}\}$$

56
A *Recv* record originates from a *dr_msg_recv* call and is defined as a tuple: \( \text{Recv} \in \text{Port} \times \text{RecvOrder} \times \text{Data} \). *Port* is the port number of the receiving endpoint. *RecvOrder* is the invocation order of this particular *dr_msg_recv* call among other *dr_msg_recv* calls at this node. *Data* is the payload of the received message.

A *Wait* record originates from a *dr_wait* call whose input request variable was initialized by a *dr_msg_recv_i* call and is defined as tuple: \( \text{Wait} \in \text{ReqInitOrder} \times \text{Data} \). *ReqInitOrder* is the initialization order of the input request variable at the current node.

A *RecvWany* record comes from a *wait_any* call that returned the index of a request variable that was initialized by a *msg_recv_i* call and is defined as tuple: \( \text{RecvWany} \in \text{WanyOrder} \times \text{Index} \times \text{Data} \). *WanyOrder* is the invocation order of this particular *wait_any* call among other *wait_any* calls at this node. *Index* is the index returned by the *wait_any* call.

The record *NonRecvWany* is defined as tuple: \( \text{NonRecvWany} \in \text{WanyOrder} \times \text{Index} \) and indicates that a *wait_any* call returned the index of a request variable that was initialized by a non-blocking function other than *msg_recv_i*.

An *ArrivalTest* originates from a sequence (one or more) of *dr_test* calls whose input request variable was initialized by a *msg_recv_i* call and dose retrieve a message from the
runtime buffers. It is defined as tuple $ArrivalTest \in ReqInitOrder \times Count \times Data$, such that $ReqInitOrder$ is the initialization order of the input request variable at the current node and $Count$ is the number of times the $dr\_test$ call had failed, before succeeding and retrieving a message.

Similarly, $NonArrivalTest \in ReqInitOrder \times Count$ record stems from a sequence of $dr\_test$ calls. However, the $NonArrivalTest$ record indicates that no messages were retrieved from the runtime buffers. That occurs when the input request variable was initialized by a non-blocking function other than $dr\_msg\_recv\_i$ or when the input request variable was initialized by a $dr\_msg\_recv\_i$ call and the sequence of $dr\_test$ calls doesn’t retrieve a message from the runtime buffers.

A Rand record represents a single invocation of the rand function and is defined as:

$Rand \in RandOrder \times Value$. $RandOrder$ is the invocation order of this particular rand call among other rand calls at this node. Value is the random number returned by the rand call.

Table 4.1 shows a trace of the example program in Figure 4.5:
Table 4.1. Data-replay trace of the program in Figure 4.5

<table>
<thead>
<tr>
<th>Node 2</th>
<th>Node 4</th>
<th>Node 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:1/0/D1</td>
<td>R:1/1/D1</td>
<td>R:1/1/D1</td>
</tr>
<tr>
<td>W:2/D2</td>
<td>T:1/10/D2</td>
<td>R:1/2/D2</td>
</tr>
</tbody>
</table>

Since nodes 1 and 3 don't receive any messages, they don't produce traces. The trace record (A:1/0/D1) is described as following: ‘A’ indicates a RecvWany record. Number 1 is the invocation order of the wait_any call that generated RecvWany record. Number 0 is the value that wait_any returned. D1 signifies the payload of the retrieved message.

The fields of the trace record (W:2/D1) are described as following: ‘W’ indicates a Wait record. Number 2 is the initialization order of the dr_wait input request variable. D2 signifies the payload of the retrieved message.

The fields of the trace record (R:1/1/D1) are described as following: ‘R’ indicates a Recv record. The first number 1 is the port number of the receiving endpoint. The second number 1 is the invocation order of the dr_msg_recv call that generated this Recv record. D1 signifies the payload of the received message.

The fields of the trace record (T:1/10/D2) are described as following: ‘T’ indicates an ArrivalTest record. Number 1 is the initialization order of the dr_test input request variable. Number 10 is the number of failed dr_test calls. D2 signifies the payload of retrieved message.
4.3.2. The replay mechanism

We now describe how a trace is used to replay an execution. To achieve a correct replay of a program, it is necessary to associate endpoints, request variables and certain calls that were observed during the recording mode with their counterparts in the replay mode. An endpoint that is observed in the replay mode is associated with an endpoint that is observed in the recording mode via the node identifier and the port number; both remain the same across executions. Request variables are tracked across an execution in the recording mode and an execution in the replay mode using their order of initialization in a node. Similarly, `dr_msg_recv` and `dr_wait_any` calls are tracked by their invocation order with respect to other `dr_msg_recv` and `dr_wait_any` calls, respectively, in the same node.

During replay, DR-MCAPI maintains two data structures for each node:

1) **Records**: a list of trace records (i.e. `Recv`, `Wait`, `RecvTest`, `RecvWany`, `ArrivalTest`, `NonArrivalTest` and `NonRecvWany`). This list is constructed directly from the trace.

2) **RequestVariables**: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly. When a request variable is initialized (by being passed to a non-blocking call), a new item is appended to this list. If the request variable was initialized by a non-blocking receive call, then we keep track of the receiving endpoint and the destination buffer pointer. If the trace indicates that `dr_test` calls were used to check the status of this request in the
record mode, then the number of failed tests is retrieved from the trace and associated with that request. All newly initialized requests are flagged as incomplete.

The MCAPI routine calls that introduce non-determinism (as mentioned in section 2.3) are handled by the DR-MCAPI library, rather than the MCAPI library, as follows:

The algorithm in Figure 4.6 shows how `dr_msg_recv` calls are handled. First, `RecvCalls` is incremented. `RecvCalls` keeps track of the number of `dr_msg_recv` function invocations at the node. Second, the `GetRecvRecord` procedure looks up the `Records` list to fetch the `Recv` record with `RecvOrder=RecvCalls`. Finally, the message payload in the `Recv` record is copied to the program buffer (lines 4-5).

```c
dr_msg_recv(Endpoint, &Buffer){
    1 RecvCalls++;
    2 PortNum=GetPortNumber(Endpoint);
    3 RecvRecord=GetRecvRecord(RecvCall);
    4 Data=RecvRecord.Data;
    5 copy(Buffer,&Data);
    6 return;
}

Figure 4.6. Handling `dr_msg_recv` calls
```

The algorithm in Figure 4.7 shows how DR-MCAPI handles `dr_wait` calls. If the input request was not initialized by a `dr_msg_recv_i` call, then it is forwarded to the MCAPI library (lines 1-3). Otherwise, the initialization order and a pointer to the program buffer of this request are retrieved (lines 4-5). Next, the `GetWaitRecord` procedure looks up the `Records` list to fetch the `Wait` record with `ReqInitOrder=Order`. Finally, the message payload in the `Wait` record is copied to the program buffer (lines 7-8).
dr_wait(Request) {
1   if not IsRecvRequest(Request) then
2       return mcapi_wait(Request);
3   end-if
4   Order=GetOrder(Request);
5   DataPtr=GetDataPtr(Request);
6   WaitRecord=GetWaitRecord(Order);
7   Data=WaitRecord.Data;
8   copy(DataPtr,&Data);
9   return;
}

Figure 4.7. Handling dr_wait calls

Figure 4.8 describes how DR-MCAPI handles dr_wait_any calls. First, WaitanyCalls is incremented (line 1). WaitanyCalls keeps track of the number of dr_wait_any function invocations at the node. If the current dr_wait_any call doesn’t retrieve a message, then the GetNonRecvWaitanyRecord procedure looks up the Records list to fetch the NonRecvWany record with WanyOrder=WaitanyCalls (line 3). In line 4, the Index in the NonRecvWany is retrieved and the request in the Requests array at Index will be forwarded to the MCAPI library (line 5). Finally, Index is returned to the program (line 6).

dr_wait_any(Requests){
1   WaitanyCalls++;  
2   if not RecvWany(WaitanyCalls) then
3       NonRecvWanyRecord=GetNonRecvWanyRecord(WaitanyCalls);
4       Index=NonRecvWanyRecord.Index;
5       mcapi_wait(Requests[Index]);
6       return Index;
7   else
8       RecvWanyRecord=GetRecvWanyRecord(WaitanyCalls);
9       Index=RecvWanyRecord.Index;
10      DataPtr=GetDataPtr(Requests[Index]);
11      Data=RecvWanyRecord.Data;
12      copy(DataPtr,&Data);
13      return Index;
14   end-if
}

Figure 4.8. Handling dr_wait_any calls
If the current `dr_want_any` call does retrieve a message, then the `GetRecvWaitAnyRecord` procedure looks up the `Records` list to fetch the `RecvWany` record with `WanyOrder=waitanyCalls` (line 8). In line 9, the `Index` in the `RecvWany` is used to retrieve the program data pointer associated with the request in the `Requests` array at `Index`. Finally, the message data in the `RecvWany` record is copied to the program buffer (lines 11-12) and `Index` is returned to the program (line 13).

A `dr_test` call is handled by the algorithm in Figure 4.9. First, the initialization order of the input request variable (Request) is retrieved (line 1). If that request variable is associated with a `NonArrivalTest` record, then the `Count` of this record is reduced by one (line 4). If `Count` reaches zero, the request is forwarded to the MCAPI runtime and true is returned to the program (lines 8-9). If that request variable is associated with a `ArrivalTest` record, then the `Count` of this record is reduced by one (line 13). If `Count` reaches zero, the request is passed to `dr_wait` and true is returned to the program (lines 17-18).
bool dr_test(Request) {
    Order = GetOrder(Request);
    if not ArrivalTest(Order) then
        NonArrivalTest = GetNonArrivalTest(Order);
        NonArrivalTest.Count --;
        if NonArrivalTest.Count > 0 then
            return false;
        else
            mcapi_wait(Request);
            return true;
        end-if
    else
        ArrivalTest = GetArrivalTest(Order);
        ArrivalTest.Count --;
        if ArrivalTest.Count > 0 then
            return false;
        else
            dr_wait(Request);
            return true;
        end-if
    end-if
}

Figure 4.9. Handling dr_test calls

4.4. DR-MCAPI order-replay

In order-replay, the outcomes of non-deterministic operations are recorded during a recording execution and are enforced during the replay execution. All nodes must be running during replay. Since in order-replay only the outcomes of non-deterministic operations are recorded, far less data than data-replay tools is recorded. We have developed two techniques for realizing order-replay: sender-based order-replay (described in section 4.4.1) and receiver-based order-replay (described in section 4.4.2).

4.4.1. Sender-based order-replay

Sender-based order-replay works by capturing the total order of messages arrival during the recorded execution and enforcing this order during the replay execution by changing the order of dispatching mcapi_msg_send (and mcapi_msg_send_i) calls to the MCAPI
runtime. First, we describe the sender-based order-replay trace structure in section 4.4.1.1 and then its replay mechanism in section 4.4.1.2.

4.4.1.1. The trace structure

In the sender-based order-replay technique, a single trace is generated for the whole program. $\text{Trace} = \{\text{Record}_1, ..., \text{Record}_{|\text{Trace}|}\}$ and there are three record types in the trace:

$\text{Record} \in \text{Send} \cup \text{Wany} \cup \text{Test} \cup \text{Rand}$

A $\text{Send}$ record represents sending a message between two endpoints and is defined as tuple: $\text{Send} \in \text{Node} \times \text{Port} \times \text{SendOrder} \times \text{UAO}$. $\text{Node}$ is the identifier of the sending node. $\text{Port}$ is the port number of the sending endpoint. $\text{SendOrder}$ is the invocation order of the particular $\text{dr}_\text{msg}_\text{send}$ (or $\text{dr}_\text{msg}_\text{send_i}$) call among other $\text{dr}_\text{msg}_\text{send}$ (and $\text{dr}_\text{msg}_\text{send_i}$) calls at the same node. $\text{UAO}$ stands for Unique Arrival Order which is a global number assigned to every received message and it establishes a total order of arrivals among all received messages in a program. A $\text{Send}$ record is constructed in two steps:

1) When a $\text{dr}_\text{msg}_\text{send}$ (or a $\text{dr}_\text{msg}_\text{send_i}$) is invoked, the message payload is augmented with the triple $\{\text{Node}, \text{Port}, \text{SendOrder}\}$.

2) When a message is retrieved from the runtime buffers (by $\text{dr}_\text{msg}_\text{recv}$, $\text{dr}_\text{wait}$, $\text{dr}_\text{wait_any}$ or $\text{dr}_\text{test}$ call), it is assigned the $\text{UAO}$ number. $\text{UAO}$ is monotonically increasing with every received message throughout the program.
A *Wany* record stems from a `dr_wait_any` call and is defined as \( Wany \in Node \times Order \times Index \). *Node* is the identifier of the current node. *Order* is the invocation order of this particular `dr_wait_any` call among other `dr_wait_any` calls at the same node. *Index* is the index returned by the `dr_wait_any` call.

A *Test* record originates from a sequence (one or more) of `dr_test` calls and is defined as \( Test \in Node \times ReqInitOrder \times Count \), such that *Node* is the identifier of the current node, *ReqInitOrder* is the initialization order of the input request variable at the current node and *Count* is the number of times the `dr_test` call had failed. Table 4.2 shows a trace of the example program in Figure 4.5.

**Table 4.2. Sender-based order-replay trace of the program in Figure 4.5**

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S:1/1/1/1</td>
<td>A:2/1/0</td>
<td>S:3/1/1/2</td>
<td>T:4/1/10</td>
</tr>
<tr>
<td>S:1/1/2/3</td>
<td>S:2/1/1/5</td>
<td>S:3/1/2/4</td>
<td></td>
</tr>
<tr>
<td>S:1/1/3/7</td>
<td>S:2/1/2/6</td>
<td>S:3/1/3/8</td>
<td></td>
</tr>
</tbody>
</table>

Since node 5 neither sends messages, nor has `dr_test` or `dr_wait_any` calls, it doesn't contribute to the trace. The trace record (S:1/1/1/1) was generated by the `dr_msg_send` call in line 7 and the `dr_wait_any` call at line 29. Its fields are described as follows: ‘S’ indicates a *Send* record. The first number 1 is the node identifier. The second number 1 is the port number of the sending endpoint. The third number 1 is the invocation order of the `dr_msg_send` call. The fourth number 1 is the UAO which indicates that the message sent by this `dr_msg_send` call was the first to be received in the recorded execution.
The trace record (A:2/1/0) was generated by the `dr_wait_any` call at line 29. Its fields are described as follows: ‘A’ indicates a \textit{Wany} record. Number 2 is the node identifier. Number 1 is the invocation order of this `dr_wait_any` call. Number 0 is the value returned by the `dr_wait_any` call.

The trace record (T:4/1/10) was generated by the `dr_test` call in line 43 and its fields are described as follows: ‘T’ indicates a \textit{Test} record. Number 4 is the node identifier. Number 1 is the `dr_test` call input request variable initialization order. Number 10 is the number of times the `dr_test` call was invoked.

A \textit{Rand} record represents a single invocation of the rand function and is defined as: \textit{Rand} \in \textit{RandOrder} \times \textit{Value}. \textit{RandOrder} is the invocation order of this particular rand call among other rand calls at this node. \textit{Value} is the random number returned by the rand call.

\begin{itemize}
\item [4.4.1.2.] \textbf{The replay mechanism}
\end{itemize}

We now describe how a trace is used to replay an execution. When a program is run in the replay mode, four data structures are created:

1) \textit{SendRecords}: a list of all \textit{Send} records from the trace.

2) \textit{RequestVariables}: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly.

3) \textit{WanyRecords}: a list of \textit{Wany} records that are obtained from the trace

4) \textit{TestRecords}: a list of \textit{Test} records that are retrieved from the trace.
In the replay mode, sending a message is a three step process as depicted in Figure 4.10:

1) When a `dr_msg_send` (or a `dr_msg_send_i`) is invoked by the program, the corresponding `Send` record in the `SendRecords` list is set to `Posted`.

2) The algorithm `SendCallsScheduler` in Figure 4.11 continuously monitors the `SendRecords` list. If `SendCallsScheduler` finds a `Send` record whose state is `Posted` and whose `UAO` equals to `LatestUAO`, this record state is set to `Pending` and a corresponding `mcapi_msg_send` (or `mcapi_msg_send_i`) is invoked to actually send a message.

3) When a message is received (via a call to `dr_msg_recv`, `dr_wait`, `dr_test`, or a `dr_wait_any`), its state is set to `Delivered`.

Figure 4.10. The three steps of sending a message
SendCallsScheduler()
1 LatestUAO=0;
2 Max=Size(SendRecords);
3 while(LatestUAO<Max) do
4   if (there is no pending send calls) then
5     S = SendRecords.GetRecord(UAO);
6     if (S is posted)
7       LatestUAO++;
8     Set S state to pending;
9     Forward S to the MCAPI runtime;
10   end-if
11 end-if
12 end-while
}

Figure 4.11. The SendCallsScheduler algorithm

Figure 4.12 shows how dr_wait_any calls are handled. First, WaitanyCalls is incremented. WaitanyCalls keeps track of the number of dr_wait_any function invocations at the node. Second, the GetWaitanyRecord procedure looks up the WanyRecords list to fetch the Wany record with Order=waitanyCalls (line 2). In line 3, the Index in the Wany is retrieved and the request in the Requests array at Index will be forwarded to the MCAPI library (line 4). Finally, Index is returned to the program.

dr_wait_any(Requests){
1   WaitanyCalls++;
2   WanyRecord=GetWaitanyRecord(WaitanyCalls);
3   Index=WanyRecord.Index;
4   mcapi_wait(Requests[Index]);
5   return Index;
}

Figure 4.12. Handling dr_wait_any calls

A dr_test call is handled by the algorithm in Figure 4.13. First, the initialization order of the input request variable (Request) is retrieved (line 1). Then, the Test record associated with this initialization order is retrieved from the TestRecords list (line 2). Next, the Count of this record is reduced by one (line 3). When Count reaches zero, the request is passed to mcapi_wait and true is returned to the program (lines 7-8).
```cpp
bool dr_test(Request){
  Order=GetOrder(Request);
  Test=GetTest(Order);
  Test.Count--;
  if Test.Count>0 then
    return false;
  else
    mcapi_wait(Request);
    return true;
  end-if
}
```

Figure 4.13. Handling dr_test calls

We use Table 4.3 and Table 4.4 to illustrate how the sender-based order-replay works.
Table 4.3 shows a list of the `dr_msg_send` calls that appear in the example program and has three columns. The first column shows the line numbers of the `dr_msg_send` calls. Column 2 assigns names to the `dr_msg_send` calls. We use these names for brevity. The third column lists the UAO associated with the `dr_msg_send` calls according to the trace in Table 4.2.

Table 4.3. Sender-based order-replay example

<table>
<thead>
<tr>
<th>Line</th>
<th>Name</th>
<th>UAO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>S2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>S3</td>
<td>7</td>
</tr>
<tr>
<td>18</td>
<td>S4</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>S5</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>S6</td>
<td>8</td>
</tr>
<tr>
<td>34</td>
<td>S7</td>
<td>5</td>
</tr>
<tr>
<td>35</td>
<td>S8</td>
<td>6</td>
</tr>
</tbody>
</table>

Now, let's assume that the `dr_msg_send` calls are invoked according to this order: S4, S5, S1, S2, S3, S6, S7, and finally S8. Table 4.4 describes how the DR-MCAPI library will
handle the `dr_msg_send` calls such that the messages order of arrival that was observed in
the recording mode will be exhibited during the replay mode.

Table 4.4. Handling `dr_msg_send` example

<table>
<thead>
<tr>
<th>Program Event</th>
<th>DR-MCAPI Library Action</th>
</tr>
</thead>
</table>
| S4 is invoked | 1) S4 `Send` record is set to `Posted`.  
2) S4 will be blocked by the `SendCallsScheduler` procedure, since there are other `dr_msg_send` calls with smaller UAO (S1) that were not delivered yet. |
| S5 is invoked | 1) S5 `Send` record is set to `Posted`.  
2) S5 will be blocked by the `SendCallsScheduler` procedure, since there are other `dr_msg_send` calls with smaller UAO (S1, S2, and S4) that were not delivered yet. |
| S1 is invoked | 1) S1 `Send` record is set to `Pending`.  
2) S1 message is forwarded to the MCAPI runtime. |
| S1 message is received | 1) S1 `Send` record is set to `Delivered`.  
2) S4 `Send` record is set to `Pending`.  
3) S4 message is forwarded to the MCAPI runtime. |
| S4 message is received | 1) S4 `Send` record is set to `Delivered`.  
2) S5 is still blocked since S2 (which has a smaller UAO) is not delivered yet. |
| S2 is invoked | 1) S2 `Send` record is set to `Posted`.  
2) S2 message is forwarded to the MCAPI runtime. |
| S2 message is received | 1) S2 `Send` record is set to `Delivered`.  
2) S5 `Send` record is set to `Pending`.  
3) S5 message is forwarded to the MCAPI runtime. |
| S5 message is received | 1) S5 `Send` record is set to `Delivered`. |
| S3 is invoked | 1) S3 `Send` record is set to `Posted`.  
2) S3 will be blocked by the `SendCallsScheduler` procedure, since there are other `dr_msg_send` calls with smaller UAO (S7 and S8) that were not delivered yet. |
| S6 is invoked | 1) S6 `Send` record is set to `Posted`.  
2) S6 will be blocked by the `SendCallsScheduler` procedure, since there are other `dr_msg_send` calls with smaller UAO (S3 and S8) that were not delivered yet. |
| S7 is invoked | 1) S7 `Send` record is set to `Pending`.  
2) S7 message is forwarded to the MCAPI runtime. |
| S7 message is received | 1) S7 `Send` record is set to `Delivered`. |
Table 4.4. – continued

<table>
<thead>
<tr>
<th>Event Description</th>
<th>Actions</th>
</tr>
</thead>
</table>
| S8 is invoked     | 1) S8 Send record is set to *Posted*.  
|                   | 2) S8 message is forwarded to the MCAPI runtime. |
| S8 message is received | 1) S8 Send record is set to *Delivered*.  
|                   | 2) S8 Send record is set to *Pending*.  
|                   | 3) S8 message is forwarded to the MCAPI runtime. |
| S3 message is received | 1) S3 Send record is set to *Delivered*.  
|                   | 2) S6 Send record is set to *Pending*.  
|                   | 3) S6 message is forwarded to the MCAPI runtime. |
| S6 message is received | 1) S6 Send record is set to *Delivered*. |

Table 4.4 shows that regardless to the order of *dr_msg_send* invocations observed in the replay executions, messages will be delivered according to the order observed in the recorded execution.

### 4.4.2. Receiver-based order-replay

Receiver-based order-replay works by capturing the order of messages arrival at a specific node during the recording phase and enforcing this order during the replay phase by manipulating the order of the messages retrieved from the runtime buffers. The order of messages arrival is established using a hash-code of the messages payload. First, we describe the receiver-based order-replay trace structure in section 4.4.2.1 and then its replay mechanism in section 4.4.2.2.

#### 4.4.2.1. The trace structure

When an instrumented program $P$ is run in the record mode, a separate trace is generated for each MCAPI node:

$$\text{Trace}_P = \{\text{Trace}^1, ..., \text{Trace}^N\},$$

where $N$ is the number of nodes in program $P$.  

---

72
A node's trace contains a list of records:

\[ \text{Trace}^n = \{\text{Record}^1, \ldots, \text{Record}^{\text{Trace}^n}\}, \text{where } n \text{ is a node identifier.} \]

There are six types of records:

\[ \text{Record} \in \text{Recv} \cup \text{Wait} \cup \text{RecvWany} \cup \text{NonRecvWany} \cup \text{ArrivalTest} \]
\[ \cup \text{NonArrivalTest} \cup \text{Rand} \]

A \textit{Recv} record originates from a \texttt{msg_recv} call and is defined as tuple \texttt{Recv} \( \in \text{Port} \times \text{RecvOrder} \times \text{Hash} \). \texttt{Port} is the port number of the receiving endpoint. \texttt{RecvOrder} is the invocation order of this particular \texttt{msg_recv} call among other \texttt{msg_recv} calls with the same endpoint. \texttt{Hash} is a hash-code of the received message data and is calculated using the CRC-32 algorithm [50].

A \textit{Wait} record originates from a \texttt{wait} call whose input request variable was initialized by a \texttt{msg_recv_i} call and is defined as \texttt{Wait} \( \in \text{ReqInitOrder} \times \text{Hash} \). \texttt{ReqInitOrder} is the initialization order of the input request variable at the current node.

A \textit{RecvWany} record comes from a \texttt{wait_any} call that returned the index of a request variable that was initialized by a \texttt{msg_recv_i} call and is defined as: \texttt{RecvWany} \( \in \text{WanyOrder} \times \text{Index} \times \text{Hash} \). \texttt{WanyOrder} is the invocation order of this particular \texttt{wait_any} call among other \texttt{wait_any} calls at this node. \texttt{Index} is the index returned by the \texttt{wait_any} call. \texttt{Hash} is a hash-code of the received message data.
The record \textit{NonRecvWany} is defined as: \textit{NonRecvWany} $\in WanyOrder \times Index$ and indicates that a \texttt{wait\_any} call returned the index of a request variable that was initialized by a non-blocking function other than \texttt{msg\_recv\_i}.

An \textit{ArrivalTest} record originates from a sequence (one or more) of \texttt{test} calls whose input request variable was initialized by a \texttt{msg\_recv\_i} call and does retrieve a message from the runtime buffers. It is defined as \textit{ArrivalTest} $\in ReqInitOrder \times Count \times Hash$, such that \textit{ReqInitOrder} is the initialization order of the input request variable at the current node, \textit{Hash} is a hash-code of the received message data and \textit{Count} is the number of times the \texttt{test} call had failed, before succeeding and retrieving a message.

Similarly, \textit{NonArrivalTest} record stems from a sequence of \texttt{test} calls. However, the \textit{NonArrivalTest} record indicates that no messages were retrieved from the runtime buffers. That occurs when the input request variable was initialized by a non-blocking function other than \texttt{msg\_recv\_i} or when the input request variable was initialized by a \texttt{msg\_recv\_i} call and the sequence of \texttt{test} calls doesn’t retrieve a message from the runtime buffers.

Table 4.5 shows a trace of the example program in Figure 4.5:
Since nodes 1 and 3 don’t receive any messages, they don’t produce traces. The trace record \((A:1/0/C1)\) is described as following: ‘A’ indicates a \textit{RecvWany} record. Number 1 is the invocation order of the \texttt{wait\_any} call that generated \textit{RecvWany} record. Number 0 is the value that \texttt{wait\_any} returned. \texttt{C1} signifies the hash-code of the payload of the retrieved message.

The fields of the trace record \((W:2/C1)\) are described as following: ‘W’ indicates a \textit{Wait} record. Number 2 is the initialization order of the \texttt{dr\_wait} input request variable. \texttt{C2} signifies the hash-code of the payload of the retrieved message.

The fields of the trace record \((R:1/1/C1)\) are described as following: ‘R’ indicates a \textit{Recv} record. The first number 1 is the port number of the receiving endpoint. The second number 1 is the invocation order of the \texttt{dr\_msg\_recv} call that generated this \textit{Recv} record. \texttt{C1} signifies the hash-code of the payload of the retrieved message.

The fields of the trace record \((T:1/10/C2)\) are described as following: ‘T’ indicates an \textit{ArrivalTest} record. Number 1 is the initialization order of the \texttt{dr\_test} input request
variable. Number 10 is the number of failed \texttt{dr_test} calls. C2 signifies the hash-code of the payload of the retrieved message.

A Rand record represents a single invocation of the rand function and is defined as: Rand $\in$ RandOrder $\times$ Value. RandOrder is the invocation order of this particular rand call among other rand calls at this node. Value is the random number returned by the rand call.

4.4.2.2. The replay mechanism

We now describe how a trace is used to replay an execution. To support the replay mode, we maintain three data structures:

1) \textit{Records}: a list of records (e.g. \texttt{Recv, Wait...}) that are retrieved from the trace.

2) \textit{RequestVariables}: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly.

3) \textit{RecievedMessages}: messages that arrive earlier than expected are stored in this list along with their hash-codes.

The algorithm in Figure 4.14 handles \texttt{dr_msg_recv} calls. First \texttt{RecvCalls} is incremented (line 1). \texttt{RecvCalls} keeps track of the number of \texttt{dr_msg_recv} function invocations at the node. Second, the hash-code of the expected message is retrieved (line 1). Second, the \texttt{GetRecvRecord} procedure looks up the \textit{Records} list to fetch the \texttt{Recv} record with \texttt{RecvOrder}=\texttt{RecvCalls} (line 2). Next, \textit{RecievedMessages} is looked up for a message whose hash-code matches the expected hash-code. If such a message is found, then its data is copied to the program buffer (line 7). Otherwise, the \texttt{mcapi_msg_recv} is repeatedly
invoked till it retrieves a message whose hash-code matches the expected hash-code (lines 10-20). When the excepted message arrives, it is copied to the program buffer (line 15). All other messages and their hash-codes are appended to RecievedMessages (line 19).

```c
dr_msg_recv(Endpoint, &Buffer) {
  1  RecvCalls++;
  2  RecvRecord=GetRecvRecord(RecvCalls);
  3  ExpectedCRC=RecvRecord.Hash;
  4  for Index=0 to RecievedMessages.size do
  5      if (RecievedMessages[Index].CRC==ExpectedCRC) then
  6          copy(Buffer, RecievedMessages[Index]);
  7          return;
  8      end-if
  9  end-while
 10  while(true) do
 11     mcapi_msg_recv(Endpoint,&TempBuffer);
 12     ArrivedCRC=CalculateCRC(TempBuffer);
 13     if (ArrivedCRC==ExpectedCRC) then
 14        copy(Buffer, TempBuffer);
 15        return;
 16     end-if
 17  else
 18     RecievedMessages.Append(TempBuffer, ArrivedCRC);
 19  end-while
}
```

Figure 4.14. Handling dr_msg_recv calls

In the program in Figure 4.5, node 5 receives two messages. Let's assume that when running that program in the record mode, it generates the trace in the Table 4.5 (i.e. the order of messages arrival is C1, C2, and then C3) and that during running the program in the replay mode, the messages arrive with a different order: C2, C3, and then C1. When dr_msg_recv is invoked for the first time, the RecievedMessages list will be empty. Hence, the while loop (lines 10-20) will iterate thrice. In the first iteration, the mcapi_msg_recv call will retrieve the message with hash-code C2. Since the retrieved message is not the excepted one, it will be added to the RecievedMessages list (line 19).
In the second iteration, the `mcapi_msg_recv` call will retrieve the message with hash-code C3 and, it will be added to the `RecieveedMessages` list as well. In the third iteration, the message with hash-code C1 will be retrieved. So, this message will be delivered to the program (line 15). When `dr_msg_recv` is invoked for the second and third times, the `RecievedMessages` list will contain the expected messages and they will be returned to the program in the correct order (lines 4-9).

Figure 4.15 shows the algorithm that handles a `dr_wait` call whose input request variable was initialized by a `msg_recv_i` call. This algorithm depends on the `RequestVariables` list that links a request variable with the endpoint and the program buffer pointer that were passed to the `msg_recv_i` call.
```c
void dr_wait(Request) {
    if not IsRecvRequest(Request) then
        return mcapi_wait(Request);
    end-if
    InitOrder=GetInitOrder(Request);
    WaitRecord=GetWaitRecord(InitOrder);
    ExpectedCRC=WaitRecord.Hash;
    BufferPtr=GetBufferPtr(Request);
    Endpoint=GetEndpoint(Request);
    Requests=GetRequests(CurrentNode);
    for Index=0 to Requests.size() do
        if (Requests[Index].isComplete) then continue;
        mcapi_wait(Requests[Index]);
        ArrivedData=GetData(Requests[Index]);
        ArrivedCRC=CalculateCRC(ArrivedData);
        RecievedMessages.Append(ArrivedData,ArrivedCRC);
        Requests[Index].setComplete();
    end-for
    for Index=0 to RecievedData.size() do
        if (RecievedMessages[Index].CRC==ExpectedCRC) then
            copy(BufferPtr, RecievedMessages[Index]);
            return;
        end-if
    end-for
}
```

**Figure 4.15. Handling dr_wait calls**

First, if the input request was not initialized by a `dr_msg_recv_1` call, then it is forwarded to the MCAPI library (lines 1-3). Otherwise, the hash-code of the expected message, the endpoint and the program buffer pointer associated with the input request variable are retrieved (lines 4-8). Second, `mcapi_wait` is invoked for all initialized (but not completed) requests at that node and retrieved messages and their hash-codes are appended to `RecievedMessages` (lines 9-17). Finally, `RecievedMessages` is looked up for a message whose hash-code matches the expected hash-code. When such message is found, it is copied to the buffer associated with the input request variable (line 21).
Figure 4.16 describes how DR-MCAPI handles dr_wait_any calls. First, \( \text{WaitanyCalls} \) is incremented (line 1). \( \text{WaitanyCalls} \) keeps track of the number of \( \text{dr_wait_any} \) function invocations at the node. If the current \( \text{dr_wait_any} \) call retrieves a message, then the \( \text{GetRecvWanyRecord} \) procedure looks up the \( \text{Records} \) list to fetch the \( \text{RecvWany} \) record with \( \text{WanyOrder} = \text{WaitanyCalls} \) (line 3). In line 4, the \( \text{Index} \) in the \( \text{RecvWany} \) record is retrieved and the request in the \( \text{Requests} \) array at \( \text{Index} \) will be forwarded to \( \text{dr_wait} \) (line 5). If the current \( \text{dr_wait_any} \) call doesn’t retrieve a message, then the \( \text{GetNRecvWanyRecord} \) procedure looks up the \( \text{Records} \) list to fetch the \( \text{NonRecvWany} \) record with \( \text{WanyOrder} = \text{WaitanyCalls} \) (line 7). In line 8, the \( \text{Index} \) in the \( \text{NonRecvWany} \) record is retrieved and the request in the \( \text{Requests} \) array at \( \text{Index} \) will be forwarded to \( \text{mcapi_wait} \) (line 9). Finally, \( \text{Index} \) is returned to the program (line 11).

\[
\text{dr_wait_any}(\text{Requests}) \{ \\
1 \quad \text{WaitanyCalls}++; \\
2 \quad \text{if RecvWany(WaitanyCalls) then} \\
3 \quad \quad \text{RecvWanyRecrd}=\text{GetRecvWanyRecord(WaitanyCalls)}; \\
4 \quad \quad \text{Index}=\text{RecvWanyRecrd.Index}; \\
5 \quad \quad \text{dr_wait}(\text{Requests}[\text{Index}]); \\
6 \quad \text{else} \\
7 \quad \quad \text{NRecvWanyRecrd}=\text{GetNRecvWanyRecord(WaitanyCalls)}; \\
8 \quad \quad \text{Index}=\text{NRecvWanyRecrd.Index}; \\
9 \quad \quad \text{mcapi_wait}(\text{Requests}[\text{Index}]); \\
10 \quad \text{end-if} \\
11 \quad \text{return Index}; \\
\}
\]

**Figure 4.16. Handling dr_wait_any calls**

In the program in Figure 4.5, node 2 receives two messages. Let’s assume that when running that program in the record mode, it generates the trace in the Table 4.5 (i.e. the order of messages arrival is C1 then C2 and that \( \text{wait_any} \) call returns 0) and that during running the program in the replay mode, the messages arrive with a different order: C2

80
then C1. When dr_wait_any is invoked, it is going to determine that the request at index 0 of the array Requests was initialized by a dr_msg_recv call and will forward this request to dr_wait. In dr_wait, the first loop (lines 10-17) will retrieve the two messages via three calls to mcapi_wait (line 12) and they will be added to the ReceivedMessages list (line 15). The second loop (lines 18-24) will iterate through the ReceivedMessages list and will return the message with hash-code C1 to the program. When dr_wait is invoked to handle the wait call at line 30 in Figure 10, the message with hash-code C2 will be already in the ReceivedMessages list and will be returned to the program.

A dr_test call is handled by the algorithm in Figure 4.17. First, the initialization order of the input request variable (Request) is retrieved (line 1). If that request variable is associated with an ArrivalTest record, then the Count of this record is reduced by one (line 4). If Count reaches zero, the request is forwarded to dr_wait and true is returned to the program (lines 8-9). If that request variable is associated with a NonArrivalTest record, then the Count of this record is reduced by one (line 13). If Count reaches zero, the request is passed to mcapi_wait and true is returned to the program (lines 17-18).
bool dr_test(Request) {
    1 InitOrder = GetInitOrder(Request);
    2 if ArrivalTest(InitOrder) then
    3     ArrivalTestRecord = GetArrivalTestRecord(Order);
    4     ArrivalTestRecord.Count --;
    5     if ArrivalTestRecord.Count > 0 then
    6         return false;
    7     else
    8         dr_wait(Request);
    9         return true;
    10    end-if
    11 else
    12     NAarrivalTestRecord = GetNAarrivalTestRecord(Order);
    13     NAarrivalTestRecord.Count --;
    14     if NAarrivalTestRecord.Count > 0 then
    15         return false;
    16     else
    17         mcapi_wait(Request);
    18         return true;
    19     end-if
    20    end-if
}

Figure 4.17. Handling dr_test calls

In the program in Figure 4.5, node 4 receives three messages. Let's assume that when run in the record mode, this program generates the trace in Table 4.5 (i.e. three messages are retrieved with order: C1, C2, and then C3 and that the dr_test call at line 43 retrieves the messages with hash-code C2 at the 11th invocation). Let's assume that during replay, the messages arrive with a different order (C2, C1, and then C3). When the dr_msg_recv call at line 42 is invoked, messages with hash-codes C1 and C2 will be retrieved from the runtime buffers and C1 will be returned to the program. When the dr_test call at line 43 is invoked, it will return false for 10 times and at the 11th invocation, it will invoke dr_wait. dr_wait will find the message with hash-code C2 in the ReceivedMessages list. When the dr_wait call at line 44 is invoked, it will retrieve the message with hash-code C3.
4.5. Results

In this section we analyze the performance of the replay approaches in terms of trace size, memory usage and runtime overheads.

4.5.1. Methodology

We performed experiments on three sets of MCAPI programs developed by ourselves and a set of programs obtained from an external source [51]. Our experiments were conducted on a machine with Core 2 Duo 1.4 GHz CPU and 4 GB RAM using MCAPI runtime V1.063. We evaluate DR-MCAPI using the following set of programs:

4) Binary Tree benchmark (BT): This is a set of 10 programs that create networks of nodes with sizes from 3 nodes to 21 nodes. Each two nodes send a message to the same parent node forming a binary tree in which messages travel from the leaves to the root node. The smallest tree has 3 nodes and exchanges 20 messages. The largest one has 21 nodes and exchanges 155 messages. This benchmark has a master/slave communication pattern.

5) Complete Graph benchmark (CG): This is a set of 10 programs that create networks of nodes with increasing sizes from 2 nodes to 11 nodes. All nodes send and receive messages to/from each other forming a complete graph. The number of exchanged messages is between 20 message (for a 2 nodes graph) and 1100 messages (for a 11 nodes graph). This benchmark has an all-to-all communication pattern.

6) 10-nodes benchmark (TN): This is a set of 10 programs that create networks of nodes with a fixed size of 10 nodes. However, the number of messages exchanged
among the nodes increases monotonically. The number of messages exchanged is between 90 and 900. This benchmark allows us to isolate the effect of the number of messages on performance.

7) Bully benchmark (Bully): This is a set of 10 programs that create networks of nodes with different sizes and use the Bully leader selection algorithm [52] to select a leader node. The number of exchanged messages is between 35 messages (for a 3 nodes network) and 314 messages (for a 12 nodes network). This benchmark was provided by the V&V research group at Brigham Young University.

In all benchmarks, except the Bully benchmark, the message size is 50 bytes. The Bully benchmark message size is 4 bytes.

To analyze the runtime and memory usage, a given program is executed three times: 1) without the DR-Library, 2) with the DR-MCAPI library in recording mode and 3) with the DR-MCAPI library in replay mode. We use a pair of gettimeofday function calls; when a program starts execution and when it ends execution to calculate total runtime and use the Massif [53] heap profiler to measure the heap memory used by a given execution.

For the sake for brevity, we refer to data-replay as D-replay, sender-based order-replay as S-replay, and receiver-based order-replay as R-replay.
4.5.2. Log size

Our first analysis is for the trace size. Figure 4.18 shows the trace size relative to the number of exchanged messages using the D-replay, S-replay and R-replay techniques. The x-axis is the number of messages and the y-axis is the trace size in kilobytes.

D-replay produces a large trace compared to R-replay and S-replay in the benchmarks BT, CG, and TN. However, it is the opposite with the Bully benchmark. This is due to the small size of the messages exchanged in the Bully benchmark (4 bytes) compared to the other benchmarks (50 bytes). Table 4.6 shows the typical record size in the three replay techniques.
Table 4.6. Records structures and sizes in D-replay, S-replay and R-replay

<table>
<thead>
<tr>
<th>Technique</th>
<th>Typical Record Structure</th>
<th>Record Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-replay</td>
<td>$\text{Recv} \in \text{Port} \times \text{RecvOrder} \times \text{Data}$</td>
<td>$1+4+\text{sizeof(Data)}$</td>
</tr>
<tr>
<td>S-replay</td>
<td>$\text{Send} \in \text{Node} \times \text{Port} \times \text{SendOrder} \times \text{UAO}$</td>
<td>$1+1+4+4$</td>
</tr>
<tr>
<td>R-replay</td>
<td>$\text{Recv} \in \text{Port} \times \text{RecvOrder} \times \text{Hash}$</td>
<td>$1+4+8$</td>
</tr>
</tbody>
</table>

In D-replay, the record size is 5 bytes plus the size of the message payload. Hence, in the Bully benchmark, the size of a trace record is 9 bytes, which is less than 13 bytes and 10 bytes for the S-replay and R-replay, respectively.

4.5.3. Runtime overhead

Figure 4.19 compares the running times of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the D-replay technique.
Runtime overhead during recorded executions are 1.8x, 1.5x, 1.5x and 1.9x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.7x. However, replay executions runtime is less than baseline executions. This is because of two reasons: 1) during a replay execution, only one node is being replayed; 2) in fact, messages neither sent nor received. Actually, messages arrival is simulated. Hence, messages transfer time is eliminated.

Figure 4.20 compares the running times of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the S-replay technique.
Figure 4.20. The runtime overhead in S-replay

Runtime overhead during recorded executions are 2.6x, 3.3x, 4.2x and 2.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 3.0x.

Runtime overhead during replay executions are 5.6x, 4.6x, 5.4x and 3.5x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 4.7x.

Figure 4.21 compares the running times of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the R-replay technique.
Runtime overhead during recorded executions are 1.5x, 1.3x, 1.3x and 1.3x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.3x. Runtime overhead during replay executions are 2.3x, 1.9x, 1.8x and 1.6x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.9x.

4.5.4. Memory usage overhead

Figure 4.22 compares the memory usages of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the D-replay technique.
Memory usage overhead during recorded executions are 2.6x, 8.0x, 7.5x and 6.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 6.0x. Memory usage overhead during replay executions are 1.3x, 2.7x, 2.3x and 2.7x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.2x.

Figure 4.23 compares the memory usages of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the S-replay technique.
Memory usage overhead during recorded executions are 2.1x, 2.8x, 2.6x and 2.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.4x.

Memory usage overhead during replay executions are 3.0x, 2.8x, 2.6x and 2.6x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.7x.

Figure 4.24 compares the memory usages of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the R-replay technique.
Memory usage overhead during recorded executions are 2.7x, 2.9x, 3.0x and 2.8x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.8x.

Memory usage overhead during replay executions are 2.7x, 3.1x, 3.2x and 2.8x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 3.0x.

4.6. Features

In this section we discuss the features of DR-MCAPI such as usability, portability and scalability.

4.6.1. Usability

DR-MCAPI is a push-button solution. The user needn’t to change the source code or change/re-compile the MCAPI library. Using DR-MCAPI involves three steps: 1)
instrumenting the source code, 2) compiling the instrumented program, and 3) running the generated executable. These steps are easily automated using a batch script or could be incorporated into the MCAPI compilation chain.

4.6.2. Portability

DR-MCAPI doesn't require hardware amendments and since it sits as layer between the program and the MCAPI library, it is portable across different implementations of the MCAPI specification. For example, DR-MCAPI is usable with the OpenMCAPI [21] implementation without any changes.

4.6.3. Scalability

Now, we discuss the scalability of DR-MCAPI in terms of the trace size, runtime and memory overheads. The trace size scales linearly with the number of messages exchanged. However, since in D-replay the message payload itself is stored in the trace, the sizes of the messages affect the trace size resulting in larger traces. The S-replay and R-replay trace record sizes are independent of the message payload size. Table 4.7 compares runtime and memory usage overheads for the three replay techniques during recording and replay.

Table 4.7. Runtime and memory usage overheads of the replay techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Runtime Overhead</th>
<th>Memory Usage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recording</td>
<td>Replay</td>
</tr>
<tr>
<td>D-replay</td>
<td>1.7x</td>
<td>0.6x</td>
</tr>
<tr>
<td>S-replay</td>
<td>3.0x</td>
<td>4.7x</td>
</tr>
<tr>
<td>R-replay</td>
<td>1.3x</td>
<td>1.9x</td>
</tr>
</tbody>
</table>
A major factor affecting the scalability of DR-MCAPI is the time overhead, especially during a recorded execution. A replay execution is only needed when errors have been discovered and the developer needs to scrutinize the details. S-replay encounters high runtime overhead during a recorded execution since all messages payloads are modified before being sent and are unpacked and processed after being received. On the other side, R-replay has the least runtime overhead during recording since messages are accessed once (at the receiving node) to calculate the hash-code. S-replay exhibits a very high runtime overhead during a replay execution since it manipulates the orders of executing `dr_msg_send` (and `dr_msg_send_i`) calls across the whole program and not within a node similar to R-replay. The memory overhead is due to the DR-MCAPI data structures. D-replay memory overhead in the recording mode is the largest, since it buffers the contents of all messages exchanged till the trace is written to the disk. R-replay requires more memory than S-replay in the replay mode since it buffers messages received out of expected order.

### 4.6.4. Equivalent vs. identical replay

R-replay guarantees an *equivalent* replay of the recorded execution; however S-replay produces an *identical* replay. We use the program in Figure 4.25 to demonstrate the difference.
Let's assume that in a recording session of the program in Figure 4.25, the order of arrival of the message was as in Table 4.8.

Table 4.8. An order of arrival in a recorded execution

<table>
<thead>
<tr>
<th>Message</th>
<th>Destination Node</th>
<th>Arrival Order at destination node</th>
<th>Total Arrival Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent at line 6</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sent at line 7</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Sent at line 15</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Sent at line 16</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

When using R-replay, the local order of messages arrival at a given node during a replay execution is guaranteed to be the same as in the recorded execution. However, the total order of arrival of messages is not guaranteed to be the same. During a R-replay replay session, it is possible to have the order of message arrival as in Table 4.9 which is
equivalent to the one in Table 4.8, but not identical to it. It is worth mentioning that D-replay also produces an identical replay execution.

Table 4.9. Equivalent but not identical order of arrival

<table>
<thead>
<tr>
<th>Message</th>
<th>Destination Node</th>
<th>Arrival Order at destination node</th>
<th>Total Arrival Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent at line 6</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Sent at line 7</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sent at line 15</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Sent at line 16</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

S-replay guarantees a replay session that adheres to both the local and total orders of messages arrival.

4.6.5. D-replay vs. S-replay vs. R-replay

In this section we compare the three replay techniques. As shown in Table 4.10, R-replay exhibits better performance than S-replay and D-replay. D-replay is useful when only specific nodes are required to be replayed. S-replay is useful when identical replay is needed.

Table 4.10. D-replay vs. S-replay vs. R-replay

<table>
<thead>
<tr>
<th>Criteria</th>
<th>D-replay</th>
<th>S-replay</th>
<th>R-replay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Size</td>
<td>Worst</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td>Recording time overhead</td>
<td>Good</td>
<td>Worst</td>
<td>Best</td>
</tr>
<tr>
<td>Replay time overhead</td>
<td>Best</td>
<td>Worst</td>
<td>Good</td>
</tr>
<tr>
<td>Recording memory overhead</td>
<td>Worst</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Replay memory overhead</td>
<td>Best</td>
<td>Good</td>
<td>Worst</td>
</tr>
<tr>
<td>Replay specific nodes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Identical replay</td>
<td>Identical</td>
<td>Identical</td>
<td>Equivalent</td>
</tr>
</tbody>
</table>
4.7. Conclusion and future research directions

With the current trend of increasing a processor performance by adding more cores rather than increasing the clock speed, we may have processors with 10s or 100s cores in the near future. Currently, only a handful of applications can exploit the potentials of these multicore processors since only the very skilled programmers can develop applications for these processors. This must change. Every programmer should be able to write programs that take advantage of the multicore era processors. Hence, it is important to develop programming practices and tools that support multicore development. Providing a deterministic replay capability to multicore-specific standards such as MCAPI will greatly improve the debugging process. This is both an important and challenging problem. Any replay tool must be easy to use, scale well and handles all non-determinism sources in a program.

In this chapter, we presented DR-MCAPI. To the best of our knowledge, DR-MCAPI is the first replay tool that considers all non-determinism sources in MCAPI programs. The deterministic replay ability provided by DR-MCAPI allows a programmer to repeatedly execute the program under supervision of a debugger to catch flaws.

In terms of future work, we are considering these directions:

- Currently, the trace scales linearly with the number of messages exchanged during the runtime of a program. Reducing the trace size will decrease both the time overhead and memory usage, hence improving the scalability our tool. We plan to
investigate trace compression methods similar to the ones in [54] and [55] for DR-MCAPI traces.

- Check-pointing is a technique that allows recovery of a failed program to its state prior to failing [56]. Check-pointing works by periodically saving the state of a program to a stable storage during execution; when a failure takes place, the program is restarted from the last checkpoint [57]. We are exploring how to modify DR-MCAPI to support check-pointing for non-terminating MCAPI programs.

- Usability is of prime importance to any tool. That is why we are developing an Eclipse plugin that uses DR-MCAPI as a back-end to allow the user to perform interactive debugging.
CHAPTER 5

MESSAGE RACE DETECTION FOR WEB SERVICES

5.1. Introduction

Reliability is one of the four pillars necessary for producing trustworthy Web services [58]. Writing reliable Web services is difficult due to the unique challenges of this domain. In particular, Web services are prone to concurrency errors due to 1) concurrent processing of user/service requests; and 2) complex interaction behavior resulting from diverse communication mechanisms such as synchronous and asynchronous operations.

In order to develop reliable Web services, effective testing, analysis and verification techniques must be available to address these challenges. In this chapter we attack the problem of detecting message races in Web services. Race conditions are listed among the top 25 dangerous programming errors [59]; hence, detecting them is critical for Web services development.

Figure 5.1 illustrates a simple message race. WS1, WS2, and WS3 are three Web services. WS1 sends messages M1 and M2 to WS2 and WS3, respectively. WS3 reacts to the received message, by sending message M3 to WS2. Since M3 is sent in response to M1, WS3 would expect receiving M2 before receiving M3 as in scenario A. However, M3 may arrive at WS2 before M2 (scenario B) due to unexpected network latency between WS1 and WS2, or due to unforeseen impediment at WS1 that delays sending M2. Messages M2 and M3 are said to be racing with each other. Intuitively, two messages race with each other if either could be received first due to the unpredictability
of schedulers and message delays. Message races should be detected since they may be manifestations of bugs and can cause unpredictable results.

Unfortunately, traditional testing approaches that repeatedly execute or simulate a Web Service are not effective in detecting message races. First, such testing can be used to prove the existence of errors, but not the absence of them. Not detecting message races in multiple executions or simulations does not necessarily imply that they can’t happen. To completely verify the behavior of a Web Service, all possible scenarios must be examined. Explicitly examining all possible scenarios is a taunting task, if not impossible, as the number of possible scenarios is astronomical. Also, controlled testing can’t take into account unpredictable interactions that appear in the field. Second, Web services testers have to interpret vast amount of output to determine whether there exists
message races. This task alone takes non-trivial amount of time, and in many cases the output of an execution or simulation is considered correct by mistake even if there are message races. In the case where a message race is detected, the particular execution sequence that manifested the message race cannot be easily reproduced.

In this chapter we present a novel approach that addresses these problems that plague traditional testing approaches. Our approach can be used to prove the absence of message races within a bound specified by the user. Unlike most other static analysis approaches that report large amount of false negatives, only real message races are reported by our approach. In order to explore the astronomical amount of possible scenarios we model Web services using suitable classes of constraints and reduce various analysis problems to constraint solving. Figure 5.2 depicts the steps of our approach. First, a BPEL [60] process is translated to a WSMG model. Second, the WSMG is encoded as an SMT [61] formula. Third, an SMT solver is used to decide the satisfiability of the formula. We chose using SMT solvers as their performance has benefited from recent significant advances in Boolean satisfiability (SAT) solvers (e.g. [25], [23], [24]) and SMT solvers (e.g. [32], [31]).

![Figure 5.2. Steps for finding messages races in a Web service](image-url)
The semantics of non-determinism such as network latency is represented implicitly by
the SMT formula. The solution reported by the SMT solver offers detailed information
that explains how the message race happened. Thus the bug is reproducible in the sense
that the user can always simulate Web Service execution based on our bug report to
obtain the same message race.

The rest of this chapter is organized as follows. Section 5.2 presents our modeling
language for Web services. Section 5.3 details our approach to reduce message race
detection problem to constraint solving problem. Section 5.4 describes two case studies
and we conclude in Section 5.5 with contributions and limits of our approach.

5.2. Web Service Modeling Graph (WSMG)

In this section we define the Web Service Modeling Graph (WSMG) that is inspired by
hierarchical reactive modules [62]. WSMG is a compact representation that exhibits
concurrency and control flow in Web services.

A WSMG model represents a Web Service as a set of threads that communicate via
messages over a set of channels. A thread consists of a set of sequential transitions \( T \).
The set of transitions is defined as \( T \subseteq P \times Q \times Guard \times Action \), where \( P \) is the state
before the transition, \( Q \) is the state after the transition, \( Guard \) is a conditional expressions
and \( Action \in Asgn \cup \text{Snd} \cup \text{Rcv} \cup \{-\} \). \( Asgn \) is a set of assignment statements.
\( \text{Snd} \in Ch \times E \) sends the result of expression \( E \) over a channel in \( Ch \). \( \text{Rcv} \in Ch \times Var \)
receives a value from a channel and saves the value to a variable in \( Var \). No-op is
denoted by -. In WSMG there are two types of channels \( Ch = Ch_s \cup Ch_a \). \( Ch_s \) is a set of synchronous channels, over which both send and receive are blocking. \( Ch_a \) is a set of asynchronous channels, over which both send and receive are non-blocking if the buffer in a channel is not full during send action, and not empty during receive action.

We say a transition \( \tau \) in thread \( t \) has a \textit{token}, denoted as \( tk_t = \tau \), if it is a candidate for execution in a thread \( t \). At any time one transition per thread can have the token. We say a transition \( \tau \) is \textit{fired} if it is selected for execution. When \( \tau \) is fired, the token moves to the next transition in that thread. \( succ(\tau) \) denotes the next transition of transition \( \tau \). In the following we explain the execution semantics of a WSMG model:

- Let \( \tau = (g, v := expr) \) be a transition in thread \( t \). \( \tau \) can be fired if \( t \) is scheduled and \( tk_t = \tau \land g = true \). After the firing, \( tk_t = succ(\tau) \), and the assignment is executed.

- Let \( \tau = (g, snd(ch, E)) \) be a transition in thread \( t \) that sends the value of \( E \) to synchronous channel \( ch \), and \( \tau' = (g', rcv(ch, v)) \) is a transition in thread \( t' \) that receives to the variable \( v \) from channel \( ch \). Transition \( \tau \) can be fired if \( tk_t = \tau \), \( tk_{t'} = \tau' \), \( t' \) is scheduled, and both \( g \) and \( g' \) is true. In this case, \( \tau \) and \( \tau' \) are fired simultaneously. After the firings, the value of \( v \) is updated by the result of \( E \), and the tokens in \( t \) and \( t' \) are transferred to \( succ(\tau) \) and \( succ(\tau') \), respectively.

- Let \( \tau = (g, asnd(ch, E)) \) be a transition in thread \( t \) that sends the value of \( E \) to asynchronous channel \( ch \). Transition \( \tau \) can be fired if \( t \) is scheduled, \( tk_t = \tau \),
$g = \text{true}$ and the buffer in $ch$ is not full. After the firing, $tk_t = \text{succ}(\tau)$, and the value of $E$ is delivered to $ch$'s buffer.

- Let $\tau = (g, \text{arcv}(ch, v))$ be a transition in thread $t$ that receives a value from asynchronous channel $ch$. Transition $\tau$ can be fired if $t$ is scheduled, $tk_t = \tau$, $g = \text{true}$, and the buffer in $ch$ is not empty. After the firing $tk_t = \text{succ}(\tau)$, and the value of $v$ is updated by the removed value from $ch$.

- Let $\tau = (g, \text{fork}(t'))$ be a transition in thread $t$ that forks thread $t'$, and $\tau'$ be the first transition in thread $t'$. Both $\tau$ and $\tau'$ will be fired if $t$ is scheduled, $g = \text{true}$ and $tk_t = \tau$. After the firings $tk_t = \text{succ}(\tau)$ and $tk_{t'} = \tau'$.

- Let $\tau = (g, \text{join}(t'))$ be a transition in thread $t$ that joins thread $t$ with thread $t'$, and $\tau'$ be the last transition in thread $t'$. Both $\tau$ and $\tau'$ will be fired if $tk_t = \tau \wedge tk_{t'} = \tau'$, $g = \text{true}$ and $t'$ is scheduled. After the firings, $tk_t = \text{succ}(\tau)$ and $tk_{t'} = \bot$.

### 5.3. Symbolic encoding

In this section we present an encoding approach that converts a given WSMG model $G$ to an SMT formula that consists of initial constraint $\iota_0(G)$, thread scheduling constraint $\chi_B(G)$, transition constraint $\tau_B(G)$ and message race constraint $\rho_B(G)$. Whether there is message race up to the predefined bound $B$ can be checked by the validity of formula 1 which is equivalent to checking the satisfiability for formula 2.

$$\iota_0(G) \wedge \chi_B(G) \wedge \tau_B(G) \wedge \rho_B(G)$$

(1)
\[ u_0(G) \land \chi_B(G) \land \tau_B(G) \rightarrow \neg \rho_B(G) \] (2)

We use the SMT solver Yices [31] to solve formula 2. If the formula is satisfiable, the solution gives a trace that leads to a message race from the initial state in \( G \); otherwise, it is proved that \( G \) has no message race within \( B \) steps. In the following we first discuss the symbolic variables needed for the encoding, and then discuss the constraints.

5.3.1. Symbolic variables

In our symbolic analysis we check race conditions up to a pre-defined bound \( B \). For each step \( i < B \), we add a fresh copy for each variable introduced in this section. That is, \( \text{var}[i] \) denotes the copy of \( \text{var} \) at the \( i \)-th step. The symbolic variables are:

- **Token variable**: In order to encode the threads interleaving semantics symbolically we identify the set of threads in a given WSMG model and introduce one token variable \( t_k_t \) for each thread \( t \). A transition \( \tau \) has a token iff \( t_k_t = \tau \). Before a thread \( t \) is created or after it is terminated, we set \( t_k_t \) to be \( T \) or \( \bot \), respectively.

- **Model variables**: Given a WSMG model \( G \), we introduce a symbolic variable for each model variable in \( G \).

- **Scheduling variable**: To model non-determinism in the scheduler, we add a symbolic variable \( s \) whose domain is the set of thread identifiers. The value of \( s[i] \) indicates which thread is scheduled to execute at step \( i \). This is an important feature to our symbolic analysis in our approach. As in most cases the value of
s[i] is unspecified, the SMT solver is forced to consider the case where any thread can be scheduled to execute at step i.

- Asynchronous channel buffers: In our encoding we only consider channels with finite size buffers. Let the size of the buffer in ch be F, we introduce F symbolic variables $buf_1^{ch} \ldots buf_F^{ch}$, each of which represents a cell in the buffer of ch. A buffer is treated as a queue with $buf_1^{ch}$ and $buf_F^{ch}$ as its tail and head, respectively. We use a sentinel value $stnl$ to denote a cell without valid information. The buffer in ch is full iff $buf_1^{ch} \neq stnl$ and is empty iff $buf_F^{ch} = stnl$.

5.3.2. Initial condition constraint

The initial condition constraint $\iota_0(G)$ specifies the starting locations for each thread as well as the initial values of model variables, including the values set by the input vector.

5.3.3. Scheduling constraint

Our approach analyzes all possible valid interleavings, and excludes invalid ones. Therefore, we add thread scheduling constraint $\chi_B(G)$ to prevent invalid interleavings from being considered. In a WSMG model, a thread t must not be scheduled at step i in four cases: 1) before its creation, or after its termination (formula 3), 2) when an asynchronous send transition is pending and the relevant buffer is full, or when an asynchronous receive transition is pending and the relevant buffer is empty (formula 4), 3) when a synchronous send transition is pending, and there is no corresponding pending receive transition at another thread (formula 5), or 4) when a synchronous receive transition is pending, and there is no corresponding pending send transition at another
thread (formula 6). \( \tau_{as} \) is an asynchronous send transition, \( \tau_{ar} \) is an asynchronous receive transition, \( \tau_{ss} \) is a synchronous send transition, and \( Rv \) is all potential receive transitions of \( \tau_{ss} \), \( \tau_{sr} \) is a synchronous receive transition, and \( Sd \) is all potential send transitions of \( \tau_{sr} \).

\[
(tk_t[i] = \top \lor tk_t[i] = \bot) \rightarrow s[i] \neq t
\]

\[
(tk_t[i] = \tau_{as} \land \text{buf}_i^{\text{ch}} \neq \text{stnl})
\]

\[
\lor (tk_t[i] = \tau_{ar} \land \text{buf}_i^{\text{ch}} = \text{stnl}) \rightarrow s[i] \neq t
\]

\[
(tk_t[i] = \tau_{ss} \land \bigwedge_{(tr,p) \in Rv} (tk_{tr}[i] \neq p')) \rightarrow s[i] \neq t
\]

\[
(tk_t[i] = \tau_{sr} \land \bigwedge_{(tr,p) \in Sd} (tk_{tr}[i] \neq p')) \rightarrow s[i] \neq t
\]

The thread scheduling constraint is encoded as in formula 7, where \( \chi_B[i] \) is the conjunction of the constraints listed in formulas 3, 4, 5, and 6.

\[
\chi_B(G) = \bigwedge_{i=1}^B \chi_B[i]
\]

5.3.4. Transition constraint

The execution semantics of a thread is specified by the encoding of its transitions in a WSMG model. In the following we discuss the translation from transitions to SMT formulas based on the types of transitions:
An assignment transition in the format of $\tau = (g, v := expr)$ where $g$ is a guard, and $v := E$ assigns the results of $E$ to variable $v$ is encoded as in formula 8.

$$s[i] = t \land tk_t[i] = \tau \land g[i] \rightarrow tk_t[i] = succ(\tau) \land v[i + 1] = E[i] \land \delta([s, tk_t, v])$$

Formula 8 states that at step $i$, $\tau$ is fired under the following conditions: Thread $t$ is selected ($s[i] = t$), $\tau$ has token ($tk_t[i] = \tau$) and guard is true ($g[i]$). Note that $g[i]$ (or $E[i]$) means that all variables in the guard $g$ (or expression $E$) are replaced by their corresponding versions at step $i$. The following updates occur at step $i + 1$ when $\tau$ is fired at step $i$: the transition that succeeds $\tau$ in $t$ will have the token ($tk_t[i + 1] = succ(\tau)$), the value of $v$ at step $i + 1$ is the result of $E$ at step $i$ ($v[i + 1] = E[i]$) and the values of all variables except $s, tk_t, v$ remain unchanged from step $i$ to $i + 1$. Note that $\delta_t(S)$ means that all the variables except those listed in set $S$ keep their values step $i$ to $i + 1$.

A synchronous send/receive transition pair in the format of $\tau = (g, snd(ch, E))$ and $\tau' = (g', recv(ch, v))$, will be encoded as:

$$s[i] = t \land tk_t[i] = \tau \land g \land g' \land tk_{t'}[i] = \tau' \rightarrow (tk_t[i + 1] = succ(\tau) \land tk_{t'}[i + 1] = succ(\tau') \land v[i + 1] = E[i] \land \delta([s, tk_t, tk_{t'}, v]))$$
An asynchronous send transition in the format of $\tau = (g, \text{asnd}(ch, E))$, is encoded as:

$$s[i] = t \land tk_t[i] = \tau \land g \land buf_{ch}^F[i] = \text{stnl} \rightarrow (tk_t[i + 1] = \text{succ}(\tau) \land$$

$$\delta(\{s, tk_t, buf_{ch}^F, ..., buf_{ch}^F\}) \land (buf_{ch}^F[i] = \text{stnl} \lor buf_{ch}^F[i + 1] = E[i]; \text{E}[i]; buf_{ch}^F[i + 1] = E[i])) \tag{10}$$

An asynchronous receive transition in the format of $\tau = (g, \text{arcv}(ch, v))$, is encoded as:

$$s[i] = t \land tk_t[i] = \tau \land g \land buf_{ch}^F[i] \neq \text{stnl} \rightarrow$$

$$(tk_t[i + 1] = \text{succ}(\tau) \land v[i + 1] = buf_{ch}^F[i] \land$$

$$\land_{f=2}(buf_{ch}^F[i + 1] = buf_{ch}^{f-1}[i]) \land buf_{ch}^F[i + 1] = \text{stnl} \land$$

$$\delta(\{s, tk_t, buf_{ch}^F, ..., buf_{ch}^F, v\})) \tag{11}$$

A fork transitions in the format of $\tau = (true, \text{fork}(t'))$, will be encoded according to formula 12, such that $\tau'$ is the first transition in thread $t'$.

$$s[i] = t \land tk_t[i] = \tau \rightarrow tk_t[i + 1] = \text{succ}(\tau) \land tk_{t'}[i + 1] =$$

$$\tau' \land \delta([s, tk_t, tk_{t'}]) \tag{12}$$

A join transitions in the format of $\tau = (true, \text{join}(t'))$, will be encoded according to formula 13, such that $\tau'$ is the last transition in $t'$.

$$109$$
s[i] = t' \land tk_t[i] = \tau \land tk_{t'}[i] = \tau' \rightarrow tk_t[i + 1] =

\text{succ}(\tau) \land tk_{t'}[i + 1] = \perp \land \delta((s, tk_t, tk_{t'})) 

(13)

Let \gamma_t[i] denote the constraint for transition \tau at step \(i\) and \(T\) be the set of all transitions in a WSMG model, the transition constraint can be specified as

\gamma_B(G) = \bigwedge_{\tau \in T} (\bigwedge_{i=1}^B \gamma_t[i])

(13)

5.3.5. Message race constraint

A message race occurs on a synchronous channel ch when two conditions exist: a receive operation on ch is pending, and two or more send operations simultaneously attempt to deliver messages on ch. In such case, the received message is non-deterministic. Let \(SR_{ch}\) be the set of transitions with synchronous receive from ch and \(SS_{ch}\) be the set of transitions with synchronous send to ch. The constraint for synchronous message race at step \(i\) on channel ch can be specified as:

\alpha_{ch}[i] \equiv \exists_{\tau \in SR_{ch}} ((s[i] = t \land tk_t = \tau)

\land \exists_{\tau_1 \in SS_{ch}} \exists_{\tau_2 \in SS_{ch}} (tk_{\tau_1}[i] = \tau_1 \land tk_{\tau_2}[i] = \tau_2))

(14)

Message race happens on an asynchronous channel ch if ch is not full and there are multiple transitions trying to send messages over ch at the same time. In such case, the message saved in the buffer of ch is non-deterministic. Let \(AS_{ch}\) be the set of transitions with asynchronous send to ch. The constraint for asynchronous message race at step \(i\) on
channel $ch$ can be specified as in formula 15, where $\tau_1 = (g_1, asnd(ch, E_1))$ and $\tau_2 = (g_2, asnd(ch, E_2))$ are transitions in thread $t_1$ and $t_2$, respectively.

$$
\beta_{ch}[i] \equiv \left( \begin{array}{l}
ch \neq full \\
\wedge \exists \tau_1 \in AS_{ch} \\exists \tau_2 \in AS_{ch} (tk_{t1}[i] = \tau_1 \wedge tk_{t2}[i] = \tau_2) 
\end{array} \right) \tag{15}
$$

Let $ACH$ and $SCH$ be the set of asynchronous and synchronous channels in a WSMG model $G$. The message race property, up to bound $B$, can be specified by:

$$
\rho_B(G) = \bigvee_{i=1}^B \left( \bigvee_{ch \in ACH} a_{ch}[i] \vee \bigvee_{ch \in SCH} \beta_{ch}[i] \right) \tag{16}
$$

### 5.4. Experiments

To assess the feasibility of our approach, we applied it on the stock-trading and the loan-approval case studies from the BPEL-WS 1.1 standard [60].

As shown in Figure 5.3, the stock-trading case study consists of three sub-services: a quote service (SQS), a trading service (STS), and a bank service (Bank). The quote service has two threads that continuously send updated stock prices to the bank and the trading services. The trading service compares a received price to a minimum threshold.
and a maximum threshold. If the price is less than the minimum threshold, the trading service will send to the bank a buy request message. If the price is greater than the maximum threshold, the trading service will send to the bank a sell request message. Otherwise the trading service does nothing. The bank service updates its database when it receives new stocks prices from the quote service, and performs either selling or buying operations according to the requests received from the trading service.

Figure 5.3. The stock-trading Web service

We followed the steps depicted in Figure 5.2 and used Yices as the SMT solver. The solution produced by Yices indicates that a message race will occur when two quote services send prices-update messages to the bank service.
Table 5.1 shows the output of Yices which is an interpreted partial valuation to the symbolic variables in the SMT formula. In particular, we show the values of the token variables and the thread selection variable. The values of token variables indicate which transition is ready to be executed in a thread, and the value of thread selection variable shows which thread is scheduled at a given step. With the values of these two kinds of variables, the trace that leads to a message race can be replayed, thus solving the non-repeatability problem in the debugging of Web services. According to table 1, at the 10th step, the variable values satisfy the message race constraint: the bank thread is scheduled for execution \((S, T4)\) and its pending transition is a receive operation \((tk_4, B1)\). At the same time, there exist two send operations \((tk_3, Q6)\) and \((tk_5, S3)\) and all the three operations are on the same channel.

Table 5.1. Partial valuation of the stock-trading FOL formula

<table>
<thead>
<tr>
<th>Step</th>
<th>Partial Valuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((tk_0, M0), (tk_1, T), (tk_2, T), (tk_3, T), (tk_4, T), (tk_5, T), (S, T0))</td>
</tr>
<tr>
<td>1</td>
<td>((tk_0, \bot), (tk_1, Q0), (tk_2, T), (tk_3, T), (tk_4, B0), (tk_5, S0), (S, T1))</td>
</tr>
<tr>
<td>2</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q2), (tk_4, B0), (tk_5, S0), (S, T3))</td>
</tr>
<tr>
<td>3</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q4), (tk_4, B0), (tk_5, S0), (S, T3))</td>
</tr>
<tr>
<td>4</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q6), (tk_4, B0), (tk_5, S0), (S, T2))</td>
</tr>
<tr>
<td>5</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q3), (tk_3, Q6), (tk_4, B0), (tk_5, S0), (S, T5))</td>
</tr>
<tr>
<td>6</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q3), (tk_3, Q6), (tk_4, B0), (tk_5, S1), (S, T2))</td>
</tr>
<tr>
<td>7</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q5), (tk_3, Q6), (tk_4, B0), (tk_5, S1), (S, T2))</td>
</tr>
<tr>
<td>8</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B0), (tk_5, S2), (S, T5))</td>
</tr>
<tr>
<td>9</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B0), (tk_5, S3), (S, T4))</td>
</tr>
<tr>
<td>10</td>
<td>((tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B1), (tk_5, S3), (S, T4))</td>
</tr>
</tbody>
</table>

The second case study is based on the loan-approval Web Service which is shown in Figure 5.4. It consists of four sub-services: a customer service (Customer), an approval service (Approval), an approver service (Approver), and an assessor service (Assessor).
The approval service receives loan requests from the customer service. If the requested loan amount is less than a predetermined threshold, the loan request is sent to the approver service for automatic approval. Otherwise, the loan request is sent to the assessor service. When the assessor service receives a loan request, it assesses the risk associated with the customer, and then sends the risk assessment to the approval process. If the risk is high, the approval process denies the request; otherwise, the request is forwarded to the approver process. When the approver process receives a request, it automatically stamps the request as approved, and sends it back to the approval process. When the approval process receives an approved request from the approver process, it forwards the request to the customer.

Figure 5.4. The loan-approval Web service
When Yices is fed the SMT formulas corresponding to the loan-approval Web Service, it was able to detect a potential message race that happens when two quote services send prices-update messages to the bank service. Table 5.2 shows the output of Yices. At the 8th step, the variable values satisfy the message race constraint: \( L1 \) is scheduled for execution and it is a receive operation. At the same time both the pending transitions of \( C4 \), and \( C5 \) are send operations.

Table 5.2. Partial valuation of the loan-approval FOL formula

<table>
<thead>
<tr>
<th>Step</th>
<th>Partial Valuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( (tk_0, M0), (tk_1, T), (tk_2, T), (tk_3, T), (tk_4, T), (tk_5, T), (tk_6, T), (S, T0) )</td>
</tr>
<tr>
<td>1</td>
<td>( (tk_0, \bot), (tk_1, C0), (tk_2, R0), (tk_3, L0), (tk_4, S0), (tk_5, T), (tk_6, T), (S, T1) )</td>
</tr>
<tr>
<td>2</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L0), (tk_4, S0), (tk_5, C1), (tk_6, C2), (S, T3) )</td>
</tr>
<tr>
<td>3</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L1), (tk_4, S0), (tk_5, C1), (tk_6, C2), (S, T5) )</td>
</tr>
<tr>
<td>4</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L1), (tk_4, S0), (tk_5, C3), (tk_6, C2), (S, T5) )</td>
</tr>
<tr>
<td>5</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L1), (tk_4, S0), (tk_5, C5), (tk_6, C2), (S, T4) )</td>
</tr>
<tr>
<td>6</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L1), (tk_4, S1), (tk_5, C5), (tk_6, C2), (S, T6) )</td>
</tr>
<tr>
<td>7</td>
<td>( (tk_0, \bot), (tk_1, \bot), (tk_2, R0), (tk_3, L1), (tk_4, S1), (tk_5, C5), (tk_6, C4), (S, T6) )</td>
</tr>
</tbody>
</table>

The experiments were performed on a computer with Intel Core 2 Duo 2.6GHz processor and 4GB memory. Table 5.3 reports statistics that are related to solving the SMT formulas in the two case studies, including the number of decisions, number of conflicts, number of Boolean variables and memory usage during the SMT solving procedure. The last two rows list the memory and time usage of the two case studies.
Table 5.3. Yices statistics

<table>
<thead>
<tr>
<th>Yices Statistics</th>
<th>Loan-approval</th>
<th>Stock-trading</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Decisions</td>
<td>11833</td>
<td>7954</td>
</tr>
<tr>
<td>#Conflicts</td>
<td>6411</td>
<td>869</td>
</tr>
<tr>
<td>Boolean</td>
<td>8845</td>
<td>5176</td>
</tr>
<tr>
<td>Memory used</td>
<td>20.1</td>
<td>13</td>
</tr>
<tr>
<td>CPU Time</td>
<td>2.8</td>
<td>0.45</td>
</tr>
</tbody>
</table>

5.5. Conclusion and discussion

To improve the reliability and consequently the trustworthiness of Web services, potential messages races should be detected. We have addressed the problem of detecting message races in BPEL Web services. The main contribution of this chapter is a novel approach that reduces message race detection to constraint solving and uses modern SMT solvers to check the satisfiability of the SMT formula translated from the WSMG models. Given a predefined bound $B$, our approach is both sound and complete within the bound. Compared with traditional testing approaches that repeatedly execute or simulate a Web Service, the advantages of our approach include 1) ability to prove the absence of message races within a predefined bound, 2) implicit exploration of astronomical amount of possible scenarios, 3) no need to control the non-deterministic factors in Web services in testing environment, and 4) detailed bug reports.

However, even though all the message races reported by our approach are real, there are benign message races that are allowed by certain Web services. How to differentiate benign and malicious message races is an important area that is out of the scope of this chapter. For the future work, we plan to perform more significant case studies to future investigate the effectiveness of the approach.
CHAPTER 6

LITERATURE REVIEW

In this chapter, existing techniques and tools related to our work are reviewed, discussed and contrasted to our work.

6.1. Trace-based SMT-driven predictive analysis

Related work in this area could be divided into two categories; applying predictive analysis on multi-threaded programs and verifying message-passing software.

Sen, Rosu, and Agha pioneered using predictive analysis as a mean for detecting concurrency errors in multithread applications. In [63], they present the first tool capable of analyzing a trace of a Java program execution to predict the possibility of safety properties violations. They instrument the target program such that it emits a trace containing a list of particularly interesting events (e.g. shared variables access) while running. From the observed trace, a computation lattice is constructed. Paths in the lattice correspond to different executions and a node in a path represents a set of events. By traversing the computation lattice level by level, it is possible to generate every possible execution path that contains the events in the lattice. These executions are examined to determine whether an input safety property is violated.

In [36], Wang, Kundu, Ganai and Gupta introduced the notion of concurrent trace program which is a symbolic model that captures all feasible interleavings that can be predicted from an execution trace. The concurrent trace program is constructed by
combining an execution trace and information extracted from the source code. Then, the concurrent trace program is transformed into a concurrent single assignment form that is translated to a quantifier-free first-order logic formula. The satisfiability of this formula indicates the existence of an interleaving of the concurrent trace program that violates a given safety property.

The techniques introduced in [63] and [36] are inapplicable to message-passing software without major alterations since they target shared-memory multi-threaded programs.

MPI [20] has been dominating message-passing software development for a long time. Hence, the work on verifying the correctness of message-passing software was almost limited to MPI programs. Siegel et al. built MPI-Spin [64] which extends the SPIN [65] model checker with primitives that correspond to MPI-specific calls. MPI-Spin is based on the Urgent Algorithm that abstracts communication channels to rendezvous channels. The Urgent Algorithm is only applicable to programs with block and synchronous receives [66]; hence it is not usable for MCAPI programs. MPI-Spin was used to verify properties such as deadlock-freedom and halting. Similar to our work, MPI-Spin works by exhaustively exploring all the interleavings of a program in a symbolic manner. In contrast to our work, MPI-Spin requires a model of the MPI program as an input, rather than the program itself. This requirement severely limits the applicability of MPI-Spin, since a user would need to manually construct the model manually. MPI-Spin has an advantage over our work: it is able to verify the equivalency of a sequential implementation and an MPI-based implementation of the same algorithm.
In-Situ Partial Order (ISP) [67] presents a more-automated approach for verifying MPI programs. ISP creates a scheduling layer above the MPI runtime layer which allows intercepting MPI calls and discovering potentially matching send/receive ones. ISP provides an auxiliary function for every MPI function. These auxiliary functions consult a central scheduler through sockets when invoked. If the scheduler gives approval, the auxiliary function invokes the original MPI function. This permits the scheduler to go through the processes of an input MPI program according to an arbitrary interleaving, till all processes end. Then, ISP inspects the resultant trace of actions and records at every choice point, whether another process could have been selected. Such alternative choices are considered needed based on the dynamic dependence between the actions in the trace. For example, if, at choice point, another process P1 is found necessary to have been run, ISP will re-execute the entire program till it comes to choice point C, and picks P1 to run. This allows ISP to explore all possible execution scenarios resulting from different orders of messages’ arrival. ISP uses DPOR [68] technique to reduce the number of examined execution scenarios and has been used to verify deadlock-freedom, object leaks and safety properties.

MCAPI Checker (MCC) [41] is the first tool that attempts verifying MCAPI programs functional correctness. MCC employs the same technique used by ISP, but applies it on MCAPI constructs instead of MPI constructs. We couldn’t compare the performance of MCC and mzPredictor as MCC is not available for the public.
A very closely related work [69] presents an approach for modeling MCAPI programs using QF-FOL. A program trace is analyzed to generate a set of match pairs; a possible matching between a send operation and a receive operation. This set is encoded as a formula such that solving the formula assigns a send operation to each receive operation. They mention that the formula consists of five constraints, but they explain only three of them. Also, they neither provide experimental results nor sufficient technical details.

6.2. Deterministic replay for MCAPI

MPI [20] has been dominating message-passing software development for a long time. Hence, the current literature on replaying message-passing software is almost limited to MPI programs. In [70], Kranzlmuller et al. present a record and replay mechanism for MPI that adopts the order-replay approach and handles both promiscuous receive calls and test operations. Their approach is based on modifying the MPICH library source code. Different than MCAPI, not all MPI receive calls are promiscuous. MPI receive calls have a source parameter that can be used to state a specific sender process. If the source parameter is set to MPI_ANY_SOURCE, then the receive call may receive a message from any process allowing message races, otherwise, no message races can take place. Receive calls with MPI_ANY_SOURCE are handled by storing the identifier of the source process of the message that was received during the record phase. During replay, when the source parameter of a receive call is MPI_ANY_SOURCE, it is replaced with the source process identifier obtained during the record phase. Test operations are handled by counting the number of consecutive failing test operations associated with the same request variable during the record phase. In the replay phase,
test operations are forced to fail (i.e. return false) till the recorded number of failed tests
has reached. They report a whopping 200% time increase during the record phase. Also,
this approach is library-dependent (based on the MPICH library) which limits its
portability.

In [71], the authors disabuse the impracticality of data-replay and argue that the ability to
replay one process justifies the excessive logging overhead. They implement their data-
replay mechanism as a layer between the program and the MPI library. Recorded data
includes: MPI function calls return values and the contents and the source processes
identifiers of received messages. During replay, when the program posts a receive call;
the data-replay layer returns the data recorded at the corresponding receive call during the
record phase. In other words, receive calls are simulated rather than being executed. As
expected, the log size is 100's of times larger than when order-replay is used. In one
experiment, the data log was 907MB while an order-replay would produce 0.84MB for
the same program. The disk space requirement of this approach is prohibitively large for
long-running applications. Unfortunately the approaches described in [71] and [70] don’t
capture all forms of non-determinism in MPI programs, making it difficult to ensure a
completely faithful replay.

The authors of [72] propose subgroup-reproducible replay (SRR) which combines order-
replay and data-replay. During the record phase, disjoint groups of processes are formed
and the contents of messages crossing group boundaries are recorded. The contents of the
messages that are sent and received within a group are not recorded, but the order of
arrival of such messages is recorded. This approach allows replaying a specific group of processes independently of other groups. During replay of a group, messages coming from outside that group are reproduced from the log; inter-group messages are produced through direct execution. Setting the size and the membership of groups can be done manually by the user or automated based on communication locality. Performance evaluation of the SRR approach shows that it increases the runtime by an average of 120% during the recording phase and generates a log that is half the size of the log generated by a pure data-replay approach. Also this work handles all non-determinism sources in MPI programs.

Another related tool is MCC [41] which implements an automated approach for verifying MCAPi programs. MCC creates a scheduling layer above the MCAPi runtime layer that allows intercepting MCAPi calls and discovering potentially matching send/receive ones. This allows MCC to explore all possible execution scenarios resulting from different orders of messages’ arrival. MCC uses DPOR [68] technique to reduce the number of examined execution scenarios. MCC handles only promiscuous receive calls making it unsuitable for any programs using mcapi_test and mcapi_wait_any calls.

6.3. Message race detection for Web services

Although significant past work exists on detecting data races in shared memory programs and some past work exists on detecting message races in message-passing programs, the problem of detecting races in Web services is different enough that the past work does not directly apply.
Netzer and Miller [73] first characterize message races and design an on-the-fly algorithm for detecting them. Afterwards, Netzer et al. [22] improve their previous approaches by using a two pass hybrid on-the-fly/post-mortem scheme, and remove artifact races that are side effects of non-determinism from the bug report. In [74], Park et al. present an on-the-fly detection tool, which detects message races in MPI programs by checking communication concurrency in distributed processes.

Within the Web services research community, races are perceived as one of the problems arising due to feature interactions. In [75], a new approach for modeling and detecting undesirable interactions among web services is proposed. Race conditions are among these undesirable interactions. The work of Zhang, Su and Yang [76] focuses on detecting race conditions in Web services. They model a Web service as a Petri net, and then apply a race detection algorithm on that Petri net. The authors of [77] use a different approach to detect various feature interactions problems, including race conditions, in Web services. Their technique involves the translation of a Web service to the Promela language (the input language of the SPIN model checker), and expressing undesirable properties (i.e. the feature interactions problems) as linear temporal logic (LTL) formulas. Both the Promela translation and the LTL formulas are fed to SPIN which checks whether the properties hold or not. In essence, the approach of [77] is similar to ours, but no empirical results are shown that would allow us to compare the two approaches.
The most significant difference between previous work and ours is that most previous work uses existing languages and models that are intended for other domains such as hardware and network protocol designs. On the other hand, we use our Web Service Modeling Graph (WSMG) which is specifically tailored for modeling Web services. That makes our model compact, and leads to a small SMT formula. A smaller SMT formula means faster verification.

Also, our SMT-based analysis eliminates false positives and produces a trace that facilities pinpointing the source of the message race.
CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. Conclusion

Concurrent programming is not new. It has been around for almost half a century. However, due to its challenges, concurrent programming is accessible to a handful of programmers. Since new processors are offering more and more cores rather than faster clock speeds, concurrent programming is on its way to become mainstream. Message-passing based software development is gaining momentum as it provides a scalable and efficient alternative to shared-memory based software development.

The primary goal of this research is to alleviate the burden of debugging message-passing based concurrent programs. In particular, we enhance traditional testing by detecting much more errors that couldn’t be detected before, and we propose efficient techniques for deterministically replaying an execution in case of failure. In testing enhancement, we apply a novel predictive analysis technique in which a program execution model and a set of safety properties are encoded as a quantifier-free first-order logic formula, whose satisfiability determines if there exists an execution in which at least one safety property is violated. Also, we have developed DR-MCAPI, the first tool to introduce the deterministic replay capability to message-passing programs for multicore systems. During a recording phase, an unobstructed execution of an input program is monitored to produce a trace. In case of failure, the stored trace is used to enforce an execution that is logically equivalent to the one observed in the recording phase. We have successfully
applied our techniques on multicore programs developed using the new MCAPI specification and Web services described using the BPEL language.

7.2. Future research directions

GUI Frontend: Both mzPredictor and DR-MCAPI are command-line tools. We plan extending them by developing graphical front-ends. Such capability will allow the user to observe an erroneous execution (in case of mzPredictor) or a replayed execution (in case of DR-MCAPI) being carried out.

MCAPI 2.0 Compatibility: The multicore association has recently released the second version of the MCAPI specification. We plan to study it and adopt it in our tools.

Erlang: Currently, synchronization primitives used in shared-memory programs and communication constructs used in message-passing programs are implemented via third-party libraries that are not part of the programming language itself. In Earlang (a concurrent, distributed functional programming language) [78], concurrency is a first-class language feature whose support is deeply rooted in the language semantics. Even though that it has been available as open source for the past 10 years, Earlang lacks debugging support. We are planning on investigating whether it is possible to apply our predictive analysis and deterministic replay techniques on Earlang programs.
REFERENCES


[38] K. Sen and G. Agha, “Detecting errors in multithreaded programs by generalized predictive analysis of executions,” in *In Proceedings of 7th IFIP International*


