Regression Testing of Distributed Real-Time Embedded Systems in the Context of Model-Driven Development

by

Majid Babaei

A thesis submitted to the School of Computing in conformity with the requirements for the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
December 2021

Copyright © Majid Babaei, 2021
Abstract

Modern Distributed Real-Time Embedded (DRTE) systems are composed of large-scale, heterogeneous networks of software systems that constitute the backbone of many applications we rely on day-to-day. DRTE systems may evolve over time due to new requirements and technology improvements. Each revision requires regression testing to ensure that existing functionality is not affected by such changes. Despite recent progress in regression testing techniques and tools, it turns out they are inadequate for testing distributed systems. As such, a typical distributed system with a few nodes can produce a large amount of (possibly out of order) traces only after a few minutes of execution. Therefore, effective replay-based regression testing for distributed systems may require efficient mechanisms for generating compact traces, as well as reordering and replaying.

Model-driven Development (MDD) is a software development approach that advocates the use of models as the primary software development artifacts. Essentially, MDD leverages abstraction and automation in order to simplify design of a software system in terms of communication and activities.

In this work, we provide evidence that modeling can facilitate the ordering and replay of execution traces collected from a distributed system, and thus regression testing. Concretely, we show how the use of communicating state machines to describe
a distributed system obviates the need for timestamps and can be also leveraged for significant reduction in the amount of runtime information collected at reasonable cost.

As the first step of this work we present a novel trace reordering and replay technique and its tool support called \textit{MReplayer}. \textit{MReplayer} takes advantage of existing techniques and tools for model transformation and static analysis. \textit{MReplayer} relies on access to the descriptions of the behaviours of each component for static analysis and replay.

For the second step of this work, we present \textit{MRegTest}, an efficient replay-based regression testing approach for distributed systems at the model level. \textit{MRegTest} builds on our reordering approach and inherits from it a reduction in the number and size of traces required compared to standard, timestamp-based approaches.
Related Publications

Earlier versions of the work in this thesis were published as follows:


The following paper is not directly related to the work in this thesis, but was produced in parallel to the main research performed for this thesis:

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor Professor Juergen Dingel, for his invaluable feedback and continuous support throughout my Ph.D. studies. His immense knowledge, motivation and patience have given me more power and spirit to excel throughout this research project.

Besides my advisor, I would like to extend my sincere thanks to my thesis committee: Professors Farhana Zulkernine and Jana Dunfield for giving the encouragement and sharing insightful suggestions. I am also very grateful to my Ph.D. defence examination committee Professors Mohammad Zulkernine, Stephane Some, and Tom Dean for reviewing my Ph.D. work and providing valuable feedback.

I would always remember my former and current colleagues of the MASE research group, especially Mojtaba Bagherzadeh and Karim Jahed, for the fun time we spent together, sleepless nights that gave us the courage to complete tasks before deadlines.

My thanks also go to Dr. Bran Selic, Dr. Francis Bordeleau, and Dr. Ernesto Posse for their constructive advice and unwavering support throughout this project. Our meetings and conversations were vital in inspiring me to think out of the box and come up with creative solutions.

I am grateful for my parents whose constant love and support keep me motivated and confident. My accomplishments and success are because they believed in me.
Deepest thanks to my brothers, Hossein and Alireza and the rest of my family in Iran who have provided continued support throughout this phase and my life in general.

I owe my deepest gratitude to Shadi, who is my love. I am forever thankful for her unconditional love and support throughout the entire process and every day. You have been amazing, and now you can consider our summer vacation in Paris scheduled, as I promised!

Finally, this thesis would not have been possible without the financial support of the Natural Sciences and Engineering Research Council (NSERC), for which I am incredibly grateful.
Statement of Originality

I, Majid Babaei, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.
Contents

Abstract i
Related Publications iii
Acknowledgments v
Statement of Originality vii
Contents viii
List of Tables xi
List of Figures xii

Chapter 1: Introduction 1
1.1 Problem Statement .................................................. 3
1.2 Thesis Overview ................................................... 4
1.3 Thesis Contribution ............................................... 7

Chapter 2: Background 9
2.1 Core Concepts ..................................................... 9
  2.1.1 Model-Driven Development .................................. 9
  2.1.2 Distributed Real-Time Embedded (DRTE) systems ........ 12
  2.1.3 Regression Testing .......................................... 13
2.2 Combined Concepts .............................................. 15
  2.2.1 Model-Driven Development for Distributed Real-Time Embedded systems .............................................. 15
  2.2.2 Regression Testing at Model-level ......................... 16
  2.2.3 Regression Testing for Distributed Real-Time Embedded systems .......................................................... 17
2.3 UML-RT ............................................................. 20
  2.3.1 Modelling Structure of a System in UML-RT .......... 21
Chapter 5: Model-Level Replay-based Regression Testing for Distributed Systems

5.5 Experimental Evaluation
   5.5.1 Hypotheses
   5.5.2 Verification Approach
   5.5.3 Results and Discussions
   5.5.4 Threats to validity

5.6 Summary

Chapter 6: Model-Level Replay-based Regression Testing for Distributed Systems

6.1 Introduction
6.2 Running Example
   6.2.1 Leveraging MReplayer
6.3 Description of Approach
   6.3.1 Overview
   6.3.2 Detecting Regressions based on Modifications on re-steps
   6.3.3 Critical Variable Identification (CVI)
   6.3.4 Execution Selection (ES)
   6.3.5 Regression Detection (RD)
   6.3.6 Example
6.4 Tool Support
   6.4.1 Automatic Mutant Generation (AMG)
   6.4.2 Regression Testing (RT)
   6.4.3 MRegTest Features
6.5 Experimental Evaluation
   6.5.1 Prototype Implementation
   6.5.2 Evaluation Approach
   6.5.3 Results and Discussions
6.6 Threats to validity and limitations
6.7 Summary

Chapter 7: Conclusions and Future Work

7.1 Future Work
   7.1.1 Extend the support for more advanced features
   7.1.2 Extend the support for automatic critical variables identification
   7.1.3 Extend the evaluation of the proposed work

Bibliography
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Some of the rc-steps of CRT, BNK, and PPD (ref. Section 2.3.2)</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Complexity of Case Studies, Processing Time, Code Size Overhead, and Size of Collected Traces</td>
<td>82</td>
</tr>
<tr>
<td>6.1</td>
<td>Complexity of Use-cases, Size Reduction, and Performance of $ES$</td>
<td>116</td>
</tr>
<tr>
<td>6.2</td>
<td>Complexity of Use-cases, and Performance of $RD$</td>
<td>116</td>
</tr>
</tbody>
</table>
List of Figures

1.1 An overview of the content of the thesis ............................................ 5
2.1 Concepts related to our work ............................................................. 10
2.2 A general model of a distributed system ........................................... 20
2.3 The graphical UML-RT notation used in this work. .............................. 22
2.4 Models of Structure of Simplified Automated Teller Machine (ATM). .... 23
2.5 Models of Behaviour of Simplified Automated Teller Machine (ATM) ... 25

4.1 The capsule diagram of Robosoccer Player model ................................. 43
4.2 The capsule diagram of our proposed solution .................................... 44
4.3 The PlayerController, Interface and EventObserver state machines ..... 46
4.4 The TRACKING and SCORING state machines .................................. 46
4.5 The behaviour of the composite states of the SCORING state ............ 46
4.6 Structure of Simplified Content Management System ......................... 48
4.7 Behaviour of Simplified Content Management System ........................ 48
4.8 Sequence Diagram for Simplified Content Management System .......... 49
4.9 The structure of FailOver system ..................................................... 50
4.10 The behaviour of a server in FailOver system .................................... 51
4.11 The structure of Parcel Router system ............................................. 53
4.12 Models of a Dining Philosophers Problem .......................................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>An overview of MReplayer</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>Sample Execution of MReplayer on ATM using the input trace $T_{in}$</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>MReplayer WebUI</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>MReplayer Control Panel</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>ProcessResponse</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>RequestReply</td>
<td>85</td>
</tr>
<tr>
<td>5.7</td>
<td>SendKeepAlive</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Modified Behaviour of CTR’s USM in ATM</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>Sample Executions of the ATM system</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>An overview of MRegTest</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Sample Execution of our regression testing approach on ATM system</td>
<td>107</td>
</tr>
<tr>
<td>6.5</td>
<td>MRegTest’s AMG Control Panel</td>
<td>110</td>
</tr>
<tr>
<td>6.6</td>
<td>MRegTest’s RT Control Panel</td>
<td>110</td>
</tr>
<tr>
<td>6.7</td>
<td>MRegTest’s WebUI</td>
<td>111</td>
</tr>
<tr>
<td>6.8</td>
<td>Cost and benefit of our approach</td>
<td>120</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Due to advances in hardware technology, Real-Time Embedded (RTE) systems now often benefit from larger memory and enhanced computational power. Consequently, individual computing nodes are increasingly combined to form Distributed Real-Time Embedded (DRTE) systems. They typically contain many processors that interoperate via intricate network connections.

Recently, DRTE systems have been used widely in controlling diverse application domains including telecommunication systems, automobiles, aircraft, smart buildings, logistics, robotics, and many other applications. DRTE systems typically are reactive which means computation steps are caused by outside stimuli presented to system components in the form of messages. Basically, they rely on asynchronous message-passing mechanism that ensures loose coupling and isolation between nodes. Due to the increasing complexity of reactive DRTE systems, code-centric development approaches are not able to provide an appropriate level of support required for the development such complex systems.

Model-Driven Development (MDD) is a model-centric software engineering approach that aims at improving the productivity and the quality of software artifacts
by focusing on models as primary software development artifacts, rather than source code [22]. Abstraction and automation are the key concepts of MDD that can simplify communication and design activities, increase compatibility between systems, and boost development efficiency.

Models of DRTE systems may encompass large state machines including several components that communicate using various protocols. Indeed constructing such complex systems requires not only well-defined modeling constructs, but also strong tool support.

UML-RT [161, 151] is a domain-specific language specifically designed for RTE systems with soft real-time constraints. UML-RT has its roots in the well-known Real-time Object-Oriented Modeling (ROOM) language [159]. UML-RT has been used successfully in industry to develop several large-scale industrial projects, via, e.g., IBM RSA-RTE [97], HCL RTist [86], Protos eTrice [63] and Papyrus-RT [75].

The presented work is part of a larger research agenda that aims to identify and leverage model-driven software development concepts and techniques to facilitate the development of distributed systems.

**Thesis Statement:** Regression testing of distributed systems at the model-level can be facilitated via a flexible framework that provides deterministic reordering, as well as sufficient control over replay of execution traces collected from a distributed system at runtime. This not only allows the support of model-level regression testing in a way that is compatible with the semantics of the modeling languages, but also reduces the resource requirements on the nodes in the system and on the testing environment.
1.1 Problem Statement

The majority of software development time is spent typically on debugging, testing and verification of the software components individually and collectively. According to recent studies, software testing accounts for almost 50% to 75% of the software development budget \[22, 175\]. Existing modeling tools only provide limited support for testing models of distributed systems at the model-level. Thus, developers first must generate code from models and then use existing testing methods at the source code level to run their experiments and locate possible bugs in the software. To fix the bugs, again they need to go back to the model level and find the failing component in the model. Fixing bugs in this way is time-consuming and can be error-prone for developers who use only models for development.

Moreover, nondeterminism makes regression testing of distributed systems difficult. Without the ability to precisely reproduce a regression from the expected behaviour, it is challenging for developers to track down the root cause of a failure in the system. Traditionally, tools for observing and testing distributed systems rely on some form of timestamp, i.e., some information that is included in the traces and that allows the recipient to determine any temporal or causal order between them. Two kinds of timestamp can be distinguished. (1) Physical: Traces are annotated with the values from a local clock (e.g., \[112, 174, 50\]); (2) Logical: Traces are annotated with counter values \[114, 71, 158\]. The former requires sufficiently precise and synchronized clocks that can be expensive or even impossible for some resource-constrained systems, e.g., IoT devices, to maintain. The latter essentially increases the size of traces with the number of nodes in the system. Therefore, it is not applicable especially for distributed systems with many nodes, because the maintenance
and distribution of counter values can cause significant network overhead [181].

Despite recent progress in replay-based regression testing techniques and tools, e.g., [5, 93, 169], it turns out they are inadequate for testing distributed systems [111]. As such, a typical distributed system with a few nodes can produce a large number of (possibly out of order) traces even only after a few minutes of execution [166]. Therefore, an effective replay-based regression testing approach for distributed systems may require efficient mechanisms for replaying and regression detection [148].

In this thesis, we address two essential problems of regression testing of DRTE systems in the context of model-driven development. First, we propose an approach for reordering and replay of execution traces collected from a distributed system. To this end, we use model transformation to instrument the original model. Our approach relies on a model instrumentation process introduced in MDebugger [23]. Second, we propose an efficient replay-based regression testing method that is tailored to the characteristics of resource-constrained DRTE systems. Our work on regression testing builds on our reordering approach and inherits from it a reduction in the number and size of traces required. In fact, the work benefits from the increased level of abstraction together with strategic semantic simplifications, e.g., ‘run-to-completion’ assumptions, that a model-based system description can offer.

1.2 Thesis Overview

In the following, we provide an overview of the thesis. Figure 1.1 briefly describes the scope of this thesis. We first present introductory chapters that include Background (ref. Chapter 2), Related Work (ref. Chapter 3), and Case Studies (ref. Chapter 4), as follows.
1.2. THESIS OVERVIEW

Figure 1.1: An overview of the content of the thesis

Chapter 2: Background and Definitions
This chapter provides the reader with background information, definitions and the underlying techniques including an introduction to UML-RT, as well as a running example that we will use throughout the thesis.

Chapter 3: Related Work
To position this thesis with respect to prior research, we introduce the current state-of-the-art for reordering and replay of execution traces, and software regression testing.
Chapter 4: Case Studies

This chapter describes use cases with various complexities that are used to evaluate our proposed solutions.

The body of the thesis presents our novel solutions for two important problems of regression testing of DRTE systems. To facilitate the presentation of proposed solutions, we maintain the same structure for each chapter that consists of the following sections:

1. **Introduction**: gives an overview of the solution, requirements that must be met by the solution, and complementary background information specific to the solution.

2. **Running Example**: uses the running example introduced in Chapter 2 to explain the specific problem that is addressed by the solution.

3. **Description of Approach**: motivates and articulates the proposed solutions.

4. **Tool Support**: briefly introduces the prototypes developed based on the proposed solution and their features from the users’ points of view.

5. **Experimental Evaluation**: discusses the evaluation methods, conducted experiments, and results. A summary of the chapter is presented at the end of each chapter.

In the following, we present an overview of the main chapters.

Chapter 5: Model-Level Reordering and Replay of Execution Traces of Distributed Systems

Ordering and replaying of execution traces of distributed systems is a challenging problem. State-of-the-art approaches annotate the traces with logical or physical timestamps. In this chapter, we examine the problem of determining consistent orderings of execution traces in the context of model-driven development of distributed systems. By leveraging key concepts of state machines and existing model analysis
and transformation techniques, we propose a solution to collecting and reordering execution traces that does not rely on timestamps. Then, we describe a prototype implementation of our approach and an evaluation.

Chapter 6: Model-Level Replay-based Regression Testing of Distributed Systems

As software evolves, regression testing techniques are typically used to ensure the new changes are not adversely affecting the existing features. Despite recent advances, regression testing for distributed systems remains challenging and extremely costly. Existing techniques often require running a failing system several times before detecting a regression. As a result, conventional approaches that use re-execution without considering the inherent non-determinism of distributed systems, and providing no (or low) control over execution are inadequate in many ways. In this chapter, we present MRegTest, a replay-based regression testing framework in the context of model-driven development to facilitate deterministic replay of traces for detecting regressions while offering sufficient control over the execution of the changed system for the purpose of testing.

Finally, in Chapter 7 we introduce some potential future work including our ongoing work on extending our replay-based regression testing to handle other sources of modifications in a behavioral model such as some expressions in action code, as well as modifications in a structural model such as ports and protocols of a capsule.

1.3 Thesis Contribution

This thesis advances the state of the art theoretically and practically in the context of model-level regression testing of distributed systems. We propose efficient solutions
1.3. THESIS CONTRIBUTION

along with open source tooling. The most important contributions of the thesis are as follows.

• Proposing a conceptual framework for reordering and replay of execution traces in the context of model-driven development.

• Proposing a novel approach for replay-based regression testing of distributed systems in the context of model-driven development.

• As proof of concepts for the proposed solutions, we implemented two open source tools that can be integrated with existing modeling tools and extended for future research.
Chapter 2

Background

The objective of this chapter is to discuss the terms and concepts that are used in this work. The Venn diagram in Figure 2.1 shows the core and combined concepts related to our work. First, we briefly discuss each of these core concepts, i.e., Distributed Real-Time Embedded (DRTE) systems, Regression Testing, and Model-Driven Development. Then, we consider their combinations, i.e., Regression Testing at Model-level, Regression Testing for DRTE Systems, Model-Driven Development for DRTE Systems, and Regression Testing for DRTE Systems at the Model-level.

2.1 Core Concepts

There are a few concepts that lie at the root of our work. We discuss them in this section in more detail.

2.1.1 Model-Driven Development

A model is an abstract representation that captures certain aspects of a system, and offers a simplified view to process, analyze, and derive conclusions from the system
2.1. CORE CONCEPTS

Model-Driven Development (MDD)

Software Regression Testing (RT)

Distributed Real-Time Embedded (DRTE) systems

RT at the Model-level

RT for DRTE systems at the Model-level

MDD for DRTE systems

RT for DRTE systems

Figure 2.1: Concepts related to our work
under consideration. Traditionally, in the domain of software development, models have been used mainly for documentation purposes. Over the last few decades, however, models have attracted attention and have been employed as central artifacts throughout the entire software development process. This approach in software development is known as Model-Driven Development (MDD) [55]. By leveraging abstraction and automation, MDD can simplify communication and design activities, increase compatibility between systems, and boost development efficiency. In fact, MDD techniques and tools can be used at both design-time and run-time, to alleviate the complexity and provide an abstract, precise, and unambiguous representation of the system [15]. At design-time, Metamodels and Meta-metamodels (e.g., MOF [4], Ecore [2], and KM3 [103]) are used to define the appropriate and necessary structures, and properties to which a model must conform.

Model Transformation

A model transformation is an MDD technique that allows developers to transform a model from one representation to another (e.g., code) [104]. There are different types of model transformation such as endogenous and exogenous. In endogenous transformations, the source and target model conform to the same metamodel. In general, this type of model transformation is used to perform tasks such as model refactoring and optimization. However, in exogenous the source model and the target model conform to different metamodels. This type of model transformation is mostly used to handle tasks such as code generation, reverse engineering and migration.

Moreover, according to the representations of the source and target models, model transformation tools can be classified into three main categories: model-to-model,
2.1. CORE CONCEPTS

model-to-text and text-to-model.

In our work we use a model-to-model *endogenous* transformation technique to instrument a UML-RT model and then collect traces emitted from the code generated from the instrumented model.

2.1.2 Distributed Real-Time Embedded (DRTE) systems

Nowadays, Distributed Real-Time Embedded (RTE) systems play a fundamental role in controlling complex software systems. They are being utilized in many application domains such as automotive electronics, avionics, railways, telecommunication, health care, security, consumer electronics, fabrication equipment, smart buildings, logistics, robotics, and military. In general, the correctness of such systems is not only depends on the logical results of the computations, but also on the time within which results are produced. Consequently, a higher level of reliability and safety are required in such systems in comparison with general-purpose computing systems.

Typically, DRTE systems are categorized into two groups: (1) hard real-time systems; (2) soft real-time systems. The former group represents applications in which any timing fault is intolerable, and can cause catastrophic failure such as braking systems in automobiles. In contrast, violation of timing constraints in soft real-time systems only downgrades the system performance and it does not lead to catastrophic failure [22].

The inherent non-determinism of DRTE systems complicates the development and management of such systems. In fact, without the ability to precisely reproduce an execution that exhibits a regression, it is challenging for developers of DRTE systems to track down the root cause of a regression. Non-determinism of DRTE systems
arises from two main sources: (1) the local behavior of each individual node due to timing variations in the operating system and hardware; (2) the behavior of the network connecting the nodes for arbitrary network latency and routing mechanisms.

Thus, without mechanisms to control the non-determinism of DRTE systems, it is exceptionally challenging to get consistent results from multiple runs of a regression test.

There are two main approaches for handling the non-determinism in DRTE systems. The first approach is concerned with providing record & replay mechanisms that enable developers to reproduce a regressions repeatedly. The second approach eliminates non-determinism by controlling the execution environment. Our work was built on the first approach and introduces an efficient replay-based regression testing for DRTE systems at the model level.

2.1.3 Regression Testing

Regression testing [193] is a type of testing to provide confidence that changes made to the software, such as adding new features or modifying existing features, have not adversely affected existing features of the software [187]. Regression testing has been extensively studied by researchers [154, 187, 141, 193, 172]. The major challenge of using regression testing effectively is the growing size of the test suite that often makes it too costly to execute the entire test suite for every change in the software. A number of different approaches have been studied to maximize the value of the accrued test suite: minimization, selection, and prioritisation [187].
2.1. CORE CONCEPTS

Test Suite Minimization (TSM)

Large test suites, especially those generated by random testing, usually include redundant test cases that are not useful for regression detection. As discussed in the existing literature such as [84, 182], reducing highly-redundant test cases may have positive impact on the efficiency of regression testing. Essentially, approaches based on Test Suite Minimization (TSM) aim to identify redundant test cases and remove them from the test suite in order to reduce the size of the test suite and improve the efficiency of the regression testing.

Test Case Prioritization (TCP)

A mature software system often has many test cases that are generated during a software development process, essentially to ensure that every part of the software complies with its specification. Thus, running all the test cases may require significant amount of resources, as well as it might take a long time. To discover regressions from an expected behaviour in the software, a possible way is to rank all the test cases so that the regressions in the software can be exposed as early as possible [6]. TCP seeks to find an efficient ordering of test case execution for regression testing. TCP can have many variations depending on the underlying coverage criteria. Researchers have been using many criteria, such as statement coverage, block coverage, branch coverage, function coverage, and modified condition/decision coverage for prioritization [136].

Test Case Selection (TCS)

The main purpose of TCS is to reduce the cost of regression testing, especially when the number of test cases in a test suite is large. Test case selection is the process to
2.2. COMBINED CONCEPTS

select the most relevant test cases with respect to changes or updates made to the system under the test. Approaches based on TCS identify the test cases that are most relevant to the recent changes and most likely to exhibit regressions \cite{81,134}. TCS is essentially similar to the TSM-based approaches. However, there are some differences that are discussed in the existing literature such as \cite{193}. In fact, TSM is often concerned with the coverage of the software under the test when the test suite is minimized. By contrast, in TCS, test cases are selected because of their relevance to the changes between the original and the modified version of the software.

Inspired by TCS approaches, this work presents an optimization approach that reduces the cost of a replay-based regression testing significantly.

2.2 Combined Concepts

2.2.1 Model-Driven Development for Distributed Real-Time Embedded systems

The number and complexity of distributed systems is likely to continue to increase. Modeling techniques and tools have the potential to ease some of the resulting development, operation, and evolution challenges. Many of these systems will be reactive, i.e., rely on strong encapsulation and asynchronous message passing to achieve responsiveness and resilience \cite{35}. Examples include high-performance web-based applications such as Microsoft’s Halo \cite{127}, but also many cyber-physical systems (CPS) and Internet of Things (IoT) applications. Some of these systems will be resource-constrained, i.e., will have nodes that have limited capacity to receive, process, store, or send information.

Thanks to existing MDD techniques and tools such as \cite{24,17} the development
of complex systems can be greatly simplified. Many of these techniques benefit from *models@run.time* that enable developers to abstract certain aspects of a running system and mirror the state of the system at any time during its execution. According to this, the model and the running system are said to be *causally connected*, i.e., 1) changes in the system are reflected (as appropriate) in the model; 2) changes to the model are reflected in the system. Like design models, runtime models can be used to reason about systems. They can be used to observe executions, understand runtime behavior, and steer the development of the system \[33\].

In this work, we will use MDD approaches to reduce the size and the number of traces required for analyzing software behavior at runtime. These approaches allow us to obtain runtime information (i.e., traces) from the runtime execution steps using the corresponding elements in the behavioral model. Moreover, the amount of runtime information collected is further reduced through the use of model analysis, resulting in fewer and smaller traces.

### 2.2.2 Regression Testing at Model-level

Producing high-quality software is an ambitious goal that all developers of software systems want to achieve. Model-centric development creates opportunities for effective software testing. In particular, it enables developers to perform the testing process at a higher level of abstraction and to demonstrate compliance of the software with its predefined specifications \[138\]. Regression testing is an essential part of the testing process for ensuring the quality of evolving software and for gaining confidence in modified software \[42\].
According to the recent survey studies, e.g., [193, 123], the cost of regression testing accounts for almost 50% to 75% of the software development budget. Typically, developers spend a significant part of their time debugging applications and fixing bugs supported by advanced debugging tools. In recent years, numerous Model-Driven Development (MDD) techniques and tools such as [47, 13, 94, 12, 135] have been developed to mitigate the complexity of testing and debugging systems at model-level.

2.2.3 Regression Testing for Distributed Real-Time Embedded systems

Regression testing for distributed systems is a fairly important part of the development process of distributed systems especially when the software is supposed to be run on several nodes. Traditionally regression testing was performed manually on a distributed system which is expensive and error prone [200]. However, automated distributed regression testing is used extensively in more recent approaches such as [67, 174, 130, 122] that support a broader class of requirements and improve resource utilization [88].

Traditionally, tools for observing and debugging distributed systems rely on some form of timestamp, i.e., some information that is included in the traces and that allows the recipient to determine any temporal or causal order between them. Two kinds of timestamp can be distinguished. (1) Physical: Traces are annotated with the values from a local clock (e.g., [112, 171, 50]); however, the costs associated with keeping clocks sufficiently precise and synchronized can be significant [152, 158]. (2) Logical: Traces are annotated with counter values [114, 71, 158], which can be totally or partially ordered. For totally ordered logical clocks a global event counter
2.2. COMBINED CONCEPTS

for all processes is kept, while partially ordered logical clocks typically consist of an array of counters (usually called vector time or vector clock) associated with each process.

In this work we neither use physical nor logical timestamps to order traces collected from a DRTE system. Essentially, because in both methods the size of a timestamp increases with the number of nodes in the system which is unacceptable for resource-constraint distributed systems with possibly many nodes.

One difficulty in testing DRTE systems is that their distributed nature imposes theoretical limitations on the conformance faults that can be detected by the test components. Three architectures for testing distributed systems have been proposed in the past, with different fault detection capabilities:

Purely Distributed Architecture

It consists of several testers located at different sites. It implies three types of testing: (1) a remote test method, there is no coordination between the testers and each of them behaves independently, (2) a distributed test method where coordination among testers is allowed and testers can exchange messages between each other, (3) a coordinated test method where the coordination among testers has to follow specific procedures [26]. It also has been shown that the use of distributed testers can introduce controllability and observability problems, which can make it harder to apply a given test case [88]. It is easy to notice that these problems might have a great influence on several aspects of the testing activity such as the execution of test cases.
2.2. COMBINED CONCEPTS

Purely Centralized Architecture

It consists of one tester that controls the whole testing process. The centralized architecture has some advantages over the distributed architecture. For example, it is easier to verify the global properties of the distributed system once the whole state is represented in a single process. However, centralized control is less scalable, and thus it might prevent the framework from effective testing of larger deployments [130]. A commonly used approach in centralized architecture is testing by simulation, because running experiments on a realistic platform is costly and even impossible in some large distributed systems. Another approach is isolating a component under test and emulating other components of a distributed system [180]. Essentially, this method integrates individual components of a distributed system into a virtual environment that emulates the adjacent components of the system. This work falls within this group.

Hybrid Architecture

It combines centralized and distributed architecture approaches to improve fault detection by having a centralized tester that interacts asynchronously with the system under test and local testers that observe local traces [88]. Hybrid-based methods typically resolve the problems of controllability and observability by testing each component locally and exchanging coordination messages between local testers and the central tester [180]. However, it often poses higher complexity which may prevent its applicability in resource-constrained IoT devices.
2.3 UML-RT

In this thesis, we use UML-RT language to implement and evaluate our proposed solutions. UML-RT [161, 151] is a language specifically designed for real-time embedded systems with soft real-time constraints. Over the past two decades, it has been used successfully in industry to develop several large-scale industrial projects, via, e.g., IBM RSA-RTE [97], HCL RTist [86], Protos eTrice [63] and Papyrus-RT [75].

Recently, an extension of Papyrus-RT, proposed in [106], allows the development of distributed systems using UML-RT. Figure 2.2 shows a general model of a distributed system with \( m \) nodes (i.e., \{Node-1, Node-2, ..., Node-m\}) where each node consists of some capsules (i.e., components) that communicate with one or more controllers through a message passing mechanism. For example, Node-1 consists of \( p \) controllers (i.e., \{Controller-1, Controller-2, ..., Controller-p\}). Also, it shows that one or more capsules can be assigned to each controller. Eg., Node-1 consists of \( n \) capsules (i.e., \{Capsule-1, Capsule-2, ..., Capsule-n\}).
2.3. UML-RT

2.3.1 Modelling Structure of a System in UML-RT

UML-RT only provides two kinds of diagrams: capsule and state machine diagrams. In fact, based on UML-RT a system is designed as a set of interacting capsules. A capsule is similar to an active class in object-oriented programming, meaning that it may have autonomous behaviour. Capsules own a set of internal and external ports that are typed with protocols. A protocol defines the different incoming and outgoing messages that a capsule can receive or send through its ports. A port is the only interface for the communication between the capsules, which guarantees high encapsulation. Ports of two capsules can be connected through connectors only if they are typed with the same protocol. A port can be conjugated which means that the direction of messages is reversed. Furthermore, capsules can have attributes, operations, and parts [161, 160].

We also adopted the UML-RT formalization from [22]. Interested readers can refer to [151, 161, 186] for more in-depth information regarding UML-RT.

Definition 1. (Read function (Projection)) Let \( tp \) be a tuple \( \langle r_1 \ldots r_n \rangle \) where \( r_1 \ldots r_n \) refer to the names of the tuple entries. We use \( tp.r_i \) to read the value of entry \( r_i \). E.g., to read the value of entry name of tuple person\langle name, family \rangle we can use person.name.

Definition 2. (Structure of a Distributed System by UML-RT) Let us define a protocol as a set of pairs \( (m, d) \), where \( m \) and \( d \) denote a message and its current status, respectively. Also, \( m \in \mathcal{M} \), (i.e., a universal set of messages), and \( d \in \{ \text{input}, \text{output} \} \) specifies whether a message is consumed (input message) or produced (output message). A message can have a payload, which is a set of values conveyed by the message.

Let us define \( \mathcal{I} \) as a set of protocols and a capsule as a tuple \( \langle \mathcal{P}, \mathcal{V}, \beta \rangle \), where \( \mathcal{P} \subseteq \mathcal{P} \) (i.e., a universal set of ports) is a set of ports, \( \mathcal{V} \) is a set of variables,
and $\beta$ refers to the specification of the capsule’s behaviour. A port is defined as a pair $(t, \text{conjugated})$, where $t \in I$ refers to the type of the port, and $\text{conjugated} \in \{\text{true}, \text{false}\}$ specifies whether the port is conjugated (in UML-RT, connectors are always binary and connect ports that have the same type, but opposite conjugation).

Let us define the structure of a distributed system as a tuple $(C, I, \text{con}, \text{in})$, where $C$ is a set of capsules, $I$ is a set of protocols, $\text{con}$ is a connectivity relationship (i.e., $\text{con} \subseteq P \times P$), and $\text{in}$ is an acyclic containment relationship (i.e., $\text{in} \subseteq C \times C$).

### 2.3.2 A Running Example

We use a simplified Automated Teller Machine (ATM) UML-RT model as a running example throughout the thesis. Figure 2.4 shows the design of this model with PinPad server (i.e., PPD), a Controller server (CTR) and a Bank server (BNK), as shown in the structure diagram. Also, Figure 2.3 shows the graphical notation used to demonstrate UML-RT structural and behavioural models.
2.3. **UML-RT**

In UML-RT, the behavior of each capsule is specified using state machines which is an extension of a Mealy state machine \[128\] augmented with extra features, including state actions, composite states, and concurrency \[10\]. UML-RT State Machines (USM) consists of several states connected with transitions. Entry and exit actions of states, and transitions actions are expressed using an action language. Action languages support primitive operations such as accessing/updating variables, arithmetic/boolean expressions, control flow constructs, and sending messages. Typically, MDD tools such as Papyrus-RT provide action languages either by adapting a subset of well-known programming languages or by offering a specific, dedicated action language. In Papyrus-RT a subset of C++ is used as the action language \[22\]. In the following some aspects of a UML-RT behavioural model are explained in more details.

![System Structure of ATM](image-url)

**Figure 2.4:** Models of Structure of Simplified Automated Teller Machine (ATM)
**Definition 3.** (UML-RT state machine (USM)) The behaviour of a capsule is described with a UML-RT state machine (USM) defined as a tuple \( \langle S, T, \text{in} \rangle \) where \( S = S_b \cup S_c \cup S_p \) is a set of states, \( T \) is a set of transitions, and \( \text{in} \subseteq S_c \times (S \cup T) \) denotes an acyclic containment relationship. States can be basic \((S_b)\), composite \((S_c)\), or pseudo-states \((S_p)\). States are active until an outgoing transition is triggered. Composite states contain a sub-state machine. Pseudo-states are intermediate control-flow states. There are 6 kinds of pseudo-states, including initial, choice-point, history, junction-point, entry-point, and exit-point, \( (i.e., S_p = S_{in} \cup S_{ch} \cup S_h \cup S_j \cup S_{en} \cup S_{ex}) \). Composite and basic states can have entry and exit actions that are coded using an action language. Action languages allow expressions (over primitive types such as boolean and integer), and primitive operations such as accessing and updating variables and sending messages.

**Definition 4.** (UML-RT model) A UML-RT model \( M \) is defined as a collection of communicating UML-RT state machines \( (i.e., \text{USMs Definition 3}) \).

**Definition 5.** (Transition) Let \( \text{inp}(c) \) refer to the messages that can be received by capsule \( c \). A transition \( t \) is a 5-tuple \( \langle \text{src}, \text{guard}, \text{trig}, \text{act}, \text{des} \rangle \), where \( \text{src}, \text{des} \in S \) refer to the source and destination of the transition, respectively. Also, \( \text{guard} \) is a boolean expression in the action language, \( \text{trig} \subseteq \text{inp}(c) \) is a set of messages that trigger the transition, and \( \text{act} \) is the transition’s action code using the action language.

In the context of the running example, the state machines, that are shown in Figure 2.5, specify the behaviour of the instances of a component, for PPD, CTR, and BNK, respectively.
2.3. UML-RT

Run-to-completion (RTC)

It is a central concept in the definition of the execution semantics of many state machine languages including UML and UML-RT [69, 178]. It prevents the processing of an incoming message during the processing of an earlier message. RTC allows a sequence of micro steps to be viewed as a single execution step without loss of generality.

Definition 6. (RC-step) An rc-step rc is a tuple $\langle \sigma, C, P, \sigma' \rangle$, where $\sigma$ refers to the source state of $rc$, $\sigma'$ refers to the destination state of $rc$, $P$ is a finite sequence of transitions, and $C$ is a set of messages representing the trigger of the first transition in $P$.

Some of the rc-steps of the state machines in the ATM system are shown in Table 2.1.
2.3. UML-RT

2.3.4 Execution Semantics of UML-RT

The syntactic of UML-RT state machine have minor differences with those of UML including: (1) they do not support orthogonal composite states (multiple concurrent state machines in a composite state); (2) the use of some UML concepts such as fork, join, shallow history, and final states are prohibited in UML-RT; (3) transitions cannot cross state boundaries; (4) states do not have idle (do) actions [10, 22].

Moreover, the Run-Time System (RTS) library is used to manage the execution of an UML-RT execution. Basically, the RTS defines one or more controllers to monitor the concurrent execution of each capsule. Also, a controller is assigned to a physical thread, and controls the execution of a set of capsules.

**Definition 7. (Execution of a USM)** We define the execution of a USM using a Labelled Transition System \( \langle \gamma_0, \Gamma, R \rangle \), where \( \Gamma \) is a set of configurations and \( R \) is a set of rc-steps. A configuration \( \gamma \) is a tuple \( \langle \sigma, E \rangle \) where \( \sigma \in S \) is the active state (Definition 6) and \( E \) is a mapping from capsule variables and attributes to their values. \( \gamma_0 \in \Gamma \) is the initial configuration whose active state is the initial state and whose mapping assigns default values to all the variables and attributes. The execution of an USM is a possibly infinite sequence of executions of rc-steps from the initial configuration \( \gamma_0 \) (Definition 9).
Definition 8. (Actions of rc-steps) Let actions(rc) denote the sequence of actions of rc-step rc. This sequence includes exit actions of the source state of rc (i.e., exit(rc.σ)), actions of transitions of rc (i.e., act(t₁),...,act(tₙ) where t₁...tₙ ∈ rc.P), and entry actions of the destination state of rc (i.e., exit(rc.σ’)).

Definition 9. (Execution of a rc-step) an rc-step rc is enabled by configuration γ and message m, when the active state (γ.σ) is the same as the source state of rc, the guard of the first transition holds using the value map of γ, and m triggers the step (i.e., m ∈ rc.C).

Definition 10. (Execution Trace) Execution traces containing relevant runtime information are generated during system execution. We define a trace as a tuple ⟨L,P⟩, where P refers to the capsule id and L refers an executable element in the model (i.e., a state or transition) in the USM that is executed.

In the context of the running example, using the behavioural model of ATM we can define how user requests such as withdrawal and deposit will be processed. In below, the required operations for making and processing user requests are discussed in more detail.

In order to make either a withdrawal or a deposit request, first the component PPD sends the internal message init that initiates the operation. In state s₁ it then presents the user with options. Once a user selects one of the available options its corresponding boolean variable (i.e., optD, optW) turns to true. The component PPD interacts with CTR in order to transmit the request to BNK. Thus, PPD takes the transitions t₂,t₃ that send the message usrReq to the component CTR. Upon the reception of this message, CTR checks its connection to BNK where it responds by sending the message ack back to the CTR. This can carry either the value accept (i.e.,
transitions $t_2, t_3$) or reject (i.e., transition $t_4$) depending on whether the connection was established successfully or not. Once the value accept or reject is received by CTR, it sends the message ok or nok back to PPD, respectively.

Upon the reception of the message ok, PPD sends the request to CTR via the message req ($WDW, amt$) where $WDW$ and the variable $amt$ indicate the request’s type and the amount of money that the user wishes to withdraw from his bank account in BNK, respectively. Upon the reception of this message, the transitions $t_6, t_7, t_9$ are taken together in CTR. In this step, run-to-completion guarantees that the execution of the entire sequence of action is uninterrupted, until the execution reaches the next basic state. CTR supplements the withdrawal request with some additional information and sends the message $wdw$ and the variable $amt$ to BNK. In BNK the transition $t_5$ is taken and the value of $amt$ is deducted from the user’s balance amount and the result (i.e., $val$) is sent back to CTR via the message reply. Once this message is received by CTR, it takes transitions $t_{11}, t_{12}, t_{13}$ where it assigns the new balance amount to the variable $bamt$ and sends the message done to PPD. Then, PPD takes the transition $t_8$ and back to the state $s_1$ where it sends the internal message init and shows the available options to the user again.

2.4 Well-formedness conditions

State machine models used for MDD typically have to satisfy a number of well-formedness conditions which help ensure that correct code can be generated from the model. Our work also assumes that these conditions are satisfied. However, the following two conditions will be of particular relevance to us: (C1) ‘the guards of all transitions leaving a choice point are non-overlapping’, and (C2) ‘the guards of all
transitions leaving a choice point are exhaustive’. Together, both conditions ensures that triggering of a transition leaving a non-pseudo-state enables exactly one rc-step. While these conditions can be hard to check automatically, they are standard and typically included the user manuals and best practice descriptions of state machine tools such as IBM RSARTE and HCL RTist.
Chapter 3

Related Work

In this chapter, we survey the related research for the thesis. We organize the work into two sections: Reordering and Replay of Execution Traces and Software Regression Testing. In each section, we compare the relevant part of our approach with the existing work in the literature. Then we discuss how the related work motivates our proposed solutions.

3.1 Studies on Reordering and Replay of Execution Traces

3.1.1 Reordering and Replay for Distributed Systems

The techniques in this category essentially use recorded information collected from (possibly) several nodes in a distributed system to resolve the inherent non-determinism and replay executions accurately [51]. This, however, is insufficient for debugging distributed systems where a user wants to, e.g., stop the replay, change variable values and then examine the new execution that follows [27].

Traditionally, debugging and testing approaches for distributed systems rely on some form of timestamp that allows developers to determine any temporal or causal
3.1. STUDIES ON REORDERING AND REPLAY OF EXECUTION TRACES

order between execution traces and make diagnostic observations about the status of
the system. There are three types of timestamp, i.e., physical, logical, and hybrid,
that are discussed in the following.

**Physical-Timestamps**

One way to define an order of events in a distributed system is to maintain a global
physical-clock for the nodes in a distributed system. In this approach, a physical
timestamp is added to every trace generated from a distributed system. The accuracy
of ordering using this approach depends on precision of the global clock. The majority
of physical-timestamp approaches belong to one of the following groups.

*(a) Loosely synchronized clocks.* In this approach, clock synchronization among
nodes in a distributed system is achieved using techniques based on loosely synchro-
nized clocks [15]. For example, Adya et al. proposed a concurrency control method
based on multipart physical-timestamps [8, 7]. Also, *ORDO*, proposed by Kashyap
et al. [107], relies on invariant hardware clocks to provide the notion of a global hard-
ware clock. Recently, Yamada and Nakajima adopted synchronized global clocks for
distributed ledger technologies in order to improve scalability by reducing the number
of messages exchanged among the nodes [189].

*(b) Tightly synchronized clocks.* The Network Time Protocol (*NTP*) and the IEEE
1588 Precise Time Protocol (*PTP*) are among the most commonly used protocols for
clock synchronization between nodes in a distributed system [132, 64]. However, both
methods are bounded by the limitations of packet switching networks. So, network
characteristics such as jitter, packet buffering, and scheduling may influence their
timing properties and add non-deterministic variances to the synchronized clocks.
Recently, Lee et al. presented a Datacenter Time Protocol (DTP) using the physical layer of network devices to achieve nanosecond precision [117].

Logical-Timestamps

Providing a physically synchronous global clock for all nodes in a distributed system is very expensive and in some cases is not feasible. Logical-clock is a mechanism for capturing chronological and causal relationships between events. This mechanism adds a logical-timestamp to every trace generated from (possibly) several nodes in a distributed system. We classify work in this group into the following classes:

(a) Happened-Before (HB) relation. In 1978, Lamport introduced the concept of the HB relation as a way of timestamping and ordering events in an asynchronous distributed system [114]. Since then, several applications have resolved conflicting operations using the HB relation [125, 168, 73, 52]. For example, Virtual-time, proposed by Jefferson [101], implemented Lamport’s conditions using a Time Warp mechanism. Since the HB relation is transitively closed, its computation time can be expensive [185]. Smaragdakis et al. [168] proposed Causally Precedes (CP) over the HB relation which boosts the speed of race detection at the expense of creating a weaker relation. In addition, there is a growing body of research on Rule-Based HB relation methods such as EventTrack [125], DroidRacer [146] and SparseRacer [156], which leverage the concept of a directed graph over a set of operations to compute the HB relation. The complexity of graph traversal is very high which makes traversal-based approaches unsuitable for very large graphs. Bielik et al. [29] proposed EventRacer, which modifies graphs generated from the HB relation with vector clocks in order to speed up DFS traversals over the graph. Later, AsyncClock, proposed by Hsiao et al.
introduced a data structure to compute the set of candidate events.

(b) Vector-time. The concept of the vector-time was introduced by Mattern and Fidge as a data structure for representing causality relationships among events. Charron-Bost showed that the size of the vector-time must be as large as the number of parallel processes. Since then, several methods have been proposed, intended to reduce the size of the vector-time approach. For example, Singhal and Kshemkalyani introduced an enhanced version of vector-time which reduces the message-passing overhead at the expense of memory consumption. Also, Rodrigues and Verissimo compressed causal information using extracted knowledge from the underlying network. This approach was leveraged in the inline timestamps method proposed by Kulkarni et al. in which vector-time computation is delayed in order to reduce the size of vector-time. Shen et al. presented an encoded vector-time technique for large-scale distributed systems which optimizes the time and space complexity of the vector-time approach. In fact, the notion of causality is used in logical-clock-based approaches such as Totally Ordered Logical Clocks (TOLC) and Partially Ordered Logical Clocks (POLC) in order to determine precedence among observed events. They guarantee that the vector-times associated with events are consistent with the existing causality among events. TOLC’s maintain a global event counter for all processes, whereas in POLC-based methods an array of counters called vector-time is associated with each process.

(c) Matrix-Time. Matrix-time denotes a set of methods that maintain a timestamp value for each link between pairs of nodes in the system. Similar to vector-time, methods based on matrix-time require large amounts of memory due
to the exchange of large amounts of metadata. Also, Du et al. proposed [61] two protocols that provide scalable causal consistency using two-dimensional dependency matrices without relying on the transitivity of causality.

**Hybrid Approaches**

These approaches combine traditional vector-times with synchronized physical-times in order to reduce the size of timestamps, as well as provide more accurate synchronized clocks [139]. Kulkarni et al. [112] proposed a Hybrid Logical Clock (HLC) that maintains a relationship between the generated vector-time and physical-clocks synchronized by the NTP protocol. Demirbas et al. showed HLC’s application in Highly Auditable Distributed Systems [59]. Recently, Yingchareonthawornchai et al. [191] proposed a bounded size hybrid clock solution using a combination of HLC [112] and Hybrid Vector Clocks (HVC) [58]. They showed that their method is able to generate all possible snapshots from a distributed system at any given time [191]. Recently, Retroscope was proposed by Charapko et al. [45] in order to generate lightweight and consistent snapshots.

**Discussion.** While timestamps based approaches are suitable in many situations, the following disadvantages are typically associated with them: The physical-timestamps require, possibly highly synchronized, clocks [64] [132] [189]; the logical-timestamps tend to grow linearly with the number of nodes; and both are included in messages, increasing their size. A lot of work exists on mitigating these limitations [78] [113] [165] [167] [181]. Our work sidesteps these challenges and leverages static analysis and replay of state machines to avoid the use of timestamps.
3.1.2 Reordering and Replay in the Context of MDE

Hojaji et al. [92] survey model execution tracing, i.e., work that instruments the model or the generated code and allows replay and analysis of execution traces (e.g., [82, 177, 100, 99, 79, 90, 56, 137, 53]).

Iyengar et al. [100, 99] introduce an optimized model-based debugging technique for real-time systems with limited memory in which a monitor on the target platform collects traces, and a debugger executing on a host with sufficient memory analyzes the traces offline and displays results on the model elements.

Das et al. [56] propose a configurable tracing tool based on LTTng [60]. In this approach, code instrumentation is used to produce LTTng tracepoints. Also, trace replay allows timing analysis and can be performed offline or live, using a remote connection to the target platform.

Corley et al. [54] introduced a technique for supporting scalable omniscient debugging for model transformations. They proposed a set of basic features for omniscient debugging and extended traditional stepwise execution to support backward and forward traversal in the history of the execution. Their approach has been implemented in a cloud-based modeling solution called AToMPM. Also, a modified stepwise execution method is performed when the debugger applies forward debugging. Moreover, a full history of previous executions can be loaded from the collected log on a disk. This approach may impact the execution time of their proposed method, but it guarantees that the system remains within memory bounds for large-scale scenarios.

Bousse et al. [36] proposed a practical method for performing generic multidimensional omniscient debugging in an executable domain-specific modeling (xDSML) environment. Their method offers a generic set of debugging services for all xDSMLs
including a domain-specific trace meta-model and trace manager. They added an extension to this work in 2018 [37] and introduced a pattern to define runtime services independently of meta-programming approaches.

### 3.1.3 Reordering and Replay for Actor-based Languages

UML-RT’s execution semantics follows the actor model [87, 9], which underlies many recent highly responsive, fault-tolerant distributed applications [57, 34]. Approaches for record and replay for actor languages have been proposed by Tveito et al [183] and Aumayer et al [16]. Both approaches do not require timestamps.

Similar to our work, the work in [16] also aims to reduce the overhead on nodes. Tveito et al. formalize and prove a notion of confluence that could also be used to formally establish the equivalence of our consistent reorderings. However, neither approach directly supports trace reordering on the system level and the implementation of a centralized debugger.

In contrast, the work by Lanese et al [115, 116] and Shibanai et al. [108] does focus on debugging. In both approaches, traces are sent to a centralized component, which is similar to the work by Maiya et al. [125] on event-based programs and our work. However, both also rely on timestamps. Also, their suitability for resource-constrained systems is unclear: [115] relies on a somewhat idealized language (without, e.g., mutable objects); the experimental results in [116] are inconclusive and the number and size of traces is not measured; and [108] requires every node to run a JVM.
3.2 Studies on Software Regression Testing

3.2.1 Software Regression Testing for Distributed Systems

At a high-level we can group existing studies based on how much control they exert over the execution of the distributed system for the purposes of regression testing. Also, one can distinguish two kinds of test methods: centralized, i.e., one tester for all nodes; decentralized, i.e., local tester at each node [51]. As controllable aspects of the system and its execution environment we consider: (1) message delivery; (2) timing of executions; (3) non-deterministic operations. This creates a kind of spectrum of regression testing techniques that can be either centralized or decentralized.

**Low-Control Regression Testing.** At one end, centralized approaches such as SimRT [195], ReConTest [176], and ConTesa [194] exert no (or low) control over an execution during the testing. Since they typically require accurate clock synchronization among nodes, deterministic re-execution is hard to achieve. Also, due to the lack of control over the executions, it is difficult for developers to evaluate the performance of the system under certain conditions.

**High-Control Regression Testing.** Replay-based approaches and tools such as QF-Test [5], Selenium [93], and TestComplete [169] typically lie on the other end of the spectrum on which a fair bit of control is exerted over the execution. However, certain kinds of control such as message delivery mechanism are not feasible due to technical limitations or lack of access. Also, imposing large runtime overhead by excessive instrumentation might put the accuracy of testing at risk, because replay of collected traces might not be a good representative of the execution in the original system. Our approach is a replay-based centralized tester that benefits from a low-overhead instrumentation and offers sufficient control over the execution that allows...
a developer to stop the execution at any time and inspect the value of the variables in the system. Also, our approach leverages MReplayer’s action code interpreter to select and execute the proper rc-step at any step of the execution \[17\]. Our approach differs from MReplayer in that we use the reorder and replay mechanism to detect regressions.

3.2.2 Regression Test Selection (RTS)

According to existing surveys, e.g., [193, 32, 68], most of RTS techniques can be categorized into five groups based on granularity of software elements in test dependencies (i.e., the software elements that can be executed during each test execution): (1) basic-block-level, e.g., [142], (2) method-level, e.g., [199], (3) file-level, e.g., [80] [119], (4) module-level, e.g., [179] and (5) hybrid e.g., [198]. Although RTS approaches working on coarser granularities may have lower overhead, techniques based on finer granularities tend to be more precise in selecting tests. Our approach works on basic-block-level granularity, because it selects executions based on critical variables which can be modified in basic elements of a system at the model level (i.e., transitions). Also, depending on how the test dependencies are collected, RTS techniques can be categorized as dynamic [80] [85] [143] [199] and static [119] [118]. Unlike dynamic RTS, static RTS requires no code instrumentation or runtime information to find impacted test cases. Furthermore, static RTS uses static analysis to over-approximate the test dependencies and thus may select more tests than necessary [198]. Our approach falls within the scope of dynamic RTS category. Nevertheless, unlike the existing studies, that typically use re-execution for detecting regressions, our approach relies on replay of execution traces collected from a base model on its modified version.
3.2.3 Regression Testing in the Context of MDE

There are several approaches using model level information for regression testing \[14, 48, 31, 147, 95\]. Biswas et al. \[30\] proposed a model-based RTS technique that constructs a graph model from a program representing characteristics that are important for RTS. Their approach is not adequate for testing distributed systems because it relies on a few assumptions (e.g., synchronous message passing) that appear unreasonable for distributed systems. Zech et al. \[197, 196\] introduced a generic platform for model-based regression testing based on the model versioning engine MoVE and additional Object Constraint Language (OCL) statements. It also explains the process of generation and selection of regression test cases from the UML basic behavioural models. Similarly, Honfi et al. \[95\] proposed RtsMoT which is a model-based RTS approach for reconfigurable, autonomous robots. However, none of this work is applicable for distributed systems. Pal et al. \[145\] addressed this issue by introducing a testing framework for real-time distributed systems which is applicable to model refinements with new specifications to performing model-based distributed regressing testing. Similar to our approach online monitored data is used to obtain a coherent view of a distributed system. In contrast to the architecture of our approach which is centralized, their approach essentially uses the partitioning algorithm proposed in \[144\] to decompose a centralized tester into a set of communicating distributed local testers. Korel et al. \[110\] proposed a model-based RTS method based on Extended Finite State Machines (EFSM). Their approach considers only two types of Elementary Modifications (EM) on a model (i.e., addition or deletion of a transition). Chen et al. \[49\] addressed this issue by considering \textit{change in a transition} as another
EM type. In contrast, we reorder execution traces before doing regression testing, because we assume data is transmitted between components through an asynchronous message passing protocol. Furthermore, they don’t replay base model executions, as opposed to our approach; instead, they run the modified model multiple times to detect regressions. Also, there is some work on using UML models (i.e., class diagrams, collaboration diagrams, and statecharts) for regression testing [190, 40, 41, 70, 192].

3.3 Summary

In this chapter we have outlined the categories of research work that are relevant to our research. We divided existing approaches into two groups: (1) studies on Reordering and Replay of Execution Traces; (2) studies on Software Regression Testing.

First, we have shown examples of approaches for the reordering and replay of execution traces. In this group we have considered approaches for distributed systems, approaches in the context of MDE, and finally approaches for actor-based languages. To the best of our knowledge, the majority of these approaches are not concerned with trace reordering or they require synchronized clock for generating precise timestamps.

Second, we have presented studies on software regression testing including regression testing for distributed systems, regression test selection, and regression testing in the context of MDE. The main trend in these approaches is the use of re-execution for detecting regressions which may not be possible in some cases where the developers do not have access to the actual system just for the testing purpose.

In sum, we did not find a regression testing approach that can properly address challenges for reordering and replaying execution traces collected from resource-constrained distributed systems with possibly many nodes.
Chapter 4

Case Studies

In our work, case studies can be particularly useful for evaluating the efficiency and effectiveness of our proposed solutions. In this chapter, we briefly discuss several case studies that we use throughout the thesis. These case studies have been developed in collaboration with other members of the Modeling and Analysis in Software Engineering (MASE) group at Queen’s University. All case studies have been developed using Papyrus-RT \cite{Papyrus-RT} and their complexity ranges from simple to large models.

Papyrus-RT is an industrial-grade modeling environment that has been widely used for the development of soft Real-Time Embedded systems. Papyrus-RT provides a run-time service library, as well as modeling facilities based on UML-RT. It is built on top of Papyrus which is an open source modeling framework for UML. It has been customized for different UML profiles and purposes (e.g., SysML \cite{SysML}, MARTE \cite{MARTE}, and information modeling \cite{information_modeling}) and has recently attracted considerable attention in the modeling communities \cite{attention}.
4.1 Robosoccer Player (RP)

The Robosoccer Player system is the solution we proposed to the the RoboSoccer challenge problem\footnote{Available at \url{https://mdetools.github.io/mdetools19/challengeproblem.html}} that simulates a soccer game using Simgen\footnote{Available at \url{https://mdetools.github.io/mdetools19/challengeproblem.html}} between two robots. The goal of the game is to maneuver one of the robots to pick up the ball and deliver it into the other player’s goal. Each robot can be controlled via a dedicated TCP port. The game protocol over TCP allows for commands to move and rotate the robots, capture the ball, and shoot it in some direction. Game events such as ball possessions, timeouts, and scores are regularly published over a third TCP port.

The simulated environment of the RoboSoccer challenge contains two TCP ports (i.e., 9001 and 9003) for the external application to 1) control the blue player (i.e., Player1), and 2) control the red player (i.e., Player2). So, either the information such as the current position of the ball or the players or indicators such as goal scored are transmitted through these ports. Figure 4.1 shows an overview of our approach. In this figure TCP connections between each capsule and the simulation environment are shown using dashed lines. We start by describing the structure of the system and its components followed by the behaviour of each of those component.

The Structural Model. This model consists of two primary capsules: a) the Player capsule that implements the behaviour of the red player, b) the EventObserver capsule that connects to the simulation through a TCP port (i.e., 9007) and exchanges messages with the player capsule using an internal port eventObserverPort typed with the EventObserverProtocol protocol. The Player capsule diagram shown in Figure 4.2 consists of two sub-capsules (i.e., PlayerController and Interface). Note that both of these capsules are nested within the Player capsule as sub-components. It is entirely
possible to move these capsules to same hierarchy level as the Player. However, our design choice is intentional, as both the PlayerController and Interface implement auxiliary services that are needed only by the Player. This allows us to encapsulate the Player and all of its dependencies in one component that can be easily embedded into larger models.

The Interface capsule communicates with the PlayerController capsule through an internal port (i.e., interfacePort2) typed with the InterfaceProtocol protocol. Essentially, the Interface capsule provides network communication for the player in order to send controlling messages (e.g., spin, setSuction and moveForward) and obtain the game information through a TCP port (i.e., 9003). Also, the Interface capsule
4.1. ROBOSOCCER PLAYER (RP)

Figure 4.2: The capsule diagram of our proposed solution

is connected through a relay port (i.e., interfacePort1) to the EventObserver capsule that provides some additional information such as ball possession, goal scored and game timeout for the PlayerController capsule. Similarly, the PlayerController capsule communicates with the Interface capsule via an internal port (i.e, controllerPort) typed with the ControlProtocol protocol. In addition to the internal ports used to provide communication between sub-capsules, these sub-capsules also have two additional internal ports that connect the capsule to services provided by the run-time system. The timer port allows the PlayerController and Interface capsules to set timers and schedule events, while the logger port is used to output messages to the console.

The main purpose of the Interface capsule is to hide the network communication details from the Player capsule. The Interface capsule implements the necessary details to connect to the simulation environment over TCP and query it.

The Behavioural Model. Figure 4.3 shows the player capsule. It implements the logic for controlling and querying the player in the PlayerController State Machine and the Interface State Machine respectively.

Interface. The initial transition in the Interface leading to the CONNECTING state executes a standard piece of C++ action code to establish a TCP socket connection with the simulation environment. The state machine then transitions into the
4.1. ROBOSOCER PLAYER (RP)

IDLE state and remains in that state until a request is received from either the Player capsule or the EventObserver capsule. A request is simply one of the input messages defined in the InterfaceProtocol. Whenever such messages is received, the Interface transitions into the appropriate state to process the Player’s request. For example, upon entry to the new state, a message encoding the request, along with any parameters, is constructed and transmitted over the TCP socket to the simulation environment. The Interface remains in that state until a response message is received (from the simulation environment), and the appropriate response messages (an out message defined in the InterfaceProtocol) is sent back to the Player. The Interface then returns to the IDLE state and waits for the next message.

PlayerController. As demonstrated in Figure 4.4 and Figure 4.5, the state machine implements the main behaviour that tracks the ball and scores goals. The behaviour is divided into two phases: TRACKING and SCORING both of which are composite states.

In the TRACKING state, the player locates the ball position, moves towards the ball until it can can grab the ball and put into its front basket. So, it first sends the setSuction message with a positive value (i.e., 100). Then it obtains information about the player position, the player compass and the ball position. Then the direction of rotation and the degree are calculated and the spin message with a proper value is sent from the PlayerController capsule to the Interface capsule. The CALC ROTATION state calculates the rotation using the compass (i.e., c), playerGPS and ballGPS values. For example, if the rotation degree is less than a predefined threshold (i.e., 5 degrees) then the player sends the moveForward message to the simulation and moves towards the ball until the ball is sucked in and placed into the basket. This event is
detected by receiving a corresponding EventObserver message (i.e., possession) that generates the goScoring message. This messages would trigger a group transition from the TRACKING state to the SCORING state.

More information about the RoboSoccer case study can be found in [21].
4.2 Simplified Content Management System (SCMS)

Figure 4.6 and Figure 4.7 show the structure and behaviour of a Simplified Content Management System (SCMS) with Authentication Server (i.e., AUTH), a Content Provider Server (SRV) and a Client (CLI). The state machines, specifying the behaviour of the instances of a component, for AUTH, SRV, and CLI are shown in Figure 4.7.

For a user to access a specific document from CLI, he must first send a ticket request to the AUTH component (transition $t_2$ in CLI), which then checks the access policy of the requested document ($t_2$ in AUTH). If the document has no access restriction defined, AUTH generates an access ticket for the document and sends it to CLI ($t_3$). Otherwise ($t_4$), a ticket is sent only after successful validation of the user ($t_5$). After receiving an access ticket, CLI can send a request to SRV ($t_3$, $t_4$, and $t_6$ of CLI), which validates the ticket and provides the requested content only when the ticket is valid ($t_2$ and $t_4$ of SRV). Note that this system contains other user interaction components that are not discussed here. We assume that these omitted components forward user input (message $usrInput$) to CLI when needed. Figure 4.8 shows the message exchange between the participating capsules via a sequence diagram.

4.3 FailOver System (FO)

The FailOver system is a model that implements a basic fail-over mechanism [22]. This model is comprised of a set of servers processing client requests. It also supports two replication modes, i.e., passive and active to provide high availability of the servers for the clients. In passive replication mode, the model uses only one server to handle all the client requests and the other servers are largely idle. In this mode,
whenever the master server fails, a backup server is ranked up and substituted with the failing master server. In active replication, client requests are load-balanced between several servers.
4.3. FAILOVER SYSTEM (FO)

Figure 4.8: Sequence Diagram for Simplified Content Management System

Figure 4.9 shows the structural model of FailOver system. It consists of one Environment, i.e., ENV, two servers, i.e. SRV1, SRV2 and some clients, i.e., CLI capsules. The ENV capsule keeps track of the configuration information, such as the replication mode and the list of master servers. In case of a failure in the current master server it informs clients to send their requests to the new master server.

Figure 4.10 shows the behavioural model of server capsule. Backup is a composite state that has its own state machine. When a failure occurs, the running server takes the transition ServerFailure and goes from RunAsMaster into Failure state. When the server recovers from a failure, it may restart as either a master (i.e., transitions to state RunAsMaster) or a backup server, depending on the configuration received from the ENV. Frequently, the master server updates its state by sending two kinds of messages: IAmAlive and IAmMaster. The former is sent to the backup servers and the latter is sent to the environment capsule. If these messages are not received
within a specific period of time, its execution is considered to have failed, and a new server must be ranked up.

4.4 Refined FailOver System (RFO)

Refined FailOver is a debuggable version of a FailOver system generated using MDebugger [23]. MDebugger applies its transformations rules to enable an original FailOver system to interface with the debugger. Essentially, it adds: (1) a UML-RT port to each capsule (these ports are typed with a specific protocol used for debugging purposes); (2) required attributes and methods to support debugging such as maintaining breakpoint information, attribute view, and change operations; (3) a guard to every entry and exit code to prevent action code from being executed when the capsule is being debugged.
4.5 Rover System (RO)

The Rover system model [11] allows an autonomous robot to move in different directions. It is equipped with three wheels driven by two engines. It can move forward, move backwards, and rotate. Additionally, it is equipped with several sensors, such as temperature and humidity sensors to collect data from the environment, and an ultrasonic detection sensor to detect and avoid obstacles.

A Raspberry PI 3 board is used to build this model [11]. To provide some basic rover functionalities, the General-Purpose Input Output (GPIOs) pins of a Raspberry PI 3 are connected to Sensors and servomotors.

Moreover, three layers of UML artifacts are used to provide the Rover system’s functionalities. The bottom layer defines the services for accessing the GPIOs such as read/write access mode that can be set individually for each pin. The intermediate
model layer defines how the application layer may access the different sensors and actuators. Finally, the upper layer, which works as an application model, defines the basic rover operations such as drive to a certain location, collect data while avoiding obstacles. More detail information about Rover system can be found in [11, 10].

4.6 Parcel Router (PR)

The Parcel Router [173, 124] is simulation that routes tagged parcels through successive chutes and switchers to a corresponding bin. The system is time-sensitive and jams can happen due to variation in the time required by a parcel to transit through the different chutes. In this work, we used two versions of this model, i.e., the complete and the simple version. The former checks for potential parcel jams and prevents parcels from being transferred from one chute to another until the next chute is empty. Whereas the latter ignores jams and provides a simpler implementation of this model.

Figure 4.11 shows the capsule structure of the Parcel Router where the Gen capsule is responsible for generating parcels, and three stages are responsible for conveying parcels to one of four destination bins. In this model, each stage is further decomposed into chutes, switchers, and sensors which are not shown in Figure 4.11.

4.7 Dining Philosophers (DP)

Figure 4.12 shows the design of a Dining Philosophers model which is a classic illustration of resource contention and deadlock [98]. This model includes three philosophers (that can only think or eat) sitting around a circular table. In this example, there are three forks, one between each of two philosophers, and a philosopher needs two forks
to eat. So, there should be a proper resource allocation strategy to avoid getting stuck in a deadlock where a philosopher holds one fork while waiting for another fork to be released. In the UML-RT implementation, an Arbitrator capsule is used to guarantee that a Philosopher capsule can only pick up both forks or none at a time. Thus, the Arbitrator either accepts a Philosopher request by allocating both forks or declines its request. Since focusing on resource allocation strategies to avoid deadlock is not the main objective of this running example, we avoid using separate capsules for each fork in the model. Instead, we adopt the design model introduced in [98] and explain the most relevant features of MReplayer.

Philosopher capsule instances (i.e., PH_1, PH_2, and PH_3) interact with the Arbitrator capsule instance (i.e., ARB) concurrently through ports typed with the same protocol. This protocol defines two messages request and reply. The former message can carry either pickup or putdown and the latter can carry either true or
false as data. In the interest of space, only some important aspects of this example are explained below.

The \( \text{ARB} \) takes the initial transition \( t_1 \) and waits in the state \( s_1 \) for a message \( \text{request} \) from \( \text{PH}_1 \) or \( \text{PH}_2 \) or \( \text{PH}_3 \). Upon reception of the \( \text{request} \) messages, the transition \( t_2 \) is taken and the \( \text{ARB} \) examines whether the forks on both sides of the philosopher are free (e.g., for \( \text{PH}_1 \) \( \text{forks}[0] \) and \( \text{forks}[2] \) are checked). If so, the \( \text{ARB} \) sends \( \text{reply}(true) \) and updates variable \( \text{forks} \) accordingly (e.g., for \( \text{PH}_1 \) \( \text{forks}[0]=1 \) and \( \text{forks}[2]=1 \)). The three instances of the capsule \( \text{Philosopher} \) have the same behaviour that consists of two states, Thinking and Eating. The Eating state
is a composite state including \textit{PickupReq}, \textit{Enjoying}, and \textit{PutdownReq} sub-states. In \textit{PickupReq}, the message \textit{request(pickup)} is sent to \textit{ARB}. Upon reception of the message \textit{reply(true)} from \textit{ARB}, the transitions $t_4, t_5$ are taken and the \textit{Philosopher} enters to the sub-state \textit{Enjoying} where it sets a timer for eating. Otherwise, transitions $t_4, t_8, t_9, t_{10}$ are taken and the execution returns to the state \textit{Thinking}. Once the timer goes off in the sub-state \textit{Enjoying}, the \textit{Philosopher} takes the transition $t_6$ to the sub-state \textit{PutdownReq} where it sends the message \textit{request(putdown)} to \textit{ARB} and it returns to the state \textit{Thinking}.

\section*{4.8 Simple Models}

We also use some simple academic models such as \textit{Car Door Central Lock} (CDCL) system \cite{[22]} that simulates the opening and closing of car doors in a centralized way.
Chapter 5

Model-Level Reordering and Replay of Execution Traces for Distributed Systems

5.1 Introduction

Providing a deterministic replay of execution traces collected from a distributed system is a challenging task that occurs in various contemporary debugging and testing techniques. For example, debugging by replay \[51\] is one of the most common debugging methods for software systems \[27\], allowing developers to execute the recorded traces of a system repeatedly and make diagnostic observations. However, due to the following reasons supporting debugging of distributed systems via replay is challenging:

- It requires efficient mechanisms for generating useful traces and collecting them from possibly many nodes. The size of traces collected from a distributed systems (even with a few nodes) can grow exponentially within a short period of time.
• Trace replay must be deterministic (i.e., repeatable) and present a view consistent with the true state of the, usually concurrent and non-deterministic, system execution. Due to the distributed nature of the system, the order in which traces arrive at the replayer may not reflect neither the temporal order nor the causal order between steps.

The number and complexity of distributed systems is likely to continue to increase. Modeling techniques and tools have the potential to ease some of the resulting development, operation, and evolution challenges. Many of these systems will be reactive, i.e., rely on strong encapsulation and asynchronous message passing to achieve responsiveness and resilience [35]. Examples include high-performance web-based applications such as Microsoft’s Halo [127], but also many cyber-physical systems (CPS) and Internet of Things (IoT) applications. Some of these systems will be resource-constrained, i.e., will have nodes that have limited capacity to receive, process, store, or send information.

In this chapter, we present the first step of our work to facilitate the development of distributed systems. Concretely, we present an approach to the problem of reordering the traces of reactive distributed systems that satisfies the following three requirements:

Requirement R1: The approach leverages existing modeling techniques and tools.

Requirement R2: The approach aims to reduce the resource requirements on the nodes in the system.

Requirement R3: The approach is compatible with the implementation of a centralized debugger and tester.
Traditionally, tools for observing distributed systems rely on some form of timestamp, i.e., some information that is included in the traces and that allows the recipient to determine any temporal or causal order between them. Two kinds of timestamp can be distinguished. (1) Physical: Traces are annotated with the values from a local clock (e.g., [112, 171, 50]); however, the costs associated with keeping clocks sufficiently precise and synchronized can be significant [152, 158]. (2) Logical: Traces are annotated with counter values [114, 71, 158], which can be totally or partially ordered. For totally ordered logical clocks a global event counter for all processes is kept, while partially ordered logical clocks typically consist of an array of counters (usually called vector time or vector clock) associated with each process. In both cases, the size of a timestamp increases with the number of nodes in the system. Again, the maintenance and distribution of counter values can cause significant network overhead [181].

We also leverage an often-made atomicity assumption that greatly simplifies dealing with concurrency: run-to-completion (RTC). RTC means that the handling of an incoming message by a component is not interrupted by the arrival of another message. RTC is well-known from UML state machines [178], but also underlies many languages (or language extensions) built on the actor model such as Akka [170] or Orleans [43], as well as many dynamic languages relying on events and event loops such as JavaScript or E [57, 34, 131, 16]. Many of these languages have successfully been used to implement industrial reactive systems [127, 34]. In our work, RTC allows us to group all execution steps that make up the handling of the message into a single RTC (i.e., macro) step. This, in turn, enables a reduction of the amount of traces and instrumentation required which helps keep runtime overhead low.

We have implemented our approach in the context of the UML-RT profile in a
5.1. INTRODUCTION

A prototype called MReplayer [19, 17, 18]. The reordering component of MReplayer takes an unordered trace as input and produces all reorderings of the input trace that are consistent with the control flow and communication dependencies of the models as output. It also leverages the reordering to provide some basic debugging features such as deterministic replay with step-in, step-over, and step-back.

MReplayer is a first step in our efforts to provide regression detection support for models of distributed systems, and realize more sophisticated regression testing services as described in [193, 25, 51, 121, 83].

Moreover, MReplayer uses existing model-to-model transformation techniques [105] to implement the instrumentation and ensure that the generated code emits the expected trace information at runtime. Techniques from static analysis are used to compute all possible run-to-completion steps of each component and determine control flow and communication dependencies.

We have evaluated our approach and MReplayer on execution traces obtained from models with various levels of complexity with favourable results: In our experiments, the static analysis even of large models (i.e., models with more than 2000 states and 3000 transitions) takes less than a minute, and the size of traces is reduced by a factor of 2.7 compared to the classical vector time approach. At the same time, the costs of instrumentation and trace generation at runtime are not significantly different from those of the classical approaches based on vector time. We believe that our work provides some evidence for the claim that modeling and the increased level of abstraction it offers cannot only facilitate system design, but also subsequent activities such as debugging and maintenance.

The rest of this chapter is organized as follows. Section 5.2 presents a running
5.2. Running Example

In this chapter, we use the running example (ref. Section 2.3.2) to illustrate a situation where traces (generated from different nodes in a distributed system) arrive out of order at a centralized replayer.

Suppose the ATM system (ref. Section 2.3.2) is deployed on a distributed environment (i.e., each component is deployed on a separate node), and its components generate execution traces for each transition. Listing 5.1 shows an example sequence of traces received by the replayer application, ordered by their arrival time at the replayer.

Each trace consists of the name of the component that generated the trace followed by a timestamp indicating when the trace was generated, as well as the transition name, the source state of the transition and the destination state of the transition. For example, the first line (i.e., line #1) shows that PPD took the transition $t_1$ from

<table>
<thead>
<tr>
<th>No.</th>
<th>Component</th>
<th>Timestamp</th>
<th>Transition</th>
<th>Source State</th>
<th>Destination State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PPD</td>
<td>14:15:23.060</td>
<td>$t_1$</td>
<td>in1</td>
<td>s1</td>
</tr>
<tr>
<td>2</td>
<td>BNK</td>
<td>14:15:23.061</td>
<td>$t_1$</td>
<td>in1</td>
<td>s1</td>
</tr>
<tr>
<td>3</td>
<td>CTR</td>
<td>14:15:23.062</td>
<td>$t_1$</td>
<td>in1</td>
<td>s1</td>
</tr>
<tr>
<td>4</td>
<td>PPD</td>
<td>14:15:23.063</td>
<td>$t_2$</td>
<td>s1</td>
<td>ch1</td>
</tr>
<tr>
<td>5</td>
<td>PPD</td>
<td>14:15:23.067</td>
<td>$t_3$</td>
<td>ch1</td>
<td>s3</td>
</tr>
<tr>
<td>6</td>
<td>BNK</td>
<td>14:15:23.068</td>
<td>$t_2$</td>
<td>s1</td>
<td>ch1</td>
</tr>
<tr>
<td>7</td>
<td>BNK</td>
<td>14:15:23.068</td>
<td>$t_3$</td>
<td>ch1</td>
<td>s2</td>
</tr>
<tr>
<td>8</td>
<td>CTR</td>
<td>14:15:23.068</td>
<td>$t_2$</td>
<td>s1</td>
<td>s2</td>
</tr>
<tr>
<td>9</td>
<td>CTR</td>
<td>14:15:23.069</td>
<td>$t_3$</td>
<td>s3</td>
<td>ch1</td>
</tr>
<tr>
<td>10</td>
<td>CTR</td>
<td>14:15:23.069</td>
<td>$t_4$</td>
<td>ch1</td>
<td>s3</td>
</tr>
</tbody>
</table>

Listing 5.1: Sample sequence of tracing information for ATM example. Section 5.3 describes our approach. The evaluation, experimental results, and threads to validity are summarized in Section 5.5.
5.2. RUNNING EXAMPLE

the source state $s_1$ to the destination state $s_1$ at 14:15:23.060. In fact, traces in lines #1, #2, and #3 correspond to the initial setup actions for PPD, BNK, and CTR, respectively. The trace in line #4 indicates that PPD sends a user request message, i.e., userReq to CTR. Then, CTR receives the user request message (line #8), sends a connection request message, i.e., con to BNK. Once this message is received by BNK, it checks whether the connection is secure enough or not and then send the message ack along with the payload data accept back to CTR (line #6 and #7). Finally, CTR takes the transitions $t_3$ and $t_4$ and sends the message ok to PPD (line #9 and #10).

The order in which traces arrive at the centralized replayer shown in Listing 5.1 suggests that the bank sent the message ack to the controller, i.e., traces in lines #6 and #7, before the connection request from the controller, i.e., the trace in line #8. However, this is, in this case, an incorrect conclusion and the late, out-of-order arrival of traces in lines #6-7 is due to other factors such as the delayed transmission of messages between CTR and the centralized replayer.

Timestamping traces can help detect such out-of-order delivery. Assuming that the time values included in the traces are correct, a centralized replayer can use physical timestamp techniques to determine the proper order of messages and to, e.g., delay the replay of traces #6-7 until the trace #8 has arrived. However, keeping clocks sufficiently precise and synchronized can be expensive, especially for large, heterogeneous systems with resource-constrained nodes [117, 8].

The need to postpone the replay of traces #6-7 can also be detected using logical timestamps such as vector clocks [117, 8]. However, their size and thus also the size of the messages typically grows with the size of the system. Despite optimizations, the costs of logical timestamps can be significant [181]. To satisfy Requirement 2, our
5.3. DESCRIPTION OF APPROACH

approach does not use timestamps.

5.2.1 Totally versus Partially Ordered Traces

There are two types of ordering: (1) Total ordering that uses occurrence time as the most frequent approach for ordering the trace. So, by executing a distributed program multiple times, we may obtain various totally ordered traces depending on their occurrence time. (2) Partial ordering defined by the happened-before relation\textsuperscript{114} orders traces according to the causality relationship among them. For debugging or testing distributed systems, using a total ordering is inappropriate\textsuperscript{72}. Essentially, totally ordered traces impose an arbitrary ordering on independent traces. In the context of the running example, traces in lines #1, #2, and #3 are independent and may be received in an arbitrary order. However, total ordering approaches consider an ordering for all traces regardless of the fact that whether they are dependent to each other or not. In our work we propose a partial ordering approach without the use of logical timestamps.

5.3 Description of Approach

5.3.1 Overview

As stated before, existing approaches based on either physical or logical timestamp, essentially suffer from several limitations which may lead to inaccurate or inefficient replay of execution traces. Our approach does not use timestamps and instead relies on static analysis and replay of the models of behaviour.

A graphical overview of the implementation of our approach is given in Figure \[5.1\] and will be discussed in Section \[5.3.4\]. Our approach assumes that the traced system
5.3. DESCRIPTION OF APPROACH

Figure 5.1: An overview of MReplayer

has been instrumented and generates execution traces at runtime. Other high-level, conceptual characteristics of the approach include:

Leveraging replay

Replay will be used to recover relevant parts of the run-time state, i.e., the values of variables. As a result, this information does not need to be added to the traces. Also, rather than annotating traces with timestamps, we use the replay of the state machine execution to determine consistent orderings. For instance, consider transitions \( t_1 \) and \( t_2 \) in the state machine of \( PPD \). The destination state of \( t_1 \) is the source state of \( t_2 \). As a result, during the replay, \( t_2 \) will not be allowed to occur until \( t_1 \) has occurred at least once, and in the reordering the trace in line #1 of Listing 5.1 will always appear before the trace in line #4. Similarly, transition \( t_2 \) of \( BNK \) requires message con which is sent in transition \( t_2 \) of \( CTR \). As a result, transition \( t_2 \) of \( BNK \) will not be allowed to occur during the replay until \( t_2 \) of \( CTR \) has occurred, and the trace in line #6 of Listing 5.1 will always appear after the trace in line #8.
Leveraging rc-steps

In the context of tracing distributed systems, the execution semantics of state machines and rc-steps in particular, can be leveraged in the following ways:

a) Describing capsule behaviour: The set of rc-steps of a capsule fully describe its possible behaviours and thus can be used by the replayer instead of the, more verbose, state machine representation. In our approach, the set of rc-steps of each capsule is extracted by a static analysis (to be described in Section 5.3.2 below) that is performed before tracing starts. As long as the state machines do not change, this static analysis only has to be performed once.

b) Recognizing rc-steps: Suppose during the replay, capsule $C$ is in some configuration $\gamma$. Well-formedness condition C1 implies that an incoming message $m$ can enable at most one rc-step. Also, to recognize this rc-step, the replayer only needs the current configuration, the set of all possible rc-steps of $C$, and the trace of the first transition $t$ of the step. Thus, instead of instrumenting all transitions (as implied by Listing [5.1]), it suffices to instrument the first transition of each rc-step only. So, instead of receiving the traces in Listing [5.1], our replayer would receive the sequence of traces $T_{in}$ below in which, consistent with Definition [10], each trace identifies a transition in a capsule (with superscripts $C$, $P$, and $B$ referring to capsules $CTR$, $PPD$, and $BNK$ respectively). The sequence $T_{in}$ shows the corresponding sequence of rc-steps recognized by the replayer.

$$T_{in} = t^{P}_1 \ t^{B}_1 \ t^{C}_1 \ t^{P}_2 \ t^{B}_2 \ t^{C}_2 \ t^{C}_3$$

This allows us to reduce the amount of instrumentation that needs to be added to the traced system, the degree to which the traced system is slowed down during
execution, and the number of traces that need to be transmitted and processed.

c) Replaying rc-steps: Condition C2 means that the execution of an rc-step cannot get stuck. An incoming trace indicates that the execution of an rc-step has started. So, once a trace has been received and the rc-step it matches has been recognized, it is safe to assume that all remaining transitions of the step already have or will also be executed. Since capsules do not share state, when these remaining transitions of an rc-step \( rc \) are executed relative to the transitions of the rc-steps of other capsules, does not impact the configuration of the execution of \( rc \).

Thus, our replayer will replay an entire rc-step as soon as the trace of the first transition of \( rc \) has been received and recognized, even though, strictly speaking, the replayer has no evidence that the subsequent transitions really have been executed in the instrumented system. We know that these transitions will be executed and that executing them earlier or later will not change the resulting configuration.

d) Variable value synthesis: To synthesize variables values we create an interpreter for our action language, which is a subset of C++, using Antlr [149]. Antler is a powerful parser generator for building parse trees based on a grammar file. The action language includes the most basic imperative language constructs including integer, string and boolean expressions, variable assignment, if, while loop, blocks, send command and sequential composition of statements. For the sake of simplicity the action language does not have any input/output capabilities. Listing 5.2 presents some of the rules in the syntax of the action language that we use for re-generating variable values in our approach.

Listing 5.2: A part of the syntax for the action language
5.3. DESCRIPTION OF APPROACH

\( \text{statement} \) ::= \( \text{assignment} \) \\
| \( \text{if-stat} \) \\
| \( \text{loop-stat} \) \\
| \( \text{send-stat} \) \\

\( \text{assignment} \) ::= (‘int’|’bool’|’char’)? \( \langle \text{id} \rangle \) ‘=’ \( \langle \text{expr} \rangle \) ‘;’

\( \text{if-stat} \) ::= ‘if’ \( \langle \text{con} \rangle \) (‘else if’ \( \langle \text{con} \rangle \) )* (‘else’ \( \langle \text{stat-block} \rangle \))?

\( \text{loop-stat} \) ::= ‘while’ ‘(’ \( \langle \text{expr} \rangle \) ’)’ \( \langle \text{stat-block} \rangle \) \\
| ‘do’ \( \langle \text{stat-block} \rangle \) ‘while’ ‘(’ \( \langle \text{expr} \rangle \) ’)’ ‘;’ \\
| ‘for’ ‘(’ \( \langle \text{expr} \rangle \) ‘;’ \( \langle \text{expr} \rangle \) ‘;’ \( \langle \text{expr} \rangle \) ’)’ \( \langle \text{stat-block} \rangle \)

\( \text{send-stat} \) ::= \( \langle \text{id} \rangle \) ‘.’ \( \langle \text{id} \rangle \) ‘(’ \( \langle \text{expr} \rangle \)? ‘)’ ‘.send’ ‘(‘)’ ‘;’ \\
| \( \langle \text{id} \rangle \) ‘.’ \( \langle \text{id} \rangle \) ‘(’ \( \langle \text{expr} \rangle \)? ‘)’ ‘.sendAt’ ‘(‘\(\langle \text{expr} \rangle\)’)’ ‘;’

\( \text{con} \) ::= \( \langle \text{expr} \rangle \) \( \langle \text{stat-block} \rangle \)

\( \text{expr} \) ::= \( \langle \text{id} \rangle \) \\
| \( \langle \text{expr} \rangle \) (‘*’|’/’|’%’|’+’|’-’) \( \langle \text{expr} \rangle \) \\
| \( \langle \text{expr} \rangle \) (‘>’|’<’|’>=’|’<=>’|’==’|’!=’) \( \langle \text{expr} \rangle \) \\
| \( \langle \text{expr} \rangle \) (‘&&’|’||’) \( \langle \text{expr} \rangle \) \\
| (‘--’|’++’) \( \langle \text{expr} \rangle \) \\
| ‘!’ \( \langle \text{expr} \rangle \)

\( \text{id} \) ::= \[a - zA - Z_\_][a - zA - Z_0 - 9]^*
5.3. DESCRIPTION OF APPROACH

\[
\begin{align*}
text{(stat-block)} & ::= \{ \text{(block)} \} \nonumber \text{ \} } \\
& \quad | \text{(statement)} \\
text{(block)} & ::= \text{(statement)}^* \nonumber \\
\end{align*}
\]

5.3.2 Static Analysis

A capsule’s rc-steps are computed by function \texttt{computeRCSteps}, shown in Algorithm 1. It traverses the state machine using depth-first search and computes the set of all possible rc-steps starting from the initial state \(s_0\). For example, to extract all rc-steps of the USM of \texttt{SRV}, the function \texttt{computeRCSteps} should be called as

\[
\text{computeRCSteps}(SM, in_1, [], \emptyset)
\]

where \(SM\) and \(in_1\) denote the state machine of \texttt{SRV} and the initial state of that state machine, respectively. The function first extracts all outgoing transitions from \(s_0\) (line #1). Then, for each transition \(t\), all sequences of transitions that start with \(t\) and lead to a basic state will be added to \(P\) (lines #2-18). A set of visited states is used to avoid cycles (lines #7 and #16). Depending on the type of the target state of the transition \(t.des\), the traversal proceeds as follows:

\(1\) \textit{t.des is a basic state (line #4)}: the rc-step that started with transition \(t\) has ended, and the function adds the step to the set of rc-steps collected so far (lines #4-10).

\(2\) \textit{t.des is a pseudo-state (line #11)}: the path is still partial. Thus, the function is called recursively with the partial path and the next pseudo-state (lines #11-12).
Algorithm 1: computeRCSteps($SM : USM$, $s_0 :$ state, $P :$ sequence of transitions, $visited :$ set of states)

1. $T \leftarrow \{ t \in SM.T \mid t.src = s_0 \}$
2. **forall** $t \in T$ do
3.     $P \leftarrow append(P, t)$
4.     if $t.des \in S_b$ then
5.         $rcStep \leftarrow \langle P[0].src, P[0].trig, P, P[size(P) - 1].des \rangle$
6.         if $t.des \notin visited$ then
7.             $visited \leftarrow visited \cup \{ t.des \}$
8.             $rcSteps \leftarrow \{ rcStep \} \cup$ computeRCSteps($SM$, $t.des$, $\emptyset$, $visited$)
9.         else
10.            $rcSteps \leftarrow rcSteps \cup \{ rcStep \}$
11.     else if $t.des \in S_p$ then
12.         computeRCSteps($SM$, $t.des$, $P$, $visited$)
13.     else if $t.des \in S_c$ then
14.         $S \leftarrow children^1(t.des)$
15.         **forall** $s \in S$ do
16.             $visited \leftarrow visited \cup \{ s \}$
17.             computeRCSteps($SM$, $s$, $P$, $visited$)
18.         computeRCSteps($SM$, $t.des$, $\emptyset$, $visited$)
19. **return** $rcSteps$

(3) $t.des$ is a composite state (line #13): any transition to a composite state is assumed to end in an implicit history state inside the composite state. In this situation, the next state can be any of the states inside the composite state. Thus, the function extracts all states inside the composite state (line #14) and calls itself recursively for each of these contained states (lines #15-17). Finally, since a composite state can also be the source of a transition, the function is again called recursively with the $P$ argument set to the empty list, because a new rc-step starts (line #18).
5.3. DESCRIPTION OF APPROACH

5.3.3 Consistent reorderings

We now make the reordering task more precise by defining a couple of dependencies: one to capture control flow within a state machine and the other for communication between two state machines.

**Definition 11.** (Control flow dependency) There is a control flow dependency between two rc-steps $rc_1$ and $rc_2$, whenever the destination state of $rc_1$ is the source state of $rc_2$:

$$rc_1 \rightarrow_{CFD} rc_2 \iff rc_1.\sigma' = rc_2.\sigma$$

For CTR for example, $rc_1^C \rightarrow_{CFD} rc_2^C$ and for BNK, $rc_1^B \rightarrow_{CFD} rc_2^B$ and for PPD $rc_1^P \rightarrow_{CFD} rc_2^P$. Note that whenever there is a control flow dependency between two rc-steps, then these steps belong to the state machine of the same capsule.

**Definition 12.** (Communication dependency) There is a communication dependency between two rc-steps $rc_1$ and $rc_2$, whenever $rc_1$ sends a message that triggers $rc_2$:

$$rc_1 \rightarrow_{CMD} rc_2 \iff \exists \text{ capsules } c_1, c_2. \exists m \in \text{out}(c_1) \cap \text{in}(c_2). \text{send}(m) \in \text{actions}(rc_1) \text{ and } m \in rc_2.C$$

where $\text{in}(c)$ and $\text{out}(c)$ refer to the sets of messages that capsule $c$ can receive as trigger and produce as output, respectively. $\text{send}(m)$ denotes the action language statement used to send message $m$.

For example, for PPD and CTR, $rc_2^P \rightarrow_{CMD} rc_2^C$ and for BNK and CTR $rc_2^B \rightarrow_{CMD} rc_2^F$. Note that the sending and the receiving capsule in a communication dependency do not have to be different.

**Definition 13.** (Consistent reordering) Given an input trace $T_1$, we call $T_2$ a consistent reordering of $T_1$, iff (1) $T_2$ is a permutation of $T_1$, and (2) the sequence of
5.3. DESCRIPTION OF APPROACH

rc-steps $R_{T_2} = rc_1 \ldots rc_n$ corresponding to $T_2$ respects all control flow or communication dependencies, i.e.,

$$\forall 1 \leq i, j \leq n . rc_i \rightarrow rc_j \text{ implies } i < j$$

where $\rightarrow = \rightarrow_{CFD} \cup \rightarrow_{CMD}$.

E.g., $T_{out,1}$ below is a consistent reordering of the sequence $T_{in}$

$$T_{out,1} = t^P_1 t^B_1 t^C_1 t^P_2 t^C_2 t^B_2 t^C_3$$

5.3.4 MReplayer

An overview of the approach and our replay component $MReplayer$ is shown in Figure 5.1. When MReplayer starts, it creates abstract, simplified counterparts for each of the capsules in the system and connects them with an abstract controller. Possibly out-of-order traces emitted from the instrumented system are received by the controller and transmitted to the respective abstract capsule. Each of the abstract capsules will try to replay the execution steps corresponding to its incoming traces. For replay, each abstract capsule has a queue of incoming messages which can trigger transitions. Each abstract capsule also is provided with the set of rc-steps that its state machine can perform. Messages sent by a capsule during the replay of a transition are sent to the controller which will pass them on to their recipient capsules. A trace that has been successfully replayed by a capsule is fed into a sequence of ordered traces that the replayer outputs. The output traces will be consistent reorderings of the input sequence.
Abstract Controller

The behaviour of the abstract controller is rather simple. It listens for traces from the deployed system, and for each trace that arrives it uses the component id \( P \) contained in the trace (Definition 10) to pass on the trace to the abstract capsule representing the component that generated the trace. It also listens for messages generated by the abstract capsules and uses the recipient information that accompanies the message to relay that message on to the abstract capsule that is to receive it.

Abstract Capsule

An abstract capsule\(^1\) is created per capsule instance in the original system. Let \( C_a \) be the abstract capsule of capsule instance \( C \). Unlike \( C \), \( C_a \) does not have a state machine or ports. Instead, \( C_a \) has access to the set of rc-steps the state machine of \( C \) can perform (called rcSteps), a first-in-first-out (FIFO) queue for receiving traces (inTraces), and a FIFO queue that maintains all incoming messages (msgs). It also maintains the currently active configuration of \( C \) (Definition 7). As sketched above, \( C_a \) will replay every incoming trace, while postponing the replay whenever necessary.

To do so, function \texttt{replayRCSteps}(s\(_0\), rcSteps, inTraces, msgs, outTraces) is used as shown in Algorithm 2. As a first step, \texttt{replayRCSteps} will initialize the configuration \( \gamma \) with a default configuration \( \gamma_0 \) where initial state is assigned to its active state, as well as default values are assigned to all the variables and attributes of its value map, i.e., \( \gamma.E \). Then, the message queue, i.e., msg is reset with the initial message \texttt{startUp} in line \#4. In the line \#5, the \texttt{while} loop checks if all traces in \texttt{inTraces} not consumed. In the line \#7, the function \texttt{getRCStep}(msg,rcSteps,\( \gamma \)) is used to find

\(^1\)Strictly speaking, it should be called abstract capsule instance
the matching rc-step, i.e., \textit{rcStep} with respect to \textit{msg}, list of rc-steps, i.e., \textit{rcSteps} and the current configuration, i.e., \textit{γ}. Well-formedness condition \textit{C1} (ref. Section 2.4) implies that an incoming message \textit{msg} can enable at most one rc-step. The function \textit{getMatchingTrace} in the line #8 is used to find the matching \textit{trace} using the current rc-step, i.e., \textit{rcStep}, in \textit{inTraces}. Then, the function \textit{replay} is used to replay \textit{rcStep} and create a new configuration, i.e., \textit{γ} (line #9). According to the well-formedness condition \textit{C2} the execution of an rc-step cannot get stuck. Finally, the matching trace is appended to the sequence of outgoing traces, i.e., \textit{outTraces}, and remove the matching trace from \textit{inTraces} (line #10).

Note that the initial transition of a state machine does not have guard and contains a special trigger \textit{startUp} which we always consider to be present. Thus, a trace representing the execution of the initial transition of capsule \textit{C} will always match and be enabled when \textit{C_a} is started up (and, possibly, \textit{msgs} is still empty), and \textit{replayRCSteps} is first invoked.

Compared to existing work, our approach has the following distinguishing characteristics:

\textbf{Leverage abstraction and automation with MDD:} MDD is a software development approach that promotes the use of models as the primary software development artifact \[162\] \[151\]. In our work code is generated from models using an open-source MDD tool \[106\] such that it can be executed on different nodes in a distributed system. The behaviour of components is described using communicating state machines.

\textbf{Reduce number and size of traces by avoiding timestamps:} To sidestep the above-mentioned costs associated with timestamps and achieve requirement R2, our approach does not use timestamps. Also, the inclusion of variable values in
traces can largely be avoided. Instead, we use the state machine models to determine inconsistent trace orderings and to replay the relevant parts of the execution to recover execution state.

5.3.5 Example

Figure 5.2 shows an execution of the replayer on the input trace $T_{in}$ (ref. Section 5.3.1). Initially, all abstract capsules wait for a trace to match ($status=wait_{inT}$). All state machines are in their initial states ($in_{i1}$). In Step 0, the message initial $startUp$ is in the queues of all capsules. In Step 1, input trace $t_{1}^{P}$ is processed, and the rc-step $rc_{1}^{P}$ is replayed by the (abstract capsule of) $PPD$ which leads to state $s_{1}$. Similarly for Step 2 and 3, input traces $t_{1}^{B}$ and $t_{1}^{C}$ are processed, and their corresponding rc-steps $rc_{1}^{B}$ and $rc_{1}^{C}$ are replayed by $CTR$ and $BNK$ which put them to the state $s_{1}$. In Step 4, input $t_{2}^{P}$ is matched to $rc_{2}^{P}$ which is also found enabled according to Definition 9 and thus can be replayed, causing $PPD$ to move to state $s_{2}$, message $userReq$ be sent to $CTR$, and $t_{2}^{P}$ to be output. In Step 5, input $t_{2}^{B}$ is not matched and remains in the input sequence. In Step 5, input $t_{2}^{C}$ can be matched to $rc_{2}^{C}$ which is also found enabled and thus can be replayed, causing $CTR$ to move to state $s_{2}$, message $con$ be sent to $BNK$, and $t_{2}^{C}$ to be output. In Step 5, input $t_{2}^{B}$ is checked again and matched to $rc_{2}^{B}$ which is also found enabled and thus can be replayed, causing $BNK$ to move to state $s_{2}$, message $ack$ be sent to $CTR$, and $t_{2}^{B}$ to be output. Finally, in Step 9, input $t_{3}^{C}$ can be matched to $rc_{3}^{C}$ which is also found enabled and thus can be replayed, causing $CTR$ to move to state $s_{3}$, message $ok$ be sent to $PPD$, and $t_{3}^{C}$ to be output.

Note: in Figure 5.2 columns $inT$ and $rcStep$ show the input trace processed
### 5.3. DESCRIPTION OF APPROACH

<table>
<thead>
<tr>
<th>Step</th>
<th>$in_T$</th>
<th>rc$step$</th>
<th>$PPD$</th>
<th>$CTR$</th>
<th>$BNK$</th>
<th>$out_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>$wait_{inT}$</td>
<td>$in_1$</td>
<td>$\langle$startup$\rangle$</td>
<td>$wait_{inT}$</td>
</tr>
<tr>
<td>1</td>
<td>$t_1^P$</td>
<td>$rc_1^P$</td>
<td>$replay(rc_1^P)$</td>
<td>$s_1$</td>
<td>$\langle\rangle$</td>
<td>$wait_{inT}$</td>
</tr>
<tr>
<td>2</td>
<td>$t_1^B$</td>
<td>$rc_1^B$</td>
<td>$wait_e(init)$</td>
<td>$s_1$</td>
<td>$\langle\rangle$</td>
<td>$wait_{inT}$</td>
</tr>
<tr>
<td>3</td>
<td>$t_1^C$</td>
<td>$rc_1^C$</td>
<td>$wait_{inT}$</td>
<td>$s_1$</td>
<td>$\langle init \rangle$</td>
<td>$replay(rc_1^C)$</td>
</tr>
<tr>
<td>4</td>
<td>$t_2^P$</td>
<td>$rc_2^P$</td>
<td>$replay(rc_2^P)$</td>
<td>$s_2$</td>
<td>$\langle\rangle$</td>
<td>$wait_e(userReq)$</td>
</tr>
<tr>
<td>5</td>
<td>$t_2^B$</td>
<td>$rc_2^B$</td>
<td>$wait_e(ok)$</td>
<td>$s_2$</td>
<td>$\langle\rangle$</td>
<td>$wait_{inT}$</td>
</tr>
<tr>
<td>6</td>
<td>$t_2^C$</td>
<td>$rc_2^C$</td>
<td>$wait_e(ok)$</td>
<td>$s_2$</td>
<td>$\langle\rangle$</td>
<td>$replay(rc_2^C)$</td>
</tr>
<tr>
<td>7</td>
<td>$t_3^B$</td>
<td>$rc_3^B$</td>
<td>$wait_e(ok)$</td>
<td>$s_2$</td>
<td>$\langle\rangle$</td>
<td>$wait_{inT}$</td>
</tr>
<tr>
<td>8</td>
<td>$t_3^C$</td>
<td>$rc_3^C$</td>
<td>$wait_{inT}$</td>
<td>$s_2$</td>
<td>$\langle ok \rangle$</td>
<td>$replay(rc_3^C)$</td>
</tr>
</tbody>
</table>

Figure 5.2: Sample Execution of MReplayer on ATM using the input trace $T_{in}$
5.3. DESCRIPTION OF APPROACH

Algorithm 2: \texttt{replayRCSteps}(s_0: \text{initial state of } C, \text{rcSteps}: \text{set of rc-steps of } C, \text{inTraces}: \text{sequence of incoming traces of } C, \text{msgs}: \text{sequence of messages for } C, \text{outTraces}: \text{sequence of outgoing traces of } C)

1. Let $\gamma$ range over configurations (Definition 7)
2. Let \texttt{trace}, \texttt{rcStep}, and \texttt{msg} range over execution traces, rc-steps, and messages respectively (Definitions 10, 6, and 5)
3. $\gamma \leftarrow \gamma_0$
4. \texttt{msgs} $\leftarrow \texttt{append}(\texttt{msgs}, \texttt{startUp})$
5. \textbf{while} (\texttt{inTraces} not consumed) \textbf{do}
6. \texttt{msg} $\leftarrow \texttt{dequeue}(\texttt{msgs})$
7. \texttt{rcStep}, $\gamma_1$ $\leftarrow \texttt{getRCStep}(\texttt{msg}, \texttt{rcSteps}, \gamma)$
8. \texttt{trace} $\leftarrow \texttt{getMatchingTrace}(\texttt{inTraces}, \texttt{rcStep})$
9. $\gamma \leftarrow \texttt{replay}(\texttt{rcStep}, \gamma_1)$
10. \texttt{outTraces} $\leftarrow \texttt{append}(\texttt{outTraces}, \texttt{trace})$

and the rc-step matched in this step, respectively. For each abstract capsule, \texttt{status} indicates what the capsule is doing where \texttt{wait}_e(m) = ‘waiting for message m’ (line #6), \texttt{wait}_{inT} = ‘waiting for a match in the input traces’ (line #8), and \texttt{replay}(rc) = ‘replaying rc-step rc’ (line #9) in Algorithm 2. Columns \texttt{s} and \texttt{msgs} indicate the new state and the sequence of incoming messages. \texttt{outT} shows the output trace in each step.

5.3.6 Automatic Instrumentation

Our approach mostly relies on a model instrumentation process introduced in MDebugger [23]. The main difference is support for TCP/IP network communication that enables each instrumented component to send traces to MReplayer.

Our automatic instrumentation uses model-to-model transformation techniques to create an instrumented version of the user-defined model, allowing the generated code to emit execution traces at runtime. As mentioned, the instrumented system generates an execution trace for the first transition of every rc-step of a capsule.
5.3.7 Extensions

Non-deterministic action code

So far, we have assumed that all action code statements are deterministic so that the replay of an rc-step is guaranteed to produce the same result as during system execution. However, this assumption is unrealistic, because the behaviour of models often relies on unmodeled external input in the form of, e.g., user input, timer values, or file contents. Statements reading this input cannot be replayed deterministically.

To deal with non-deterministic statements, we extend our approach to allow for the inclusion of variable values in traces. More specifically, a value map is added to an execution trace (Definition 10) that is used to track the values of variables that depend on non-deterministic statements.

Generating all possible consistent trace orderings

As described so far, our approach outputs a single trace sequence only. That sequence may only be one of many consistent reorderings. For instance, apart from $T_{out,1}$, the sequences below also are other consistent reorderings of the input sequence $T_{in}$.

\[
T_{out,2} = t_1^B \ t_1^P \ t_1^C \ t_2^P \ t_2^C \ t_2^B \ t_3^C \\
T_{out,3} = t_1^C \ t_1^P \ t_1^B \ t_2^P \ t_2^C \ t_2^B \ t_3^C \\
T_{out,4} = t_1^P \ t_1^C \ t_1^B \ t_2^P \ t_2^C \ t_2^B \ t_3^C \\
T_{out,5} = t_1^B \ t_1^C \ t_1^P \ t_2^P \ t_2^C \ t_2^B \ t_3^C \\
T_{out,6} = t_1^C \ t_1^B \ t_1^P \ t_2^P \ t_2^C \ t_2^B \ t_3^C
\]

To extend our approach to output all consistent reorderings, the static analysis
phase discussed in Section 5.3.2 is modified to extract the control flow and communication dependencies. After Algorithm 2 has found one consistent reordering, the remaining reorderings can be enumerated.

5.4 Tool Support

We have developed MReplayer that embodies our approach and supports replay of execution traces collected from a distributed systems with possibly several nodes. Our work also benefits from MDebugger’s instrumentation [24] and Epsilon Object Language (EOL) [109] implements the transformation rules required for instrumenting an original model into an instrumented model. EOL supports a set of instructions to create, query, and modify models. The part for the execution of action code in the model during a replay is implemented using C++, ANTLR [149]. In this section we discuss MReplayer features for from a user point of view.

5.4.1 MReplayer Features

MReplayer is comprised of a set of engines (i.e., Instrument, Trace Collect, PreProcess, Webserver, and Replay) that can be executed and monitored via a simple, yet powerful control panel as shown in Fig. 5.4. In the following, we discuss the main

---

2A video that demonstrates the tool: [https://youtu.be/WG5ggqPoJHg](https://youtu.be/WG5ggqPoJHg)
features of MReplayer from the end-user point of view. Source code of the implementation along with documentation is available at [19, 75].

**Initial set-up** First of all, the user should choose a model that is used for replay via the first **browse** button (i.e., 1). The user then needs to specify a name for the instrumented model that is generated as an output of the automatic instrumentation (i.e., 2), as well as a name for the trace file (i.e., 3). Moreover, the code generation that is used by MReplayer is integrated into Papyrus-RT as an Eclipse plug-in that can be downloaded and installed from the **dRTS Repository** [106]. MReplayer’s control panel manages these tasks seamlessly without distracting the end user.

**Running the engines** In the **Executable Engines** section of MRepalyer’s control panel, the user can find a set of engines that allows running each part of the system separately. The user can execute model transformation scripts by clicking on the button **Instrument** (i.e., 4). Once the instrumentation is completed, the user can click on the button **Trace Collect** (i.e., 5) in order to establish a TCP socket connection on
port 8001 for collecting traces. After that, PreProcess (i.e., 6) should be executed in which the static analysis is performed on the model to extract all run-to-completion steps. The next one is MReplayer’s webserver (i.e., 7). Finally, the user can select one consistent ordering from the list (i.e., 9) to replay using the button Replay (i.e., 8). To avoid the confusion which may arise from different orders that a user can run the engines, MReplayer activates each engine’s button only when all the previous engines have been started successfully. Also, during this process, useful information is shown in the textbox (i.e., 10) regarding the status of the engines, whether they have started successfully or failed.

**View active transition(s) and the current execution state.** Usually, viewing the current state of a program and its components at runtime is the preliminary step to start debugging, especially for distributed systems. MReplayer facilitates observability of the system at runtime by highlighting active transitions in red color (e.g., 6) and current states in green color (e.g., 4).

**Control over the execution steps.** The capability of traversing freely through the execution history is crucial for the root-cause analysis. MReplayer allows developers to exert control over replaying the traces in both forward and backward direction in an execution path via using the Replay Panel (i.e., 7). From the left to the right the actions are step-back, run, stop and step-forward.

**View current values of variables.** Inspecting variable values is an effective approach for debugging an application. The Inspector Panel (i.e., 8) allows developers to inspect the values of variables conveniently. Sometimes the root of a failure is not manifested in the same memory stack. In fact, in some debugging approaches such as omniscient debugging [38], we need to go back in time and retrieve previous values...
of a variable to identify the root of a failure. To fulfill this demand, MReplayer keeps track of previous values of a variable using an efficient map data structure that maps a memory stack to its variables. Consequently, the Inspector Panel is updated in both forward and backward replay of an execution.

**View message(s) exchange.** Connectors are used in UML-RT models to exchange messages between capsules. Also, sending a message either with or without data from a source capsule would trigger a certain transition in the target capsule. Therefore, this could be a potential point of failure where the target capsule has received an unexpected message or incorrect data. MReplayer facilitates identifying such failure by highlighting the active connectors and annotating them with their current messages. Once a capsule sends a message to another capsule, the connector between two capsules is highlighted into blue, and also, the message’s name and its data are shown on the corresponding connector (e.g., 5).

**View the trace-sizes.** In several resource-constrained distributed systems such as IoT devices, the size of collected traces is important to avoid memory overflow. To comply with this demand, MReplayer illustrates a line chart (e.g., 1) showing the cumulative size of collected traces (e.g., 3) using a line chart. Also, we make an online comparison with traces generated by MDebugger (e.g., 2). The gap between these two lines illustrates the memory efficiency gained from our approach.

5.5 **Experimental Evaluation**

This section details experiments we conducted to assess the performance, benefits, and overhead of our approach. In the following, we describe our prototype implementation, case studies, experimental protocol, hypotheses, and results.
5.5. EXPERIMENTAL EVALUATION

5.5.1 Hypotheses

We formulate the following four hypotheses to assess the performance of our approach in comparison to trace replaying solutions based on logical timestamps.

**Hypothesis 1: Processing Time.** It is important that processing time of static analysis and instrumentation of our approach remain acceptable even when model size grows. We hypothesize that performing these steps on models with various complexities can be done within a reasonable amount of time.

**Hypothesis 2: Size of Code.** We hypothesize that the number of lines of code added by the instrumentation does not differ significantly from those added by similar approaches based on logical timestamps.

**Hypothesis 3: Size of Traces.** We hypothesize that our approach is efficient with respect to the size of generated traces in comparison to similar approaches that use logical timestamps.

**Hypothesis 4: Runtime Overhead.** Generation of execution traces in the instrumented application imposes runtime overhead. We hypothesize that runtime overhead caused by the instrumentation in our approach is reasonable.

5.5.2 Verification Approach

To evaluate our approach, we use some of the case studies discussed in Chapter 4. Indeed, none of these models are big enough to demonstrate every characteristic of a real-world distributed system. Nonetheless, small models such as SCMS, CDL and SPR are useful to show the performance of our approach where there is no source of non-determinism in the model. Also, some of them, e.g., FO and RFO that use a timer in their implementations are useful to show the performance of our approach in
Table 5.1: Complexity of Case Studies, Processing Time, Code Size Overhead, and Size of Collected Traces

<table>
<thead>
<tr>
<th>Model</th>
<th>Complexity</th>
<th>Inst. (ms)</th>
<th>Stat. (ms)</th>
<th>LOC</th>
<th>SOT (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ours</td>
<td>VT</td>
<td>Ours</td>
<td>VT</td>
</tr>
<tr>
<td>SCMS</td>
<td>3</td>
<td>17</td>
<td>23</td>
<td>1115</td>
<td>1967</td>
</tr>
<tr>
<td>CDCL</td>
<td>5</td>
<td>11</td>
<td>15</td>
<td>698</td>
<td>995</td>
</tr>
<tr>
<td>SPR</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>926</td>
<td>1896</td>
</tr>
<tr>
<td>DP</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>715</td>
<td>1101</td>
</tr>
<tr>
<td>PR</td>
<td>8</td>
<td>14</td>
<td>25</td>
<td>1023</td>
<td>2337</td>
</tr>
<tr>
<td>RP</td>
<td>3</td>
<td>40</td>
<td>71</td>
<td>1113</td>
<td>2201</td>
</tr>
<tr>
<td>RO</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>980</td>
<td>2138</td>
</tr>
<tr>
<td>FO</td>
<td>7</td>
<td>31</td>
<td>43</td>
<td>1150</td>
<td>2625</td>
</tr>
<tr>
<td>RFO</td>
<td>13</td>
<td>2304</td>
<td>3647</td>
<td>2896</td>
<td>36033</td>
</tr>
</tbody>
</table>

C: capsules, S: states, T: transitions, Inst.: instrumentation time, Stat.: static analysis, LOC: lines of code, SOT: size of collected traces, Ours: our approach, VT: vector time approach. The abbreviations under 'Model' refer to the case studies presented in Chapter 4.

the presence of a variable that can cause non-deterministic behaviour in the system.

Experimental setup

In the following, we describe the metrics and the experiments used to verify our hypotheses.

Processing Time of Model Instrumentation and Static Analysis (EXP-1). We performed each step 20 times for each case study listed in Table 5.1 and averaged the times required for each step (i.e., instrumentation and static analysis) and each case study combination.

Collection of Execution Traces (EXP-2). We extended our prototype to implement the classical vector time approach. We then instrumented the case studies in Table 5.1 using our approach as well as the vector time approach. We ran each version using identical deployment configurations and collected 100,000 execution traces. We
measured the total size of the traces in all cases.

**Measuring Runtime Overhead on Instrumented Application (EXP-3)** Using both versions of the FailOver system from EXP-2, we evaluated the performance of the uninstrumented FailOver system (i.e., normal execution), the instrumented FailOver system using our approach, and the vector time approach. Then, based on the execution traces, we collected the computation time required for replying to a server request (i.e., transition RequestReply), processing a message response by a client (i.e., transition ProcessingResponse), and notifying each server’s peers of its availability (i.e., transition SendKeepAlive). These transitions are triggered more often than other transitions and contribute most to the execution time of the model. In all cases, we executed the system until 1,000 traces had been collected from each of these transition.

**Measuring Application Size Overhead (EXP-4)** For each case study listed in Table 5.1, we generated the code from both the original and instrumented models using our approach and vector time approach. Then we calculated the overhead of the instrumentation method in our approach and vector time approach.

**Rationale for comparison with vector time.** Similar to logical time approaches, our approach determines the causality between traces. However, to achieve this, we use replay on abstract models as opposed to the annotation of traces with logical timestamps. There are many logical time approaches and it is not feasible to perform comparisons with all of them. We selected vector time because it is a mature, representative logical time approach with strong formal foundation. Other researchers have made similar comparisons in order to assess and position their work, e.g., [113].

**Experimental Environment.** We used a computer equipped with a 2.7GHz Intel
5.5. EXPERIMENTAL EVALUATION

Core i5 and 8GB of memory for experiments EXP-1 and EXP-4. For the experiment EXP-2 and EXP-3 we used 5 virtual machines with a 1.7GHz Intel Core i3 and 4GB. We used Java version 1.8.0161 in configuration -Xmx12512m.

5.5.3 Results and Discussions

In the following, we present the results of our experiments and discuss their impact on our hypotheses.

**Hypothesis 1: Processing Time.** Based on EXP-1, the instrumentation time (Inst.) and static analysis (Stat.) columns of Table 5.1 show the time required to instrument the models and compute the rc-steps respectively. In the worst case (i.e., the largest model), the instrumentation only takes around 3 seconds and static analysis takes 36 seconds. Going from FailOver system to Refined FailOver system, the number of capsules, states, and transition increase by factors 1.8, 74, and 84, respectively; however, instrumentation and static analysis times only increase by factors 2.5 and 14, respectively. While processing times increase with model size, the
Figure 5.6: RequestReply

Figure 5.7: SendKeepAlive
5.5. EXPERIMENTAL EVALUATION

experiment provides some evidence to conclude that they are reasonable and do not grow exponentially. We thus consider Hypothesis 1 verified.

**Hypothesis 2: Size of Code.** The LOC column of Table 5.1 shows the overhead of instrumentation methods in terms of lines of code using EXP-4. The percentage of overhead ranges between 8\% and 20\% for our approach. In all cases, overhead of our approach is less than the overhead imposed by the vector time approach, because our approach generates traces only from the transitions that identify a re-step, while the vector time requires not only traces from all transitions, but it also requires additional code to compute vector times. We conclude that the size overhead of our approach is acceptable. Thus, the Hypothesis 2 is verified.

**Hypothesis 3: Size of Traces.** The SOT column of Table 5.1 shows the size of the traces generated according to EXP-2. As discussed earlier, since the vector time approach annotates traces with timestamp and variable values, it increases the size of generated traces. Table 5.1 shows the size of traces generated using the vector time approach is around 2.7 times the size of the traces generated by our approach. This verifies Hypothesis 3.

**Hypothesis 4: Runtime Overhead.** Figure 5.5, Figure 5.6, and Figure 5.7 show violin plots of computation times for three transitions (i.e., RequestReply, ProcessingResponse, SendKeepAlive). The wideness bars show the density of computation time in the specific range. As shown in Figure 5.5, Figure 5.6, and Figure 5.7, for all three transitions the system performance is impeded by the use of our approach. In addition, in our approach the majority of the ProcessResponse transitions are processed within 0.3ms to 0.48ms, with an average time of 0.4ms and a median time of 0.38ms, which is close to the processing time when the system is in normal mode (average and
median times of 0.34 and 0.31ms respectively). Thus, the overhead for the Process-
Response transitions is within the order of microseconds which we consider negligible.
Moreover, the overhead is similar for RequestReply and SendKeepAlive transitions.
While the median and average of computation time for RequestReply is 44.38ms and
42.22ms, respectively, using our approach, the median and average in normal mode
are 40.79ms and 40.38ms. For the SendKeepAlive transitions, the median and aver-
age using our approach are 0.075ms and 0.89ms, respectively, and in normal mode
are 0.54ms and 0.38ms, respectively. In summary, we conclude that the experiment
provides evidence that for each transition the overhead of our approach is small and
acceptable for many distributed systems. We consider Hypothesis 4 verified.

5.5.4 Threats to validity

The implementation of our approach is not trivial and our prototype may have bugs.
To mitigate this possibility, we have reused mature tools and frameworks such as
the Eclipse Modeling Framework [76] for implementation whenever possible. We also
have tested our prototype thoroughly on several case studies.

The set of case studies used for evaluation could be an other threat to validity.
In fact, some of the case studies are not models of distributed systems. We have
mitigated this threat by using these models only for the evaluation of static analysis
and model instrumentation, where the distribution has no impact.

Finally, part of our evaluation is a comparison with the vector time approach.
Since no implementation of vector time in the context of UML-RT state machines
was available, we implemented our own. While we did our best to implement the
vector time approach optimally, there might be room for further optimization of our
implementation of the vector time approach.

5.6 Summary

In this chapter, we proposed an efficient reordering and replay of execution traces of distributed systems in the context of model-driven development. Compared with existing approaches that employ either physical or logical timestamps for reordering execution traces, our approach relies on performing static analysis on the behaviours of each component and then replay of traces on centralized image of the distributed system. As a consequence, our approach enables a reduction of the amount of traces and instrumentation required which helps keep runtime overhead low. It is therefore more efficient especially for resource-constrained distributed systems that may have low CPU clock rate. Along with the approach, we implemented a prototype called MReplayer. It supports most of the common basic debugging/testing features such as step-in, step-over, and step-back. Also, it can be used via a web-based graphical user interface we developed as an extension of Draw.io [1]. We demonstrated the performance of our approach for replaying execution traces by conducting an empirical evaluation on a set of models. The experiments showed that our approach decreases the size of the traces significantly at the expense of slight performance overhead due to the interpretation of abstract models.
Chapter 6

Model-Level Replay-based Regression Testing for Distributed Systems

6.1 Introduction

Regression testing is defined as a type of software testing to confirm recent changes have not adversely affected existing features. But it can be extremely costly \cite{66} and ineffective \cite{77} especially for testing resource-constrained distributed systems (e.g., IoT) with possibly many nodes. For example, large software companies such as Google have over 100 million tests running each day on massive clusters of powerful machines which may produce large log files of collected traces and consume a lot of memory \cite{66}.

Testing by replay is a popular testing mechanism that enables developers to execute recorded traces of a system in a repeated and deterministic manner and make diagnostic observations, e.g., \cite{16, 183, 115, 108, 164, 120}. Despite recent progress in \textit{replay-based regression testing} (RRT) techniques and tools, e.g., \cite{5, 93, 169}, it turns out they are inadequate for testing distributed systems. As such, a typical distributed system with a few nodes can produce a large amount of (possibly out of
order) traces only after a few minutes of execution [166]. Therefore, an effective RRT for distributed systems may require efficient mechanisms for replaying and regression detection [148].

In this chapter, we present *MRegTest*, a replay-based regression testing tool for distributed systems at the model-level. We have built MRegTest on MReplayer [17] which is our proposed method for reordering and replaying traces in distributed systems without the use of timestamps (ref. Chapter 5).

MRegTest inherits from MReplayer a reduction in the number and size of traces required compared to standard, timestamp-based approaches. Conceptually, MRegTest benefits from the increased level of abstraction together with strategic semantic simplifications (such as the ‘run-to-completion’ assumptions for state machines) that a model-based system description can offer. Concretely, we discuss how the use of communicating state machines to describe a distributed system can be leveraged for a significant reduction in the number of base model executions that a replay-based regression testing requires to replay for detecting a regression. Our approach consists of the following steps:

- **Critical Variable Identification (CVI)** that allows the user to adjust the level of granularity of the regression tests as appropriate.

- **Execution Selection (ES)** that reduces the number of base model executions\(^2\) that need to be replayed.

- **Regression Detection (RD)** that uses an extension of MReplayer to determine: (a) whether the replacer is able to replay traces collected from the

\(^1\)Referring to an original model whose state machine has not been changed

\(^2\)Traces collected from executions of an instrumented base model
base model executions, (b) whether the value of critical variables are consistent between a base model and its modified model.

Apart from developing an effective regression testing approach for model-based distributed systems our work was guided also by the following requirements:

**Requirement R1**: The approach leverages existing modeling techniques.

**Requirement R2**: The approach aims to detect regressions while reducing the resource requirements on the nodes in the system.

There are three main architectures of regression testing for distributed systems: (1) Local testers that use a testing agent on each nodes e.g., [184, 89, 44], (2) Centralized testers that connect all nodes in the system to a centralized tester e.g., [176, 195, 102] or (3) Hybrid approaches of local and centralized testers, e.g., [122]. The pros and cons of each architecture have been discussed in [28]. We base our work on the centralized test architecture, since unlike local and hybrid testers that typically introduce an additional network overhead for exchanging messages, this architecture maintains a centralized testing agent for possibly many nodes which is more suitable for testing resource-constrained IoT systems [180, 130].

Our approach achieves R1 by leveraging abstraction and automation with Model-Driven Development (MDD). We use an open-source MDD tool Papyrus-RT [75, 106] that allows us to generate code from models, such that it can be executed on different nodes in a distributed system. Also, in our approach the behaviour of components is described using communicating state machines.

To achieve R2, we extend the MReplayer tool described in [17, 18]. MReplayer provides an efficient deterministic replay of traces collected from a distributed system without the use of timestamps. Also, variable values are re-generated during a replay
which means that they don’t have to be included in the traces. Together, both characteristics allow for a significant reduction in the size and number of traces. In fact, only those variables values are added to a trace that cannot be regenerated during a replay (e.g., values depended on random number generators and local resources such as files). Similar to MReplayer, MRegTest takes advantage of run-to-completion (RTC), an often made atomicity assumption that greatly simplifies dealing with concurrency in distributed systems.

To assess MRegTest, we conducted an empirical study based on the evaluation framework for regression testing techniques proposed by Rothermel et al. in which we applied our approach to detect regressions in modified models with various levels of complexity. Our experiments show performing each step of our approach, i.e., Execution Selection (ES) and Regression Detection (RD) even on large models (i.e., models with more than 2000 states and 3000 transitions) takes less than a minute. Also, using our approach the size of traces collected from each case study is reduced significantly by an average factor of 1.56× compared to traditional approaches that annotate traces with timestamps and variable values, e.g., [83, 164]. Finally, we evaluated runtime overhead of our approach against models with different complexities. The results show that the extra runtime overhead imposed by re-generating variable values by replay in our approach is within a reasonable range.

The rest of this chapter is organized as follows. Section 6.2 presents a running example. Section 6.3 describes our approach. The evaluation and experimental results are discussed in Section 6.5, and finally threats to validity and limitations are summarized in Section 6.6.
6.2. Running Example

In this section we use the ATM model (ref. Chapter 2) and apply a simple modification in the CLI’s USM shown in Figure 6.1. Then we will use our proposed solution to determine whether this modification will cause a regression or not.

Since nondeterminism is pervasive in distributed systems, traces might arrive at a centralized regression testing system in an arbitrary order. E.g., the arrival of
a trace corresponding to the transition \( t_2 \) in \( CTR \) at a tester might precede one corresponding to \( t_2 \) in \( PPD \). Therefore, the use of an efficient record and replay mechanism is essential. To satisfy this requirement, we employ our trace reordering method, i.e., MReplayer \( [17] \) that offers a low overhead deterministic replay of traces. For example, \( E_1, E_2, \) and \( E_3 \) shown in Figure 6.2 are three sample executions of the ATM system received by a centralized tester. Each execution consists of a set of traces from components of the ATM system: \( BNK \) (i.e., \( B \)), \( CTR \) (i.e., \( C \)), and \( PPD \) (i.e., \( P \)). For example, \( t^P_1 \) denotes the trace \( t_1 \) in the component \( PPD \).

Let’s consider a modification scenario in the component \( CTR \) that is shown in Figure 6.1. Suppose in this component the variable balance amount (i.e., \( bamt \)) is a critical variable and a developer modifies action code of \( t^C_9 \) such that \( amt + 10 \) is sent as an argument in the \( wdw \) message to \( BNK \) instead of \( amt \). This slight change will affect the expected behaviour of the system. For example, once a user with \$100 \) initial balance amount (i.e., \( bamt = \$100 \)) wants to withdraw \$10 from its account, the component \( CTR \) makes a withdrawal request in the transition \( t^P_5 \), and sends the value of \( amt \) along with the message \( wdw \) to \( BNK \). According to this scenario, \( BNK \) is expected to send \$90 as the result of this operation back to \( CTR \), but because of this modification it sends \$80 that is directly assigned to the variable \( bamt \). We consider this modification a regression, since the value of this critical variable differs from the one produced by the base model.

Similar to MReplayer in Chapter 5, the initial transition of a state machine does not have guard and contains a special trigger \( startUp \) which we always consider to be present.
6.2. RUNNING EXAMPLE

6.2.1 Leveraging MReplayer

We leveraged MReplayer [17] to facilitate reordering traces collected from a distributed system. MReplayer replays traces on an abstract version of the distributed system in order to keep track of the execution. As discussed in Section 6.3, an abstract version of the system consists of a set of abstract capsules that are connected to an abstract controller that acts as a communication gateway for delivering messages between abstract capsules. Apart from reordering traces, MReplayer reduces the size of traces significantly. Essentially, because it removes timestamps from traces, as well as it only annotates traces with variable values that cannot be reproduced by replay of traces.

\[
E_1 = t_1^P t_1^B t_1^C t_2^P t_2^C t_3^B t_3^C t_6^P
\]

\[
E_2 = t_1^P t_1^B t_1^C t_2^P t_2^C t_3^B t_3^C t_5^P t_5^B t_6^C t_11^C
\]

\[
E_3 = t_1^P t_1^B t_1^C t_2^P t_2^C t_3^B t_3^C t_6^P t_6^B t_10^C
\]

Figure 6.2: Sample Executions of the ATM system

Leveraging Run-to-completion (RTC): As discussed in the chapter 2. RTC is a central concept in the definition of the execution semantics of many state machines. An rc-step allows us to group all execution steps into a single RTC (i.e., macro) step.

Some of the rc-steps of the state machines in the ATM system are shown in Table 2.1. According to these rc-steps, we can produce \( rc_{E_2} \) that contains a sequence of rc-steps corresponding to traces in \( E_2 \) in Figure 6.2.

\[
rc_{E_2} = rc_1^P \cdot rc_1^B \cdot rc_2^C \cdot rc_2^P \cdot rc_2^B \cdot rc_3^C \cdot rc_3^P \cdot rc_3^B \cdot rc_5^C \cdot rc_5^B \cdot rc_6^C
\]

Moreover, we will execute an rc-step according to Definition 9. For example, we execute \( rc_3^C \) in the abstract capsule CTR only if the active state is \( s_2 \), the message...
ack exists at the head of the message queue of $C$, and accept is evaluated \textit{true} in the value map of the configuration $\gamma$.

6.3 Description of Approach

6.3.1 Overview

A graphical overview of the implementation of our approach is shown in Figure 6.3. Our approach assumes that the traced system has been instrumented using MReplayer’s instrumentation approach [17] and generates execution traces at runtime.

Our approach detects regressions by replaying execution traces collected from a base model on its modified version. To achieve this goal, first critical variables for each capsule are identified by the user. Critical variables determine what constitutes a regression: in the changed model, the values of critical variables must be the same as in the base model, while the values of non-critical values can differ. This allows the user to focus the analysis on the most relevant parts and abstract from changes to irrelevant parts. The selection of critical variables can thus be used as an ‘information hiding’ mechanism, e.g., the ‘hiding’ operator in classical process algebras such as CCS and CSP [133, 91]. We suppose a set of critical variables are provided by preferably a domain expert user as an input to the functions \textit{executionSelection} and \textit{regressionDetection}. The domain expert user is usually a developer or tester who has an in-depth knowledge about the system and can determine the critically of the participating components in the system. In the context of our running example, the variable balance amount, i.e., $bamt$ is considered as the only critical variable in the capsule $CTR$. The function \textit{executionSelection} selects only those executions of a base model that may modify any variables in a given set of critical variables. The output
of this function is fed into the function `regressionDetection`. The function replays the trace on both the base and the modified model to detect steps during which the modified model (1) makes a critical variable take on a value that is different from that in the base model, or (2) is unable to complete the replay of the trace.

As discussed in Section 2.4, MRegTest assumes that state machine models satisfy well-formedness conditions which typically are considered in MDD. These conditions ensure that 1) a message arriving at a capsule enables at most one rc-step, and 2) there is a 1-to-1 correspondence between a trace observed in the execution and the rc-step it corresponds to.
6.3.2 Detecting Regressions based on Modifications on rc-steps

A change is a set of primitive modifications on a base model (e.g., add/remove/change a transition) that developers apply to fix a bug or improve a functionality. In the following, some important aspects of modifications are discussed in more detail.

Leverage the Definition of rc-steps. As discussed in Chapter 2, the set of rc-steps of a capsule, extracted by performing static analysis, is fully capable of describing its possible behaviours. Therefore, we can leverage the definition of rc-steps to identify any regressions from the expected behaviour of a base model. In fact, any modifications of the model elements defining an rc-step (i.e., $\sigma$, $C$, $P$, and $\sigma'$) can affect the behaviour of its base model. In the context of the running example, the change in action code of the transition $t_9$ in the capsule $CTR$, reflects on its corresponding rc-steps (i.e., $rc_{CTR}^C$). When this rc-step is replayed, a critical variable may take on a different value, thus our approach can determine whether this modification may cause a regression or not.

What Types of Modifications Are Supported. The range of possible modifications that a developer may want to apply to a model can be extensive and depends on the size and complexity of the model. However, the majority of them can be put them into two groups:

- Modifications of the behavioural model including action code, messages and state machines

- Modifications of the structural model including ports, protocols, and connectors.

In this study, we merely consider the modifications of the behavioural model, and in our future work we will explore modifications of the structural model.
MReplayer creates a parser for *Action Code* language which is a subset of *C++* language using ANTLR [149]. This parser supports most commonly used expressions and primitive operations such as accessing and updating variables, arithmetic and logical expressions, and control flow statements (e.g., if, else if). Nonetheless, the parser does not support expressions in action code that use the following *C++* features:

- Object-oriented programming (OOP) principles (e.g., inheritance, encapsulation, and polymorphism).
- User-defined data types (e.g., template, class, structure, union, and enumeration).
- Derived data types (e.g., function, array, pointer, and reference).
- Periodic timer functions (e.g., inform in, inform every, and cancel).

Essentially, MReplayer does not generate values from the above-mentioned expressions by replay. Thus, any regressions due to a modification in action code of such expressions are not detected by our regression testing approach. For example, suppose a critical variable whose value is modified within a function in action code of a transition. Since replay of a function is not supported in our approach, possible regressions associated with such modification will remain undetected.

### 6.3.3 Critical Variable Identification (CVI)

Critical variables are those variables whose values during execution are important for the correctness of a system, thus any discrepancy between their values in base
model executions and that of the modified model may lead to a regression from the expected behaviour of a system. These variables are domain specific and identified by preferably domain expert users. Using a given set of critical variables allows the function \textit{executionSelection} to reduce the number of \textit{executions} that are being replayed in the function \textit{regressionDetection}. Moreover, it enables the user to adjust the granularity of the analysis (by adding or removing more critical variables) as needed.

**Definition 14.** (Critical Variables) We define a critical variable \(cv\) as a subset of all capsule variables whose values define the correctness of an USM execution (Definition 7).

### 6.3.4 Execution Selection (ES)

Our approach relies on the replay of executions collected from a base model. This process can be very time consuming, especially for distributed systems with several nodes where the number of traces collected from each execution can be large. Our approach aims to mitigate this issue by using the function \textit{executionSelection} shown in Algorithm 3. The output of this function can be used in multiple regression detection (i.e., Algorithm 4) experiments with different modified models. Consequently, with a fixed set of critical variables, as long as the base model does not change, the result produced by \textit{ES} does not need be updated. When a modification is approved as regression-free, the current base model can be replaced with the modified model and then \textit{ES} needs to be re-run for the new base model. As a high-level overview, Algorithm 3 reorders and replays the input execution to determine and return the executions along which at least one of the critical variables is assigned to at least
Algorithm 3: \text{executionSelection}(rcSteps: set of re-steps of C, critVar: set of critical variables, executions: set of executions of C)

1. Let selection be an emptyset
2. forall execution \in executions do
3. \hspace{1em} \gamma \leftarrow \gamma_0
4. \hspace{1em} inTraces \leftarrow execution
5. \hspace{1em} msgs \leftarrow append(msgs, startUp)
6. \hspace{1em} while (inTraces not consumed) do
7. \hspace{2em} msg \leftarrow dequeue(msgs)
8. \hspace{2em} rcStep, \gamma_1 \leftarrow getRCStep(msg, rcSteps, \gamma)
9. \hspace{2em} removeMatchingTrace(inTraces, rcStep)
10. \hspace{2em} \gamma \leftarrow replay(rcStep, \gamma_1)
11. \hspace{2em} actSeq \leftarrow actions(rcStep)
12. \hspace{2em} var \leftarrow variables(actSeq)
13. \hspace{2em} if \{critVar \cap var\} \neq \emptyset then
14. \hspace{3em} selection \leftarrow selection \cup \{execution\}
15. \hspace{2em} break
16. return selection

Once. In other words, the selection leverages the fact that executions along which no critical variable is ever assigned to cannot uncover a regression.

The function \text{executionSelection} uses a nested loop in which the outer loop iterates over all executions collected from a base model, and the inner loop replays re-steps to find executions that use critical variables in their computations. In line \#3, configuration is initialized with \gamma_0 where initial state is assigned to its active state, as well as default values are assigned to all the variables and attributes of its value map, i.e., \gamma.E. Then, (possibly out of order) traces of \text{execution} are assigned to inTraces and the message queue, i.e., \text{msg} is reset with the initial message startUp in line \#4-5. The \text{while} loop checks if all traces in inTraces not consumed (line \#6). In the line \#8, the function \text{getRCStep}(msg, rcSteps, \gamma) is used to find the matching re-step, i.e.,
6.3. DESCRIPTION OF APPROACH

rcStep with respect to msg, list of rc-steps, i.e., rcSteps and the current configuration, i.e., \( \gamma \). Well-formedness condition \( C1 \) implies that an incoming message msg can enable at most one rc-step. Then, the function \( \text{replay} \) is used to replay rcStep and create a new configuration, i.e., \( \gamma \) (line \#10). According to the condition \( C2 \) the execution of an rc-step cannot get stuck. The function \( \text{variables} \) is used to select only the variables being assigned to in any of the action code blocks (i.e., \( \text{actSeq} \)). If any of the variables in \( \text{var} \) are critical, the execution is added into the set of selected executions (i.e., \( \text{selection} \)) that this function outputs.

For example, given the sample set of executions in Figure 6.2 as well as the set of critical variables \( \text{critVar} \) (i.e., \{bamt\}) and rc-steps of \( \text{CTR} \) in Table 2.1. The function \( \text{executionSelection} \) returns \( \text{selection} \) including the executions \( E_2 \) and \( E_3 \). In fact, it realizes that the critical variable bamt is modified in the rc-steps \( rc_4 \) and \( rc_5 \) corresponding to the traces \( t_8^C \) and \( t_9^C \) in \( E_2 \) and \( E_3 \), respectively. So, \( E_2 \) and \( E_3 \) are added into the set \( \text{selection} \). Moreover, if users want to perform regression testing at a finer granularity, they can include the variable \( \text{reject} \) in \( t_5^C \) into the set of existing critical variables. As a result, the function \( \text{executionSelection} \) will output more executions (i.e., \( E_1 \), \( E_2 \), and \( E_3 \)) which leads to replay of more executions in the function \( \text{regressionDetection} \).

In the following, we offer formal definitions for Test Case and Test Case Execution.

**Definition 15.** *(Test Case)* Let \( tc \) be a set of initial configurations i.e., \( \gamma_0 \), that are used to initialize an execution of their respective USM. All test cases are collected in the test suite \( TS \), i.e., \( TS = \{tc_1, tc_2, ..., tc_j, ..., tc_m\} \) where \( tc_j \) is the \( j \)-th test case in the test suite \( TS \).

For example, if a model consists of three capsule instances, \( c_1 \), \( c_2 \), and \( c_3 \), then
\[ tc_1 = \{ \gamma_{c_1}^{0,1}, \gamma_{c_2}^{0,1}, \gamma_{c_3}^{0,1} \} \]
where \( \gamma_{c_1}^{0,1}, \gamma_{c_2}^{0,1}, \) and \( \gamma_{c_3}^{0,1} \) are used to assign default values to the variables and attributes of capsules \( c_1, c_2, \) and \( c_3, \) respectively.

**Definition 16.** *(Test Case Execution)* Test case execution is an application of a test case to the system under test (SUT). If \( tc_i \) is a test case in the test suite \( TS \) (Definition 15) and \( M \) is the model of the SUT (Definition 4), then we will write \( TCE(M, tc_i) \) to denote that the model \( M \) is executed using the initial configuration specified by the test case \( tc_i. \) \( TCE(M, tc_i) \) produces a sequence of execution traces \( e_i \) (Definition 10) that in Algorithm 3 is called ‘execution’.

All executions obtained from a test suite \( TS \) can be grouped into: (1) regression-revealing executions, i.e., those executions collected from the base model \( M \) that will cause a regression on the modified model \( M' \); (2) safe executions, i.e., those executions collected from the base model \( M \) that will not cause regression on the modified model \( M' \).

### 6.3.5 Regression Detection (RD)

The function \( \text{regressionDetection} \) shown in Algorithm 4 is used to detect regressions from an expected behaviour of a modified model. Furthermore, this algorithm is executed for every abstract capsule instance of a modified model regardless of whether their state machines have been modified or not. As an overview, similar to Algorithm 3, it initializes current configurations and the message queue \( msgs. \) Then, it leverages MReplayer’s reorder and replay mechanism to determine: (Step \#1) whether MReplayer is able to replay traces collected from a base model on its modified version, (Step \#2) whether values of critical variables are consistent between the base and modified version of a model. Finally, this function returns \( \text{regression} \) that
can include a set of critical variables causing regression, the strings ‘NoRCStep’ (in case the incoming message does not enable any rc-step) and ‘NoMatchingTrace’ (in case no trace matching the enabled rc-step can be found in the sequence of incoming traces).

**Step #1:** The algorithm reorders (possibly out of order) traces and replays them on the modified capsule $C'$ (line #5-15). Similar to the Algorithm 3 (line #8), the function $getRCStep$ outputs an rc-step which is unique according to the well-formedness condition $C_1$. In the line #10, the function $removeMatchingTrace$ looks for the matching trace in $inTraces$. This function blocks until a trace matching the rc-step $rcStep$ appears in $inTraces$ or until a timer defined by the abstract controller expires. The timer is set to a user-defined value and starts counting down if replay is deadlocked, i.e., if the execution of function $regressionDetection$ in all abstract capsules is stuck at line #10. Note that the timeout value can be small, because the replay is centralized and does not involve network communication (also note that in Algorithm 3, function $removeMatchingTrace$ will never block and wait, because that algorithm simply replays base model executions on the same base model and so the deadlock situation described above is impossible). Finally, the function $replay$ is used to replay $rcStep'$ and create a new configuration, i.e., $\gamma'$ (line #13).

**Step #2:** The algorithm uses the same replayer mechanism to re-generate variable values based on the original capsule $C$. To this end, it identifies a unique rc-step from the set of all possible rc-steps, i.e., $rcSteps$ in the original capsule $C$ using the same function $getRCStep$. Then, the function $replay$ is used to replay the rc-step determined in the line #16 and create a new configuration $\gamma$ for the original capsule $C$ (line #19). In the line #21, the algorithm uses the boolean function $different$ that

1. Let `regressions` be an empty string
2. \( \gamma, \gamma' \leftarrow \gamma_0 \)
3. `msgs` \( \leftarrow \) append(`msgs`, `startUp`)
4. while (`inTraces` not consumed) do
   5. // Step#1: Reorder & replay on the modified capsule C'
      6. `msg` \( \leftarrow \) dequeue(`msgs`)
      7. `rcStep', \gamma'_1 \leftarrow \text{getRCStep}(msg, rcSteps', \gamma')
      8. if (`rcStep' = \emptyset`) then
         9. `regressions` \( \leftarrow \) \text{'NoRCStep'}
        10. break
      11. if (`\text{removeMatchingTrace}(inTraces, rcStep')`) then
         12. `regressions` \( \leftarrow \) \text{'NoMatchingTrace'}
        13. break
      14. `\gamma' \leftarrow \text{replay}(rcStep', \gamma'_1)`
      15. `vMap' \leftarrow \text{values}(\gamma'.E, critVar)
   6. // Step#2: Replay on the original capsule C
   7. `rcStep, \gamma_1 \leftarrow \text{getRCStep}(msg, rcSteps, \gamma)`
   8. if (`rcStep = \emptyset`) then
      9. `regressions` \( \leftarrow \) \text{'NoRCStep'}
     10. break
   11. `\gamma \leftarrow \text{replay}(rcStep, \gamma_1)`
   12. `vMap \leftarrow \text{values}(\gamma.E, critVar)`
   13. if (`\text{different}(vMap, vMap')`) then
      14. `regressions` \( \leftarrow \) \text{\text{difference}(vMap, vMap')}
     15. break
16. return `regressions`

The algorithm takes the value maps, i.e., `vMap` and `vMap'`, and checks whether value of critical variables are consistent. If yes, this function returns false and the algorithm goes back to the line #4. If no, the function `\text{difference}` assigns changed critical variables to the `regressions`, breaks the loop and returns the `regression` (line #24).
6.3. DESCRIPTION OF APPROACH

6.3.6 Example

Figure 6.4 shows an execution of our approach on the input trace $E_2$ in Figure 6.2. Initially, all abstract capsules are waiting for a trace that matches their respective rc-steps, i.e., $rcStep'$. All state machines are in their initial states ($in_1$). In Step 0, the message initial $startUp$ is in the queues of all capsules. In Step 1, input trace $t_1^P$ is processed, and the rc-step $rc_1^P'$ is replayed by the (abstract capsule of) $PPD$ which leads to state $s_1$. Similarly for Step 2 and 3. In Step 4, the algorithm checks whether is there any difference between values in the value maps (i.e., $vMap'$ and $vMap$). In Step 5, input $t_2^P$ can be matched to $rc_2^P'$ which is also found enabled and thus can be replayed, causing $PPD$ to move to state $s_3$, message $usrReq$ is sent to $CTR$. Similarly in Step 6 the capsule $CTR$ replays the rc-step $rc_2^C'$ and the message $con$ is sent to $BNK$. Step 7 is similar to Step 4 where the function $different$ results in no regression. In Step 8, the trace $t_2^B$ matches the rc-step $rc_2^B'$ in the capsule $BNK$ which is found enabled and thus is replayed and the message $ack$ is sent to $CTR$. Similarly for Step 9-14. In Step 15, $CTR$ replays $rc_6^C'$ and sends the message $done$ to $PPD$. Also, the value of the variable $val$ is assigned to the critical variable $bamt$. Then, in Step 16 the function $different$ identifies the inconsistency between the value of $bamt$ in the modified model and the base model. So, $bamt$ is assigned to $regressions$ and the algorithm terminates.

In Figure 6.2 columns $inT$, $rcStep$, and $rcStep'$ show the input trace, the matched rc-step of the original, and the modified capsule, respectively. The column $status$ indicates what the capsule is doing where $wait_{inT}$ = ‘waiting for a match in the input traces’ (line #11), $replay(rc)$ = ‘replaying rc-step $rc$’ (line #14), and $different$ = ‘checking differences between value maps’ (line #23). Columns $s$ and $msgs$ indicate
### Figure 6.4: Sample Execution of our regression testing approach on ATM system on the input trace $E_2$ in Figure 6.2
the new state and the sequence of incoming messages.

6.4 Tool Support

This section describes our implementation of MRegTest\textsuperscript{3} In our implementation we
use Papyrus-RT as the primary tool for modelling UML-RT models and Epsilon \textsuperscript{109} for designing the transformation rules required for instrumenting the models.

Our tool consists of two parts: (1) Automatic Mutant Generation\textsuperscript{4}(AMG) that
facilitates the evaluation of regression testing by generating mutants from a UML-
RT model according to a user-defined set of critical variables; (2) Regression Testing
(RT) that allows the user to perform a replay-based regression testing on single and
multiple modified models.

6.4.1 Automatic Mutant Generation (AMG)

MRegTest benefits from a built-in mutant generation engine that facilitates the eval-
uation of the regression testing by generating several mutants from a UML-RT model
according to a user-defined set of critical variables. In fact, AMG is used to examine
the richness of the execution traces collected from a base model. Since MRegTest re-
lies on the replay of traces on a modified model, its effectiveness to detect regressions
depends on how well all execution paths are covered by traces collected from the base
model.

Each mutant is defined as a new version of the input UML-RT model whose action
code in only one of its elements were changed. Basically, AMG leverages MReplayer’s

\textsuperscript{3}A video that demonstrates the tool: \url{https://youtu.be/1PXjmKgadQI}

\textsuperscript{4}AMG is not necessary for the testing, and it is only used for the evaluation of our proposed solution.
action code parser to generate mutants from only numerical critical variables using some rules that are standard ways of producing mutants from an action code [62]: (1) if a value from an expression is assigned directly to a critical variable, e.g., \( a := \text{exp} \), then AMG generates one mutant with \( a := \text{exp}+1 \) and one mutant with \( a := \text{exp}-1 \); (2) if a critical variable increments by a value from an expression, \( a += \text{exp} \), then AMG generates one mutant with \( a += (\text{exp}+1) \) and one mutant with \( a += (\text{exp}-1) \). In the context of our running example, if we consider \( \text{bamt} \) as a critical variable, then AMG will generate two variations of the ATM Model: One of which uses \( \text{bamt} := \text{val}+1 \) and the other uses \( \text{bamt} := \text{val}-1 \) as the action code of \( t_{12} \) in \( CTR \).

### 6.4.2 Regression Testing (RT)

Regression testing takes a modified model or a group of mutants as input and produces a report as output that explains whether the given modified model will cause a regression or not. Technically, it implements Execution Selection (ES), i.e., Algorithm 3 and Regression Detection (RD), i.e., Algorithm 4.

### 6.4.3 MRegTest Features

MRegTest uses separate control panels for AMG shown in Figure 6.5 and for RT shown in Figure 6.6. MRegTest is comprised of a set of engines (i.e., Generate Mutants, Test Execution, Webserver, and Replay) that can be executed and monitored via these control panels. In the following, we discuss the main features of MRegTest from the end-user point of view.

**Generate Mutants.** MRegTest allows the user to generate multiple mutants from
a base model using AMG. For this purpose, the user should choose a model (from a list of models) via the first ‘Browse’ button (i.e., 1). The user then needs to specify a directory to store mutants (i.e., 2). Also, the user must provide a list of critical variables in the input box, i.e., 3. The server uses this list and only
generates mutants from the critical variables mentioned in this list. Finally, once the user clicks the button ‘Generate Mutants’, the server starts and generates mutants in the directory. The user can check its progress via the progress bar, i.e., ④ and the output box, i.e., ⑤.

**Regression Testing.** First of all, the user should choose a testing mode between Multiple Models mode or Single Model mode using the check boxes, i.e., ① and ②, respectively. If Multiple Models was chosen, then the user would have to specify the path to the directory containing modified models using the ‘Browse’ button (i.e., ④). However, if Single Model was chosen, then the path to one modified model would be required (using the ‘Browse’ button, i.e., ③). In both cases, MRegTest requires a file containing traces collected from the base model that is selected using the ‘browse’ button, i.e., ⑤.

Similar to the other server, the user has to add some critical variables in the input box, i.e., ⑥. Finally, it can run the regression testing server using the button ‘Run’, i.e., ⑦. In Multiple Model mode, MRegTest performs the replay-based regression testing on all modified models in the specified directory one at a time and shows the output (i.e., either ‘Regression’ or ‘No Regression’) on the output
box, i.e., 10 for each modified model. In this mode the features Webserver and Replay are not available. However, in Single Model mode, the user can get an in-depth understanding of (possible) regressions by running the Webserver and replay the traces via the buttons shown in 8 and 9, respectively. MRegTest’s webserver, that is shown in Figure 6.7 adopts MReplayer’ webUI to show an integrated view of the modified distributed system and replay traces collected from the base model on the modified model. Basically, it consists of three parts: (1) workstation, i.e., 1, that shows a view of the system; (2) Replayer Panel, i.e., 2, that provides control over the replay of an execution using four buttons (i.e., ‘Back’, ‘Run’, ‘Stop’, and ‘Next’); (3) Inspection Panel, i.e., 3, that allows the user to inspect the value of variables in the system at any given time.

6.5 Experimental Evaluation

In this section, we use the evaluation framework for regression test selection techniques proposed by Rothermel et al. [155]. The framework was originally designed for code-based techniques but Briand et al. [40] showed that most of the principles of Rothermel’s framework can be applied to regression testing techniques at a model level. The results are shown in Table 6.1 and Table 6.2. In these tables, #C, #S, #T, Inst., ms, P, R indicate number of capsules, number of states, number of transitions, instrumentation, milliseconds, precision, recall, respectively.

6.5.1 Prototype Implementation

MRegTest has been developed using Java on a Linux (Ubuntu) environment running on a desktop machine equipped with a 2.7GHz Intel Core i5 and 8GB of memory.
Also, we implemented our approach in Eclipse Papyrus-RT for distributed systems \[106\], which is an industrial-grade and open-source MDD tool. We used the Epsilon Object Language (EOL) \[109\] to implement the transformation rules required for instrumentation of the models.

### 6.5.2 Evaluation Approach

In the following, we describe evaluation metrics, the experiment we conducted in order to assess the applicability of our approach and results.

**Processing Time of Our Approach (EXP-1).** This experiment aims to evaluate how much the processing time of Algorithm 3 and Algorithm 4 increases as the size of case studies listed in Table 6.2 grows. To set up this experiment: (1) We consider only one critical variable for each capsule; (2) For each case study, we collected traces from 1,000 executions. Each execution is generated from a test case that was designed manually, and it sets initial values; (3) We generated 100 mutants from each case study. First, we evaluated Algorithm 3 using (1) and (2), then its output is used as input for evaluation of Algorithm 4 against 100 mutants in (3). We performed each step (i.e., ES and RD) 20 times for each case study and averaged the times required for each step and each case study combination.

**Collection of Execution Traces (EXP-2).** The size of traces collected from base model executions can grow rapidly. This experiment measures the size of traces collected from each case study. We also extended our prototype and implemented the traditional trace replayer that annotates traces with timestamps and variable values. As discussed in Section 6.3 our approach benefits from MReplayer’s instrumentation technique that neither adds timestamps nor variable values to each trace. As the first
part of this experiment, we ran each approach using identical deployment configurations and collected 500,000 traces for each case study. We repeated this experiment 10 times for each case study and averaged sizes. In the second part, we collected the number of traces ranging from 1,000 to 500,000 using our approach and the traditional approach.

Reduction in the Number of Executions (EXP-3). In this experiment we evaluate the efficiency of the first step of our approach (i.e., Algorithm 3) to select regression-revealing executions based on a given set of critical variables. The same setup as EXP-1 is used in this experiment. We measured how many executions out of total 1,000 executions are selected by the function executionSelection, and then similar to [176] we calculated reduction for each case study.

Evaluating the Effectiveness of ES (EXP-4). This experiment measures the extent to which our approach chooses regression-revealing executions that were manually generated. We adopted the same setup as in EXP-1. Then we used Algorithm 3 to select executions based on a given set of critical variables for each case study. We evaluated the effectiveness of ES using the two widely used metrics, i.e., precision and recall. The former basically measures how accurate Algorithm 3 selects those executions that cause regression. More specifically, precision measures the True Positives divided by True Positives + False Positives. The latter aims to highlight the amount of regression-revealing executions that were incorrectly not selected, i.e., True Positives divided by True Positives + False Negatives [65].

Measuring Runtime Overhead (EXP-5). Since Algorithm 3 and Algorithm 4 rely on re-generating variable values by replay, they may entail additional runtime overhead for the replay of traces. In this experiment we compared the efficiency of our
approach with the traditional approach that we used in EXP-2 to determine whether the runtime overhead of both algorithms are reasonable as the complexity of the case studies and the size of the traces grow. To this end, we used the same setup as in EXP-2. However, in this experiment rather than size of traces, we recorded execution time for the replay of traces using both algorithms. Similar to EXP-2, as the first part of this experiment we replayed 500,000 traces collected from each case study. And as the second part, we replayed different numbers of traces (from 1,000 to 500,000).

**Evaluating the Effectiveness of RD (EXP-6).** Using this experiment we evaluated the effectiveness of Algorithm 4 in terms of the number of faulty mutants it detects as regression. We generated 100 mutants from each case study including faulty and non-faulty mutants. We also ensured that for every faulty mutant, at least one execution is collected from the base model. In this experiment, the output of Algorithm 3 in EXP-3 is used to provide a set of executions for Algorithm 4. For every mutant of the case studies in Table 6.2 we ran our regression testing approach using identical deployment configurations and measured the number of faulty mutants that had been detected as regression. Finally we calculated precision and recall for each case study.

**Experimental Environment.** We used the development environment explained above for experiments EXP-1 to EXP-6. Also, we used Java version 1.8.0161 in configuration -Xmx12512m. The source code of the experiments and models are publicly available at [20].
6.5. EXPERIMENTAL EVALUATION

Table 6.1: Complexity of Use-cases, Size Reduction, and Performance of ES

<table>
<thead>
<tr>
<th>Model</th>
<th>Complexity</th>
<th>Inst.</th>
<th>Algorithm 3 (ES)</th>
<th>Efficiency</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#C</td>
<td>#S</td>
<td>#T</td>
<td>avgSRed.</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>SCMS</td>
<td>3</td>
<td>6</td>
<td>25</td>
<td>1.21×</td>
<td>103</td>
</tr>
<tr>
<td>CDCL</td>
<td>5</td>
<td>11</td>
<td>15</td>
<td>1.32×</td>
<td>234</td>
</tr>
<tr>
<td>SPR</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>1.27×</td>
<td>494</td>
</tr>
<tr>
<td>PR</td>
<td>8</td>
<td>14</td>
<td>25</td>
<td>1.45×</td>
<td>661</td>
</tr>
<tr>
<td>RO</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>1.56×</td>
<td>703</td>
</tr>
<tr>
<td>FO</td>
<td>7</td>
<td>31</td>
<td>43</td>
<td>1.39×</td>
<td>783</td>
</tr>
<tr>
<td>ATM</td>
<td>12</td>
<td>295</td>
<td>349</td>
<td>1.98×</td>
<td>2,259</td>
</tr>
<tr>
<td>RFO</td>
<td>13</td>
<td>2304</td>
<td>3647</td>
<td>2.13×</td>
<td>6,609</td>
</tr>
</tbody>
</table>

Table 6.2: Complexity of Use-cases, and Performance of RD

<table>
<thead>
<tr>
<th>Model</th>
<th>Complexity</th>
<th>Algorithm 4 (RD)</th>
<th>Efficiency</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#C</td>
<td>#S</td>
<td>#T</td>
<td>avgSRed.</td>
</tr>
<tr>
<td>SCMS</td>
<td>3</td>
<td>6</td>
<td>25</td>
<td>651</td>
</tr>
<tr>
<td>CDCL</td>
<td>5</td>
<td>11</td>
<td>15</td>
<td>475</td>
</tr>
<tr>
<td>SPR</td>
<td>8</td>
<td>12</td>
<td>14</td>
<td>955</td>
</tr>
<tr>
<td>PR</td>
<td>8</td>
<td>14</td>
<td>25</td>
<td>1,088</td>
</tr>
<tr>
<td>RO</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>2,139</td>
</tr>
<tr>
<td>FO</td>
<td>7</td>
<td>31</td>
<td>43</td>
<td>2,282</td>
</tr>
<tr>
<td>ATM</td>
<td>12</td>
<td>295</td>
<td>349</td>
<td>17,582</td>
</tr>
<tr>
<td>RFO</td>
<td>13</td>
<td>2304</td>
<td>3647</td>
<td>34,041</td>
</tr>
</tbody>
</table>

6.5.3 Results and Discussions

In this chapter, we propose the use of MRegTest to reduce the cost of replay-based regression testing for distributed systems. Since we found no previous works with the same goal (i.e., replay-based regression testing for distributed systems in the context of MDD), we compare MRegTest with an implementation of our work that annotates traces with timestamps and variable values as baseline. Also, we do not compare our approach with orthogonal techniques that generate test cases (e.g., [194]) because test case generation is outside the scope of this work. We set a time-budget of 8
hours when analyzing each test case, which is in line with industrial practice (e.g., nightly-build-and-test) \cite{176}.

**Time-Efficiency.** It is important that processing time of the algorithms in our approach (in Section 6.3) remains within an acceptable range even when model size grows exponentially. Based on EXP-1, the columns labelled *Time* in Table 6.1 and Table 6.2 show the time required to select relevant executions and detect regressions in Algorithm 3 (i.e., *ES*) and Algorithm 4 (i.e., *RD*), respectively. In the worst case (i.e., the largest model), *ES* takes 6,609 ms and *RD* takes 34,041 ms. Going from *FO* to *RFO*, the number of capsules, states, and transitions increase by factors 1.8, 74, and 84, respectively; however, *ES* and *RD* times only increase by factors 8.4 and 14.9, respectively. While processing time increases with model size, the results show that the processing times are reasonable and do not grow exponentially.

**Size-Efficiency.** As opposed to traditional approaches that annotate traces with timestamps and variable values, our approach re-generates variable values by replay which may lead to a significant reduction in the size of generated traces. In Table 6.1 column *avgSRed* shows results of the first part of EXP-2. This indicates the amount of reduction in sizes of traces collected from our approach compared with sizes of traces collected from the traditional approach. The reduction ranges from $1.27\times$ to $2.13\times$, with a geometric mean of $1.56\times$. Results of the second part of EXP-2 for two case studies (i.e., *ATM* and *FO*) are illustrated in the line-chart at the bottom of Figure 6.8. It shows in both case studies our approach, that uses MReplayer’s instrumentation technique, generates smaller sizes of traces. For example, the size of 400,000 traces of the *ATM* in our approach (i.e., *ATM-Ours*) is 72.9MB. Whereas the size of traces generated from the traditional approach (i.e., *ATM-Trad.*) is 1.66
times higher, i.e., 121.3MB. The difference is getting even larger for 500,000 traces (from the same model) where the size of traces generated in our approach is 1.98 times smaller than that of the traditional approach.

**Execution Reduction.** The column `exeRed.` (under the column `ES`) shows the results of this experiment. The reduction ranges from $1.52\times$ to $10.44\times$, with a geometric mean of $4.67\times$. `MRegTest` achieves high reduction because the proportion of executions that affect a given set of critical variables is generally very small. Also, the effect of this reduction has a large impact on processing time of the function `regressionDetection` that is shown in the column `Time` under the column `RD`. Because as discussed in Section 6.3, an efficient `ES` technique could significantly reduce the number of executions that `MRegTest` needs to replay for detecting possible regressions in a modified model.

**Runtime Overhead.** It studies the runtime overhead of Algorithm 3 and Algorithm 4 for replay of traces in case studies with various complexity. In Table 6.1 columns `avgROver.` (under `ES` and `RD`) show average runtime overhead for both algorithms that are results of the first part of EXP-5. They show execution time increases in all case studies mainly due to the use of the replayer for re-generating variable values. Nonetheless, the imposed runtime overhead for all case studies in `ES` ranges from $1.01\times$ to $1.12\times$ and in `RD` ranges from $1.01\times$ to $1.17\times$. Also, for all case studies runtime overhead of `RD` is higher than that of `ES`. The line-chart at the top of Figure 6.8 only illustrates execution time of `RD` for two case studies (i.e., `ATM` and `FO`). It shows that in both case studies the difference between execution time of `RD` in our approach and that of the traditional approach is in the order of few seconds. This implies that benefit of replaying outweighs its costs (i.e., slowing
6.5. EXPERIMENTAL EVALUATION

down the analysis).

Effectiveness of ES. It is important for a reply-based regression testing technique to be able to select all executions that cause regressions in a modified model. The columns Precision and Recall under the column ES show the performance of Algorithm 3 in selecting regression-revealing executions for case studies listed in Table 6.1. These columns show precision and recall range from 0.95 to 1.00 and from 0.97 to 1.00, respectively. The results show MRegTest did not miss any executions in most of the case studies. However, in some case studies there are few missed regression-revealing executions due to the lack of support for replay of some statements in the action code (e.g., function calls, pointer assignments). This indicates that Algorithm 3 is sufficiently precise to select regression-revealing executions among total 1,000 executions for each case study.

Effectiveness of RD. Similar to ES, we evaluate the effectiveness of Algorithm 4 with respect to precision and recall and the results are shown in the columns with the same name under the column RD in Table 6.2. For all case studies, precision and recall range from 0.94 to 1.00 and from 0.97 to 1.00, respectively. Since in this evaluation we used the output of the EXP-3, missing some regression-revealing executions in Algorithm 3 will reduce the capability of Algorithm 4 for detecting all regression. Nevertheless, the results indicate that precision and recall in most case studies are high enough to conclude that Algorithm 4 is accurate to detect large amount of mutants that cause regressions in case studies with different complexity.
6.6 Threats to validity and limitations

The primary threat to external validity is the representativeness of our case studies. In fact, we have considered only eight models with various complexity, and the results that we have obtained during our experimental studies are limited to programs (i.e., code generated from models) of a maximum size of approximately 7,266 LOC.

In terms of the internal validity, even though we have carefully developed the experimental setup, there could be defects in the implementation of our tool, as well as the traditional approach we re-implemented to perform the experimental evaluation. We reduced this threat by extensive testing, manual inspection, and verifying their
results against models (e.g., SCMS and CDCL) for which we can manually determine the correct results.

The primary threat to construct validity are the measurements of efficiency. We have considered time and size to evaluate the efficiency of our approach. However, other metrics such as test setup and maintenance costs can play a role in technique efficiency.

In addition, the limitations of our approach can be divided into five main groups: (1) the simplifying assumptions that we have made; (2) the concept of abstract capsules and abstract controller that we have used in MReplayer; (3) heavily reliant on the richness of the test suite; (4) the gap between re-execution and replay; (5) the lack of formal proof for our approach. In the following we explain how each of these limitation might affect the effectiveness of our approach.

**Simplifying assumptions.** Our approach assumes that a set of critical variables is defined by a domain expert user who is able to identify correctly those variables whose values determine the correctness of the system. Often this assumption is not realistic, especially in complex distributed systems where each part of the system is developed in a different team. Therefore, identifying all the critical variables is not a trivial task. Consequently, identifying a normal variable as a critical (or missing a critical variable) may reduce the effectiveness of our approach in detecting regressions. Also, identifying all the variables as critical variables may undermine the performance of our approach in terms of efficiency. Because, in this case no executions will be filtered by Algorithm 3. Thus, Algorithm 4 needs to replay all executions in order to detect regressions.
6.6. THREATS TO VALIDITY AND LIMITATIONS

**Abstraction in MReplayer.** As explained in Chapter 5, MReplayer uses an abstract version of the distributed system including abstract capsules and an abstract controller to reorder (possibly) out-of-order traces. Since abstract capsules do not have ports, any regression caused due to the modification of the structural model such as changing a protocol or adding or removing an existing connector cannot be detected using our approach. In addition, our parser for Action Code language does not support C++ expressions that use object-oriented programming principles, user-defined data types, derived data types, and periodic timer functions. Thus, MRegTest is not able to detect regressions due to any modifications in these expressions.

**Richness of the test suite** Since our approach relies on the replay of traces collected from initial executions of the base model to detect possible regressions, its result (i.e., either regression or regression-free) is limited to the execution paths that the initial executions of the system have gone through. In another word, a modified model that has been evaluated as regression-free by MRegTest may exhibit a regression from its expected behaviour if it goes through an execution path that has not been tested by MRegTest. Essentially due to the non-deterministic nature of distributed systems, even a small change in the non-functional characteristics of the system such as network delay or routing mechanism may cause the system to go through a new execution path that may lead to a regression. Thus, the richness of the test suite that we use for generating initial executions of the base model plays a vital role in the accuracy of our approach.

**The gap between re-execution and replay** Our analysis is based on a replay of the changed model which is not necessarily equivalent to the re-execution of the code generated distributed system from the changed model. Because some underlying
features of an actual system such as those controlled by the operating system, e.g., scheduling the processes on the CPU, are not emulated by MRegTest. Therefore, even a slight change in these features may lead to a regression that is not detected by our approach.

**Lack of formal proof.** To verify the correctness of our implementations, we rely on testing our approach on various models and comparing the outputs with the expected results. We did not provide formal proof for the regression detection and the execution selection techniques in our approach. It can be useful to formally prove those aspects of our implementations.

### 6.7 Summary

In this chapter, we have proposed a conceptual framework for replay-based regression testing of distributed systems in the context of model-driven development. Our approach is three-fold: First it uses Critical Variable Identification that allows the user to adjust the level of granularity of the regression tests as appropriate; second, it applies Execution Selection that works as an optimization step and reduces the number of base model executions; finally, it detects regression using our model-based trace replayer from [17]. We also have created a prototype called *MRegTest* based on the proposed framework for testing distributed UML-RT models. *MRegTest* reduces the cost of regression testing by replaying only those executions that modify critical variables specified by the user. Moreover, *MRegTest* re-generates variable values by replay of the action code as opposed to the traditional approaches that read the information from traces directly. Our experimental results showed that *MRegTest* detects almost all regressions in case studies with various complexity. Also, it reduces the
size of collected traces significantly while imposing negligible runtime overhead.
Chapter 7

Conclusions and Future Work

Providing proper support for regression testing of distributed systems at the model-level is challenging. Due to the special characteristics of distributed systems such as nondeterminism, execution traces might arrive at a centralized tester in arbitrary time intervals. As a result, running an experiment repeatedly may lead to different results. Thus, regression testing by replay requires an efficient mechanism for generating compact traces, as well as reordering, and replaying possibly out of order traces. In contrast to existing approaches that employ either physical or logical timestamps for reordering execution traces, our approach relies on performing static analysis on the behavioural models and extract control flow and communication dependencies between run-to-completion steps. Also, our regression testing approach re-generates variable values by replay rather than re-execution of the system repeatedly. This enables the user to have sufficient control over the replay of the changed system in terms of stopping and running execution and inspecting variable variable values at any time. The experimental results show that compared to the traditional approaches that annotate traces with timestamps and variable values our approach detects almost all regressions while reducing the size of the trace significantly and incurring
similar runtime overhead.

7.1 Future Work

Some directions for future work include but not limited to the following:

7.1.1 Extend the support for more advanced features

The current version of our replay-based regression testing approach, we consider modifications of the behavioural model. More specifically, our parser of UML-RT action code only supports most commonly used expressions and primitive operations such as accessing and updating variables, arithmetic and logical expressions, and control flow statements (e.g., if, else if). As the first direction to improve our proposed replay-based regression testing approach, we can consider extending the support for more advanced features of action code including more complex data types and user-defined method calls. These features can be easily added to the current implementation by extending our action code parser and adding new rules for handling expression that include advanced C++ features, e.g., user-defined data types and derived data types.

7.1.2 Extend the support for automatic critical variables identification

Another important direction is to extend the support for automatic critical variables identification. These variables are domain specific and in the current version of our work they are identified by preferably domain expert users. In fact, it places the burden on the user to find, ideally, a set of critical variables that is: (1) neither too large (i.e., the regression analysis operates on a level of granularity (detail) that is too fine such that even irrelevant differences will be flagged as regressions, or executions that
cannot uncover what the user will accept as regressions will be replayed unnecessarily); (2) nor too small (i.e., the analysis is not detailed enough and regressions might go undetected.). This issue can be mitigated by assigning different levels of priority to the participating capsules. Then we can consider all variables in the capsule with the highest priority as critical variables and run Execution Selection and Regression Detection functions. If any regressions are detected, we stop regression testing and try to figure out why a variable in this capsule is getting unexpected value. Otherwise, we move on to the second-highest priority level and repeat the process for this capsule. Indeed, this solution requires some knowledge about the importance of each capsule in the correctness of the system. However, this approach offers a more systematic way to detect regressions without the need to provide a set of critical variables in advance. This solution also enables developers to perform coarse- and fine-grained regression testing by adjusting the number of priority levels. For example, coarse-grained regression testing based on this approach will consider only a small number of priority levels and if no regression is found, developers may conclude that the entire system is regression-free.

7.1.3 Extend the evaluation of the proposed work

We can also extend the evaluation of the proposed work to consider more complex models that are particularly designed for actor oriented programs and also more representative of both classic and contemporary distributed systems issues. To this end, we can use Savina, a standardized benchmark suite that represents various use-cases in actor-oriented programs [98]. It provides a set of diverse and realistic benchmarks including (1) micro benchmarks, e.g., Ping Pong, Thread Ring, and Counting Actor;
(2) concurrency benchmarks, e.g., *Concurrent Dictionary*, *Producer-Consumer*, and *Dining Philosophers*; (3) parallelism benchmarks, e.g., *All-Pairs Shortest Path*, *A-Star Search*, and *Online Facility Location*. In the current version of our evaluation method, we only considered a few of these benchmarks mostly in micro benchmarks such as *Ping Pong* and concurrency benchmarks such as *Dining Philosophers*. New benchmarks just need to be implemented in UML-RT modelling language and then they can be easily added to our proposed work.
Bibliography


xDSMLs. In 8th International Conference on Software Language Engineering (SLE), Pittsburgh, United States, October 2015.


[43] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and
Jorgen Thelin. Orleans: Cloud computing for everyone. SOCC, New York, NY,
USA, 2011. Association for Computing Machinery.


[45] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni. Retrospec-
tive lightweight distributed snapshots using loosely synchronized clocks. In
2017 IEEE 37th International Conference on Distributed Computing Systems
(ICDCS), pages 2061–2066, June 2017.

[46] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed

test suite generation using dependence analysis. In Proceedings of the 3rd In-
ternational Workshop on Advances in Model-based Testing, A-MOST ’07, pages
54–62, New York, NY, USA, 2007. ACM.

test suite generation using dependence analysis. A-MOST ’07, page 54–62,

[49] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduc-
tion using extended dependence analysis. SOQUA’07, page 62–69, New York,
NY, USA, 2007. Association for Computing Machinery.


[56] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili,


[123] Bruno Lima and João Pascoal Faria. A survey on testing distributed and heterogeneous systems: The state of the practice. In Enrique Cabello, Jorge Cardoso, André Ludwig, Leszek A. Maciaszek, and Marten van Sinderen, editors,


