Ordering and Replaying of Execution Traces of Distributed Systems in the Context of Model-driven Development

Majid Babaei, Mojtaba Bagherzadeh and Juergen Dingel
Queen's University
Kingston, Canada
{babaei,mojtaba,dingel} @cs.queensu.ca

Abstract—Ordering and replaying of execution traces of distributed systems is a challenging problem. State-of-the-art approaches annotate the traces with logical/physical timestamps and then order them in accordance with their timestamps. Physical timestamps are often inaccurate due to issues of synchronization and clock precision, and logical timestamps cause an increase in the size of traces as the number of nodes in the distributed system increases. This paper concerns trace ordering and replay in the context of model-driven development of distributed systems. By leveraging the abstract nature of models and existing model analysis and transformation techniques, we introduce a fully automated, efficient, and accurate method of trace collection and replay, without the use of timestamps. Instead, we use an abstract interpretation of models which orders and replays traces using the causality relation between traces. We have created a prototype based on our approach, and we summarize the results of applying the prototype to several use cases to determine benefits and costs in terms of trace size and performance overhead. We show that our approach decreases the size of traces significantly and imposes a similar performance overhead relative to vector-time (a well-known logical timestamp method).

Index Terms—Model-based Debugging, Model-driven Development, UML-RT, Real-time and Embedded systems, MDD, MDE

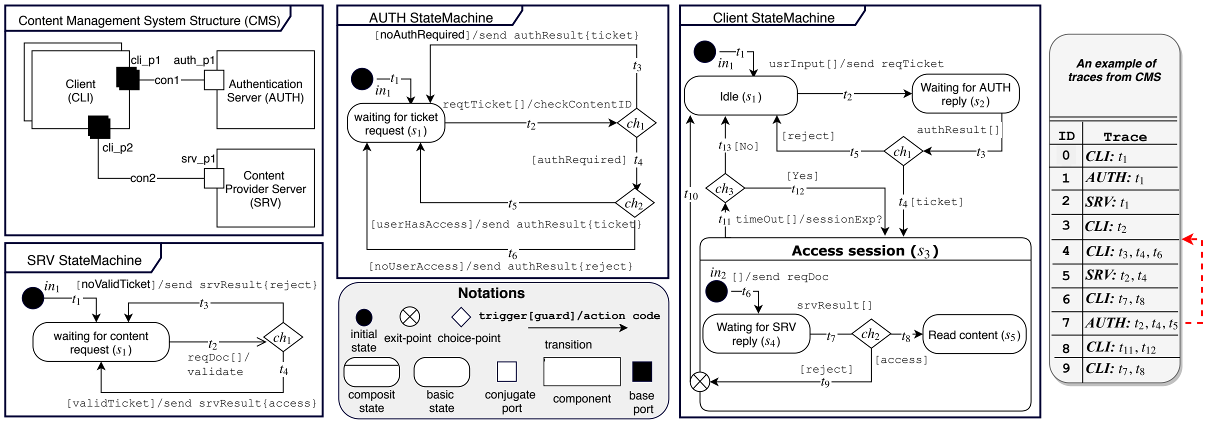
1 Introduction

Debugging by replay [1] is one of the most efficient debugging methods for software systems. It allows developers to execute the recorded traces of a system repeatedly and make diagnostic observations. However, supporting debugging of distributed systems via replay poses multiple challenges: First, it requires efficient mechanisms for generating useful traces and collecting them from possibly many nodes (e.g., [2]). Second, trace replay must also be deterministic (i.e., repeatable) analyzed and present a view consistent with the true state of the (usually concurrent and non-deterministic) system execution.

Thus, for distributed systems, deterministic replay of traces requires trace ordering techniques. Generally, these techniques can be classified into two main groups. **(1) Physical-time based approaches** use either local or global clocks, e.g., [3], [4], [5]. They annotate the traces with timestamp values obtained from either a local clock at each node, or a synchronized global clock. These approaches are mostly inaccurate, often resulting in inconsistent ordering. Also, the implementation of accurately synchronized clocks, with high precision, is expensive and difficult [6], [7]. **(2) Logical-clock based approaches** use the notion of *causality* to mitigate these problems using either Totally Ordered Logical Clocks ($TOLC$) or Partially Ordered Logical Clocks (*POLC*) [8], [9], [7]. Basically, they guarantee that timestamp values associated with traces is consistent with the underlying causality among them. *TOLC*s maintain a global event counter for all processes, whereas in *POLC-based* methods, an array of counters called *vector-times* is associated with each process. The size of a timestamp for *POLC* increases with the number of nodes in a distributed system, which may cause significant network overhead. Consequently, improvements to the efficiency and accuracy of ordering approaches has remained an active research area over the last four decades.

This paper concerns the ordering and replaying of execution traces in the context of Model-driven Development (MDD) of distributed systems. MDD is a software development approach that promotes the use of models as the primary software development artifact, rather than source code [10], [11]. By leveraging the abstract nature of models and existing model analysis and transformation techniques [12], we introduce a fully automated, efficient, and accurate method of trace collection and replay, without the use of timestamps. In our approach, we: (1) extend the model instrumentation approach introduced in [13], [14] to automatically instrument models of distributed systems and generate traces during execution. Also, we manage the size of traces by annotating them only with necessary data, such as node names and the current state of the execution. So, we generate relatively small traces without timestamps called Non-Timestamped (NT) traces. (2) replay NT traces deterministically. We introduce a new trace ordering technique for distributed systems, which relies on a causality analysis between traces and an abstract interpretation of models

Manuscript received April 19, 2005; revised August 26, 2015.


Content Management System Structure (CMS)
Client (CLI)
cli_p1
auth_p1
con1
Authentication Server (AUTH)
cli_p2
srv_p1
con2
Content Provider Server (SRV)
AUTH StateMachine
[noAuthRequired]/send authResult{ticket}
t_3
in_1
t_1
waiting for ticket request (s_1)
reqtTicket[]/checkContentID
t_2
ch_1
[authRequired] t_4
t_5
ch_2
[userHasAccess]/send authResult{ticket}
t_6
[noUserAccess]/send authResult{reject}
Client StateMachine
usrInput[]/send reqTicket
Idle (s_1)
Waiting for AUTH reply (s_2)
authResult[]
[reject]
t_{13} [No]
[Yes]
t_{12}
t_4 [ticket]
t_{10}
t_{11} timeOut[]/sessionExp?
Access session (s_3)
in_2 []/send reqDoc
srvResult[]
Wating for SRV reply (s_4)
Read content (s_5)
[access]
[reject]
t_9
SRV StateMachine
[noValidTicket]/send srvResult{reject}
waiting for content request (s_1)
reqDoc[]/validate
[validTicket]/send srvResult{access}
Notations
trigger[guard]/action code
initial state
exit-point
choice-point
transition
composit state
basic state
conjugate port
component
base port
An example of traces from CMS

ID	Trace
0	***CLI:*** t_1
1	***AUTH:*** t_1
2	***SRV:*** t_1
3	***CLI:*** t_2
4	***CLI:*** t_3, t_4, t_6
5	***SRV:*** t_2, t_4
6	***CLI:*** t_7, t_8
7	***AUTH:*** t_2, t_4, t_5
8	***CLI:*** t_{11}, t_{12}
9	***CLI:*** t_7, t_8

Figure 1: Simplified Content Management System

derived from traces. This technique accepts the NT traces, possibly unordered, as input and outputs partially ordered traces, which is sufficient for deterministic replay [15], [16].

We have applied our approach in the context of UML for real-time (UML-RT) and an extended version of Papyrus-RT, intended for distributed systems development [17], [18]. Our implementation is twofold: (1) We extended an existing model-level debugger called *MDebugger* [13] to implement trace generation and collection for distributed systems. (2) We developed an abstract interpreter for distributed UML-RT models that partially orders and replays NT traces. To maximize the impact of our work, our implementation only uses *open source* tools, including Papyrus-RT for modelling and code generation, and Epsilon [19] for model transformation and query.

The rest of this paper is organized as follows. In Section 2, we present our formalization and a running example. Section 3 describes our approach for the generation, collecting, and ordering of traces. We discuss our evaluation approach and results in Section 4. We review related work in Section 5, and conclude the paper with a discussion, summary, and directions for future research.

2 Background

In this section, we define, exemplify, and discuss the formalization used to specify and justify our approach. To illustrate our approach, we use the UML for real-time profile. UML-RT [20], [11] is a language specifically designed for Real-Time Embedded (RTE) 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, and it has a long and successful track record of application and tool support, via, e.g., IBM RSA-RTE [21] and Papyrus-RT [22]. Recently, an extension of Papyrus-RT, proposed in [18], allows the development of distributed systems using UML-RT. In the following, we provide a formalization of UML-RT for distributed systems. Our formalization of UML-RT is simplified and focuses on aspects specific to the instrumentation and abstract interpretation of models. Interested readers can refer to [11], [20], [23] to acquire more in-depth information regarding UML-RT.

2.1 Running Example

Fig. 1 shows the high-level design of a simplified Content Management System (CMS) that will be used as a running example. The *CMS* consists of an Authentication Server (i.e., *AUTH*), a Content Provider Server (i.e., *SRV*) and some Clients (i.e., *CLI*s). For a user to access a specific document from a *CLI* component, the *CLI* must first send a request to the *AUTH* component, which then checks the access policy of the document. If the document has no access restriction defined, *AUTH* generates an access ticket for the document. Otherwise, it validates the user and, if the validation is successful, generates an access ticket for the user. After receiving an access ticket, the *CLI* can send a request to the *SRV* that validates the ticket and provides the requested content only when the ticket is valid. Note that this system contains other user interaction components which are not discussed here. We assume that these omitted components forward users' input to the *CLI* when needed.

Let us assume that the system is deployed in a distributed environment (i.e., each component is deployed in a separate node), and its components generate execution traces for each transition. A sample stream of traces, ordered based on their incoming time, is shown in the rightmost side of Fig. 1. Interpreting the stream simply based on the their incoming order shows that a client received content (trace #6 t_7, t_8) before receiving a valid ticket (trace #7 t_2, t_4, t_6), which is incorrect. Detailed analysis reveals that trace #7 from *AUTH* arrived with a delay. The red dashed line shows the correct order of the traces. This hypothetical scenario is a typical situation when collecting traces from many nodes, and it can occur for a variety of reasons, e.g., delay in the network [24]. Most existing solutions for this problem can be categorized into two groups: physical-time based approaches and logical-time based approaches.

In *physical-time* approaches, the traces are annotated with a physical timestamp. In an ideal situation where clocks are synchronized and their precision is high, each trace can be annotated with a unique timestamp, then ordered and replayed in a deterministic way. However, providing synchronized clocks with high precision is a very challenging task, especially for heterogeneous systems [25], [26]. For this reason, it is possible that several traces are annotated with the same timestamp, or even with an incorrect timestamp. For example, the clock for the *SRV* may be behind the clock for the *CLI* (synchronization), or their clocks may have precisions on the order of milliseconds only, while both *SRV* and *CLI* are able to execute many instructions in a single millisecond.

Logical-time approaches use the causality relationship of traces to order them. For instance in the context of the running example, there is causality relationship between traces #5 and #6, i.e., the action of transition t_4 (*send srvResult*$\{$*access*$\}$) of *SRV* causes the execution of t_7 and t_8 of *CLI*. Logical-time approaches create timestamps that contain causality information. Vector time is a well-known logical timestamp method which preserves causal ordering by maintaining an array of counters of size N for each process in a distributed system with N processes. Whenever a process generates a trace, it increases its counter value. Whenever a process wants to send a message, it piggybacks the array of counters on the message and sends it to the target process. The receiver process updates each element in its array with the maximum value of the current element and the corresponding element in the piggybacked array [9].

In this work, we present a new way of ordering and replaying execution traces in the context of Model-driven Development of distributed systems. Similar to logical-time approaches, our approach is based on the causality relationship between traces. However, instead of using logical timestamps, we use abstract interpretation of models to order the traces, without the need for any timestamps.

2.2 Modelling of a Distributed System

Definition 1. (Read function (Projection)) Let tp be a tuple $\langle r_1 \dots r_n \rangle$ where $r_1 \dots r_n$ refer to the names of tuple's rows. We use $tp.r_i$ or $r_i(tp)$ to read the value of row r_i. E.g., to read the value of row $name$ of tuple $person\langle name, family\rangle$ we can use $person.name$ or $name(person)$.

Modelling 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 $d \in \{input, output\}$ 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 $\mathcal{P} \subseteq \boldsymbol{\mathcal{P}}$ (i.e., a universal set of ports) is a set of ports, $\mathcal{V}$ is a set of variables, and β refers to the specification of the capsule's behaviour. A port is defined as a pair $(t, conjugated)$, where $t \in \mathcal{I}$ refers to the type of the port, and $conjugated \in \{true, false\}$ specifies whether the port is conjugated (in UML-RT, connectors are always binary and run between ports that have the same type, but opposite conjugation). Finally, let us define the structure of a distributed system as a tuple $\langle \mathcal{C}, \mathcal{I}, con, in\rangle$, where $\mathcal{C}$ is a set of capsules, $\mathcal{I}$ is a set of protocols, con is a connectivity relationship $\subseteq \boldsymbol{\mathcal{P}} \times \boldsymbol{\mathcal{P}}$, and in is an acyclic containment relationship $\subseteq \mathcal{C} \times \mathcal{C}$.

Let us exemplify the above definition in the context of the running example. Each component is mapped to a capsule in UML-RT. Each *CLI* capsule has 2 base ports (*cli_p1*, *cli_p2*), and 2 connections *con1=(cli_p1, auth_p1)* and *con2=(cli_p2, srv_p1)*. *cli_p1* and *auth_p1* ports are typed with a protocol $\{$*(reqTicket, output), (authResult, input)*$\}$. *cli_p2* and *auth_p1* ports are typed with a protocol $\{$*(reqDoc, output), (srvResult, input)*$\}$. A sample message is *reqTicket* that conveys a *documentID* and *userInfo* as the payload.

Modelling Behaviour of a Component: We specify the behaviour of a capsule c using a UML-RT state machine (USM) that is defined as a tuple $\langle \mathcal{S}, \mathcal{T}, in\rangle$. $\mathcal{S} = \mathcal{S}_b \cup \mathcal{S}_c \cup \mathcal{S}_p$ is a set of states, $\mathcal{T}$ is a set of transitions, and $in \subset \mathcal{S}_c \times (\mathcal{S} \cup \mathcal{T})$ denotes an acyclic containment relationship, States can be basic ($\mathcal{S}_b$), composite ($\mathcal{S}_c$), or pseudo-states ($\mathcal{S}_p$). Basic states are primitive states which remain active until an outgoing transition is triggered. Composite states encapsulate a sub-state machine. Pseudo-states are transient control-flow states. There are six kinds of pseudo-states, including $initial$, $choice\text{-}point$, $history$, $junction\text{-}point$, $entry\text{-}point$, and $exit\text{-}point$, (i.e., $\mathcal{S}_p = \mathcal{S}_{in} \cup \mathcal{S}_{ch} \cup \mathcal{S}_h \cup \mathcal{S}_j \cup \mathcal{S}_{en} \cup \mathcal{S}_{ex}$). Composite and basic states can have entry and exit actions that are coded using an action language. Action languages support primitive operations such as accessing/updating variables, arithmetic/logical expression, and sending/receiving messages. MDD tools provide action languages either by adapting a subset of well-known programming languages or by creating a specific action language. E.g., Papyrus-RT uses a subset of C++ as the action language, the *Alf* action language [27] is designed for UML, and *YAKINDU* [28] provides its own action language.

Let $inp(c)$ refer to the messages that can be received by capsule c. A transition t is a 5-tuple $(src, guard, trig, act, des)$, where $src, des \in S$ refer to the source and destination of the transition, respectively. Also, $guard$ is a logical expression coded using the action language, $trig \subset inp(c)$ is a set of messages that trigger the transition, and act is the transition's action code using the action language.

In the context of the running example, the USM of *AUTH* has 2 basic states (*s_1*, *s_2*), an initial state (*in_1*) and 2 choice-points (*ch_1, ch_2*). It also has 6 transitions *t_1, . . . ,t_6*. Transition *t_2* is triggered by message *reqTicket* and has function *checkContentID* as the action. Transitions *t_3,. . . ,t_6* have guards.

Execution of a USM: In this work, we use the execution semantics of UML-RT, which are discussed in [11], [23]. In general, the execution of capsules is managed by a Run-Time System (RTS) library, which defines one or more controllers. A controller is assigned to a physical thread, controls the execution of a set of capsules, and provides message-passing services to them. Also, in a distributed context, controllers can be assigned to different nodes. E.g., the bottom-right of Fig. 2 shows an example of a collection of capsules assigned different controllers which, in turn, are assigned to different nodes. An important characteristic of the UML-RT execution semantics is the run-to-completion assumption, which guarantees that an incoming message will be fully processed before processing of the next message is started.

The execution of a USM can be observed as a set of execution steps that is defined as a tuple $\langle cause,\ cs,\ cs', chain\rangle$, where $cause$ is a message that starts the execution step that moves the execution from cs, (i.e., its current execution state $\in \mathcal{S}_b \cup \mathcal{S}_{in}$) to cs', (i.e., a new execution state). A $chain$ is a sequence of actions, i.e., $\langle exit(cs), act(head), \cdots, entry(cs') \rangle$ that is applied during the execution step, where $head$ refers to the first transition in the chain such that $cause \in trig(head)$. To simplify, we assume that the execution of a USM starts from the initial state by receiving a $startUP$ message from the RTS, and reaches a basic state, then continues by applying chains based on the received messages. For instance, in the context of the running example, the execution of SRV starts from in_1 and reaches s_1 by applying a chain $\langle exit(in_1), act(t_1), entry(s_1) \rangle$. It then stops at s_1 until a $reqDoc$ message is received, then one of the following chains may be applied, depending on the $guards$ of $t3$ and $t4$. In both cases, the execution returns to state s_1.

(1) $\langle exit(s_1), act(t_2), act(t_3), entry(s_1) \rangle$
(2) $\langle exit(s_1), act(t_2), act(t_4), entry(s_1) \rangle$.

Execution Trace: We define an execution trace as a tuple $\langle timestamp,\ capID,\ kind,\ actID,\ payload \rangle$ which is generated according to the execution flow of the system. $timestamp$ is an optional field that contains either a physical or logical timestamp, and $capID$ refers to the identical ID of the capsule instance. $kind \in \{exit, entry, trans, msgSend\}$ denotes the kind of trace, where $exit$, $entry$, $trans$, $msgSend$ respectively correspond to the trace that is generated when executing an exit action, executing an entry action, executing a transition action, and sending a message. $actID$ contains the name of a state, transition, or message, depending on the kind of the trace. $payload$ is a set of additional fields that can be included in the trace, depending on the context of the execution, e.g., variable values. In the context of the running example, a trace $\langle time_1, SRV, trans, t_2 \rangle$ shows that at time $time_1$ the action of transition t_2 of capsule SRV is executed, and $\langle time_2, AUTH, msgSend, ticket,\ auth_p1 \rangle$ shows that at time $time_2$, capsule $AUTH$ sent a message $ticket$. The payload $auth_p1$ refers to a port, through which the message is sent.

Motivated by [11], we consider a USM well-formed when the following conditions hold: (1) Only transitions that start from a choice-point can have a guard, and no transition that starts from a pseudo-state can have a trigger. (2) There are no AND-states (orthogonal regions), UML concepts *fork*, *join*, *shallow history*, or *final states*. (3) Transitions cannot cross state boundaries, i.e., $\forall t \in \mathcal{T} : parent(t.src) = parent(t.des)$. Entry-point and exit-point states can be used to create transitions with different parents. (4) States do not have idle (*do*) actions. (5) There is no notation for history. Instead, any transition to a composite state is assumed to end in an implicit history state inside the composite state. (6) Triggers of transitions which start from the same basic or composite state must be disjoint, i.e., $\forall t1, t2 \in \mathcal{T} : t1.src = t2.src \wedge t1.src \notin \mathcal{S}_p \implies t1.trig \cap t2.trig = \emptyset$. (7) No pseudo-states, except choice-point states, can have more than one outgoing transition. (8) No composite state can have more than one initial state. This restriction also applies to the root of the USM.

2.3 MDebugger

For the instrumentation of distributed systems and collection of execution traces, our work relies on MDebugger

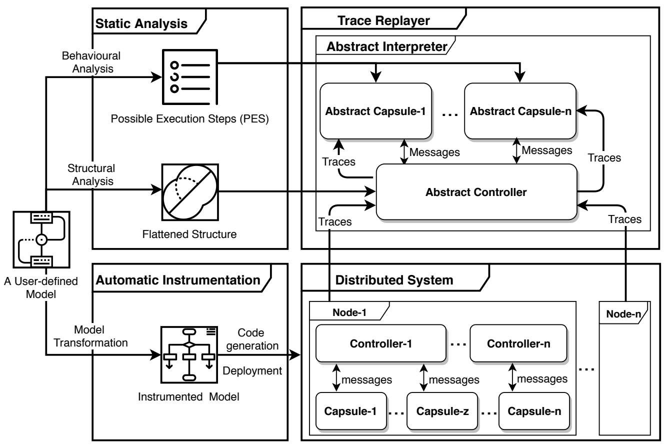


Figure 2: An Overview of Our Approach

[14], [13], [29]. MDebugger is an Eclipse-based model-level debugger for UML-RT that allows debugging of generated applications on their respective target platforms. Rather than code-level instrumentation, MDebugger uses model transformations to instrument models and generate instrumented code from the models. By default, MDebugger instruments models in a manner that supports a rich set of debugging features, such as breakpoints and variable modification at run-time. These features increase the complexity of models as well as the size of the generated code. However, in this work we apply only the model instrumentations required to generate and collect execution traces.

3 Approach

3.1 An Overview

Fig. 2 shows an overview of our approach to collecting and ordering execution traces of distributed systems in the context of MDD. Rather than annotating traces with physical/logical timestamps, which can be complex to implement, compute, and deliver, our approach uses the abstract interpretation of models and causality analysis to order and replay Non-Timestamped traces (NT). Fig. 2 shows an overview of our approach, which consists of three primary parts. **(1) Automatic Instrumentation:** We extend the model instrumentation approach introduced in [14], [13] to instrument models of distributed systems and generate the necessary execution traces. Executable code is generated from the instrumented models and deployed on a distributed platform, using [18]. **(2) Static analysis:** We perform static analysis on the model and extract the structural and behavioural information that is required for applying abstract interpretation. **(3) Trace Replayer:** We replay execution traces based on the results of the static analysis, using abstract interpretation. Essentially, we construct an abstract and centralized version of the distributed system that collects the (possibly unordered) traces and orders them partially using causality analysis.

Note that while we address instrumentation, static analysis, and replay to provide a comprehensive solution for debugging by replay, it is also possible to use each part of the approach individually in conjunction with other tools or approaches, including in different contexts. For example, the result of the static analysis can be used for symbolic execution.

3.2 Automatic Instrumentation

Our approach relies on a model instrumentation process introduced in MDebugger [13]. We extend $Model-to-Model(M2M)$ transformation techniques for the creation of an instrumented version of the user-defined model, allowing generation of execution traces at run-time on the generated code. The extension is threefold.

(1) Optimization of trace structure. In addition to removing timestamps, we also remove variable values from the traces, as these can be synthesized by abstract interpretation. In fact, we only trace their initial values and synthesize their subsequent values by interpretation of actions, whenever it is needed. Details of the synthesis are not discussed here, as this topic lies outside the scope of this paper.

(2) Minimizing the generation of traces. Based on the execution semantics of UML-RT discussed in Section 2, efficient construction of an execution step from traces in terms of size requires only three types of traces. (a) The act of the first transition (i.e., $act(head)$) in the chain, to detect whether the chain has been started, and (b) the act of transitions initiated from a choice-point, which allows the detection of a branched execution. (c) The entry actions of states inside composite states, which allows the handling the history states. For instance, in the context of the running example, to construct the execution step $\langle reqDoc,\, s_1,\, s_1, chain_1\rangle$ where $chain_1 = \langle exit(s_1),\ act(t_2),\ act(t_3),\ entry(s_1)\rangle$, only two traces are required, as follows. Note that $None$ denotes that the traces do not contain a timestamp.

$$(1)\ \langle None,\, SRV,\, trans,\, t_2\rangle \quad (2)\ \langle None,\, SRV,\, trans,\, t_3\rangle$$

(3) Collection of traces in the distributed mode. MDebugger only supports the collection of traces from a single node, using a debugging agent. To support distributed systems, we add debugging agents on each node, which act as a gateway to deliver traces to the monitoring application. Also, we add a globally unique capsule ID to each trace, which allows the identification of the source of each trace in monitoring applications.

3.3 Static Analysis

We perform static analysis on the model in order to analyze and extract the required information for the abstract interpretation. In addition, we flatten the hierarchical structure of the model into a new data form called Flattened Structure (FS) and calculate each capsule's Possible Execution Steps (PES). The former allows for the construction of a centralized image of a distributed system, and the latter enables the creation of an abstract version of each capsule instance for abstract interpretation. In the following, we discuss the details of extracting the aforementioned information.

3.3.1 Extract flattened structure

As shown in Fig. 2, the structure of a distributed system is designed hierarchically and deployed on possibly many nodes. Creating the same structure for abstract interpretation is a complicated and unnecessary task. Accordingly, we analyze the structure and deployment configuration of the distributed system, and we create a flat and centralized version in four steps. (1) Clone capsules based on their multiplicity, and set the multiplicity of each capsule to one. (2) Remove the containment relationship between capsules. (3) Adjust the connection relationship based on the results of step 2. (4) Create an address table for capsules, based on the deployment configuration and their ports.

Algorithm 1: $EPaths(SM : USM,\, s_0 : State,\, path : Sequence)$

```
T ← {t | (t ∈ SM.T) ∧ (t.src = s_0)}
paths ← ∅
forall t ∈ T do
    path ← append(path, t)
    if t.des ∈ S_b then
        paths ← paths ∪ path ∪ EPaths(t.des, ∅)
    else if t.des ∈ S_p then
        paths ← paths ∪ {EPaths(t.des, path)}
    else if t.des ∈ S_c then
        paths ← paths ∪ EPaths(t.des, ∅)
        S ← {s | (s ∈ s_0) ∧ (s ∈ t.des)}
        forall s ∈ child(t.des) do
            paths ← paths ∪ EPaths(s, path)
return paths
```

Algorithm 2: $PES(SM : USM)$

```
paths ← Epaths(SM, SM.Initial, ∅)
steps ← ∅
forall path ∈ paths do
    forall msg ∈ path[0].trig do
        step ←
          {msg, path[0].src, path[size(path) − 1].des, ∅}
        append(step.chain, exit(path[0].src))
        forall t ∈ path do
            append(step.chain, act(t))
            if t.des ∈ S_c then
                append(step.chain, entry(t.des))
        append(step.chain, entry(path[size(path) −
          1].des))
    steps ← steps ∪ step
return steps
```

3.3.2 Extract PES of each capsules' USM

Algorithms 1 and 2 show the functions $EPaths$ and PES which are used to extract all possible execution steps of an USM. First, Algorithm $EPaths$ extracts all possible execution paths (a sequence of transitions that starts from an initial, basic, or composite state and ends in a basic state), and then Algorithm PES populates all execution steps based on the results of $EPath$. Next, we discuss the details of these algorithms.

Algorithm $EPath$ recursively computes a set of possible execution paths starting from state s_0. For example, to extract all possible paths of SRV's USM, the Algorithm should be called as $EPath(SRV, in_1, \emptyset)$. It begins by extracting all outgoing transitions from s_0 (line# 1). Then, for each of the transitions, it creates a path by appending the transition and the next transitions until it reaches a basic state (line# 3-12). Depending on the destination of the transition, the path computation flow branches as follows. **(1) Basic state:** the path has ended, and the function returns a new path (line# 5-6). **(2) Pseudo-state:** the path is still partial. Thus, the function is called recursively by passing the partial path and the next pseudo-state (line# 7-8). **(3) Composite state:** as previously stated in Section 2, 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 (a child of the state). Thus, the function is called recursively by passing the partial path for all of the child states of the composite state (line# 9-12). Since a composite state can also be a source of transitions, the function is called recursively, in a manner similar to the case in which the transition source is a basic state (line# 10). In the context of the running example,

$$EPath\,(SRV,\, in_1,\, \emptyset) = \{\, \langle t_1 \rangle,\, \langle t_2,\, t_4 \rangle,\, \langle t_2,\, t_3 \rangle \}$$

Algorithm PES iterates over the paths returned by $EPath$ and computes possible execution steps of an USM (line# 3-12). Each path is converted to $1 \cdot\cdot n$ execution steps based on the number of messages in the trigger of the first transition of the path ($path[0]$) (line#4). Intuitively, the cause of each step is set to a triggering message, and cs and cs' are set, respectively, to the source and destination of the first and last transition of the path (line# 5). Also, the chain of the execution step is populated based on the actions of the transitions along the path and the entry and exit actions of their source and destination (line# 6-11). In the context of the running example

$$\begin{aligned} PES\,(SRV) = \{\, &\langle StartUP, in_1,\, s_1,\, \langle act(t_1), entry(s_1)\rangle\rangle, \\ &\langle reqDoc, s_1,\, s_s,\, \langle exit(s_1), act(t_2), act(t_3), entry(s_1)\rangle\rangle, \\ &\langle reqDoc, s_1,\, s_1,\, \langle exit(s_1), act(t_2), act(t_4), entry(s_1)\rangle\rangle \} \end{aligned}$$

3.4 Trace Replayer

To replay the traces, we create an abstract model interpreter, consisting of an abstract controller and abstract capsules. When the abstract interpreter starts, it creates abstract capsules based on the flattened structure of the distributed system, it creates a controller, and it sets up the communication between the controller and capsules. In the following, we discuss the details of the abstract controller and abstract capsules.

3.4.1 Abstract controller

The controller provides two main functionalities. (1) It collects traces from the deployed system, which is distributed. Then, it checks the traces and delivers them to the appropriate abstract capsule based on the trace $capID$. (2) It acts as a communication gateway for delivering messages between abstract capsules.

3.4.2 Abstract capsule

One abstract capsule is created per each capsule instance. Unlike their real capsule counterparts, abstract capsules do not have a USM or ports. Instead, they have a first-in-first-out queue for receiving traces, and a list that maintains all incoming messages. An abstract capsule orders the traces and replays them. Algorithm 3 shows the function

Algorithm 3: $AInterpret(cap : AbstractCapsule, step : PES)$

```
1  Let $traceQ$ be a FIFO queue that keeps incoming traces
     of $cap$
2  Let msgL be a list to keep incoming messages of $cap$
3  while (true) do
4      $currentStep \leftarrow \emptyset$
5      $traceB \leftarrow \emptyset$
6      repeat
7          $append(traceB, head(traceQ))$
8          $currentStep \leftarrow match(traceB, PES)$
9      until currentStep=$\emptyset$
10     while find(currentStep.cause,msgL)=false do
11         // wait until the causality of matchS be met
12     $remove(currentStep.cause, msgL)$
13     $replay(currentStep)$
```

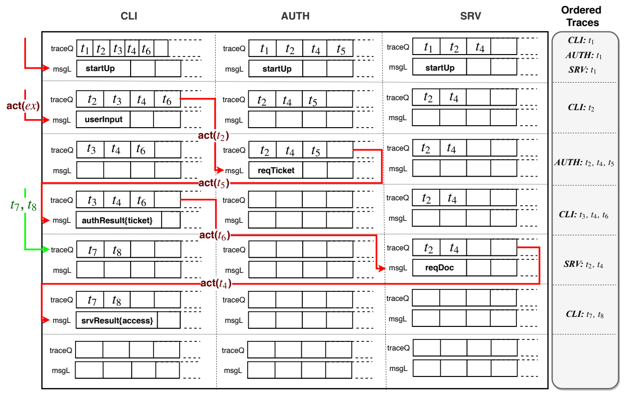


Figure 3: An Example of Ordering and Replaying

$AInterpreter$ which orders and replays the traces. The function runs in a loop, in which it first tries to read the traces from $traceQ$ and saves them in the $traceB$ buffer until only one execution step matches $traceB$. When an execution step is found, it checks the message in the $msgL$ list and waits until the $cause$ of the execution step is received (that is, the causality of the execution step is met). It then removes the $cause$ from the list and replays the execution step.

During the replay, all actions in the chain of the execution step must be interpreted. Depending on the goal of the interpretation, the actions are executed in different ways. E.g., if the goal is trace-ordering, the $replay$ functions only interpret communication actions, resulting in outgoing messages to the controller. The controller checks the messages, consults the address table to find the destination capsule, and delivers the message to the capsule. This drives the execution of dependent capsules that may be causally dependent on a given message.

3.4.3 Example of trace ordering and replay

Fig. 3 shows how Algorithm $AInterpreter$ orders and replays the sample stream of the execution traces (Fig. 1) in the context of the running example. Each row shows a step of the replay, columns 1, 2, and 3 show, respectively, how each abstract capsule for *SRV*, *AUTH*, and *CLI* interpret the related traces, the last column shows the replayed traces in each step, a red arrow denotes an incoming message to a capsule (the label on each arrow refers to the action which initiated the sending of the message), and finally the green arrow shows the flow of incoming traces. In each step, the contents of $traceQ$ and $msgL$ for each capsule is shown to provide details of the execution. **Step-1**, the traces of initial transitions for all capsules are replayed since their related causes are in their respective $msgL$. **Step-2**, traces of *SRV* and *AUTH* cannot be replayed since their causes are not available. Only trace t_2 of component *CLI* can be replayed since its cause is available. The red arrow from the left is a message initiated by a user-component interaction. As mentioned previously, the details of interactions between components and users are not discussed in this paper. **Step-3**, the replaying of $t2$ in step-2 sends a message to *AUTH*. In the current step, *AUTH* can replay traces of t_2, t_4, t_5, all of which belong to the same execution step. As discussed earlier (Ref. 2), this trace is out of order, and our approach can handle the out of order trace. **Step-4-6** These steps are similar to the previous steps. The exception is step-5, in which new traces are received by *CLI*.

4 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, use-cases, experimental protocol, hypotheses, and results.

4.1 Prototype Implementation

We implemented our approach in Eclipse Papyrus-RT for distributed systems [18], which is an industrial-grade and open-source MDD tool. We used the Epsilon Object Language (EOL) [19] to implement the transformation rules required for instrumentation of the models. EOL supports a set of instructions to create, query, and modify models expressed in languages described with the Eclipse Modeling Framework (EMF). The static analysis algorithms and abstract interpreter were developed using Java and the EMF API. For trace collection, code generation, and deployment, the default features of distributed Papyrus-RT are used.

Source code of the implementation along with documentation is available at [30], [22].

4.2 Hypotheses

We formulated the following three hypotheses to assess the performance and cost of our approach, as well as the benefits of our approach in comparison to trace replaying solutions based on logical timestamps.

Hypothesis 1 (Performance). As discussed earlier, our approach consists of three parts: static analysis of models, instrumentation of models, and replaying traces. It is important that each of these parts are applicable on any valid model within a reasonable amount of time. We hypothesize that performing each part of the approach on models can be done with reasonable performance.

Hypothesis 2 (Costs). In general, replaying of traces requires instrumented applications that generate execution traces. Instrumentation of applications increases their size, and generation of execution traces imposes performance overhead. We hypothesize that the costs associated with our approach are reasonable.

Hypothesis 3 (Benefits). We have introduced a new approach for the ordering and replaying of execution traces in the context of MDD. We hypothesize that our approach is efficient (with respect to the size of generated traces and the performance overhead of the instrumented application) in comparison to similar approaches that use logical timestamps.

4.3 Verification Approach

4.3.1 *Use-cases*

In order to verify our approach, several use-cases are used. As shown in Table 1, models have different complexities that range from simple models containing eight states to models with more than $2,304$ states. Simple models include the Content Management System, which is used as running example in this paper, and the Car Door Central Lock system. The Car Door Central Lock system is a control system for locking and unlocking car doors.

The Parcel Router [31], [32] is an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The system is time-sensitive and jams can appear due to variation in the time required by a parcel to transit through the different chutes. We used two different versions of the same system. The complete version checks for potential parcel jams and prevents parcels from being transferred from one chute to another until the next chute is empty. The simplified version ignores jams.

The Rover system model [33] 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.

The FailOver system [34], [35] is an implementation of the fail-over mechanism. It involves a set of servers processing client requests. To meet high availability, the system supports two replication modes, passive and active [36]. In passive replication, one server component works as the master, handling all the client requests while backup servers are largely idle, except for handshake operations. Whenever a malfunction occurs, resulting in a failure of the master server, a backup server is ranked up as the new master. In active replication, client requests are load-balanced between several servers. In addition to processing client requests, each server has to update its status to inform other servers of its availability. Therefore, each server can be notified whenever a malfunction causes the failure of one of its peers.

The refined FailOver system is a debuggable version of the FailOver system which is generated using MDebugger. The complexity of this model is high and allows us to check that the instrumentation and analysis time do not skyrocket when the model size grows exponentially.

Note that some of the above models are not models of distributed systems. We used them for the evaluation of static analysis and model instrumentation, where the distribution has no impact.

Table 1: Model Complexity, Instrumentation and Analysis Time, and Code Size of Original and Instrumented Models

Model	**Model Complexity**			**Instr. Time (ms)**	**Analysis Time**		**LOC**		
	C	S	T		PES	Flat.	Org.	Inst.	Over.
Content Management System	3	17	23	1115	1967	95	3255	3876	16
Car Door Central Lock	4	8	18	698	995	70	2072	2548	19
Simplified Parcel Router	8	12	14	926	1896	125	4727	5490	14
Parcel Router	8	14	25	1023	2337	229	5153	6478	21
Rover	6	16	21	980	2138	201	3881	4555	15
FailOver System	9	28	43	1150	2625	312	5074	6059	17
Refined FailOver System	13	2304	3647	2896	36033	986	12185	15736	23

4.3.2 Experimental setup

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

Applying Static Analysis and Model Instrumentation (EXP-1). To verify the performance of the model instrumentation, PES, and flattened structure algorithms, we applied each algorithm 20 times for each use case listed in Table 1 and averaged the time required for each algorithm/use case combination.

Collection and Replaying of Execution Traces (EXP-2). We extended our prototype to implement a vector-time approach in addition to our own approach. We then instrumented the FailOver system using both approaches. We ran each version using identical deployment configurations and collected 5 sets of execution traces with respective counts of 1000, 10000, 50000, 100000, 500000 traces. We measured the total size of the traces in both cases. We then used our prototype to replay the traces based on both approaches and measured the time required to do so.

Measuring Performance Overhead on Instrumented Application (EXP-3) Based on both versions of the FailOver system from EXP-2, we set up a benchmark for evaluating the performance of the uninstrumented FailOver system under normal execution (to show the real performance of the system) and the instrumented FailOver system using both approaches. In all cases, we executed the system until 10,000 messages were sent by the clients and processed by the servers. Then, based on the execution traces, we collected the computation time required for replying to a server request (i.e., $RequestReply$ transition), processing a message response by a client (i.e., $ProcessingResponse$ transition), and notifying each server's peers of its availability (i.e., $SendKeepAlive$ transition).

Measuring Application Size Overhead (EXP-4) For each use case listed in Table 1, we generated the code from both the original and instrumented models using our approach and measured the resulting lines of code for each case. In addition, we generated code from the FailOver system instrumented using vector-time and measured the resulting lines of code.

4.3.3 Rationale for comparison with vector time

In the same vein as logical-time approaches, our approach is based on the causality relation between traces. However, we use abstract interpretation as opposed to annotation of traces with logical timestamps. There are several logical-time approaches and it is not feasible to perform comparisons with all of them. We selected vector-time as a representative logical-time approach. Our rationale for choosing vector-time is its strong formal foundation. Existing methods in the literature have made similar comparisons in order to position their work, e.g,[24].

It worth mentioning that Charron-Bost showed in [37] that N is the lower-bound on the size of vector-time needed to characterize the underlying causality of a distributed system with N nodes [7]. Comparison with a method which has a defined lower-bound of timestamp size is a practical choice.

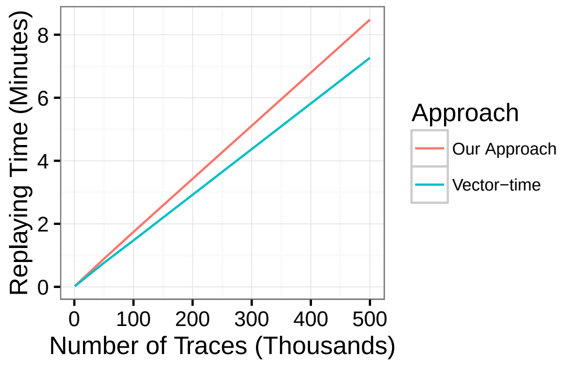


Figure 4: Replay Time of Execution Traces

4.3.4 Experimental Environment

We used a computer equipped with a 2.7 GHz Intel 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 1.7 GHz Intel Core i3 and 4GB. We used Java version 1.8.0_161 in configuration -Xmx12512m. The source code of the experiments and models are publicly available at [30] and can be used repeat our experiments.

4.4 Results and Discussions

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

Hypothesis 1 (Performance). Based on EXP-1, the *Inst. Time*, *PES. Time* and *Flat. Time* columns of Table 1 show the time required to instrument the models, create the PES, and flatten the models, respectively. In the worst case (i.e., the largest model), the instrumentation only takes around 3 seconds, PES takes 36 seconds and flattening takes less than a second. The order of time is seconds in all cases. It is therefore safe to assume that the computation time of these three parts is reasonable, and we consider the first part of Hypothesis 1 verified.

Based on the results of EXP-2, Fig. 4 shows the replaying time for sets of traces of size 10,000 - 500,000, using our approach and using vector-time. Unsurprisingly, our approach is slower than vector-time due to the abstract interpretation of traces. We argue that (1) the difference is not significant (i.e., our approach is 13% slower than vector-time). This is a reasonable cost for replaying the traces without timestamps, as removing timestamps from the traces results in decreased trace size and performance overhead. (2) The replaying of traces is usually a diagnostic activity which is performed offline for a small subset of the traces, so marginally slower replaying time should not cause problems in practice. From the above justification and a linear trend in the replaying time (i.e., Fig.4), we can conclude that second part of Hypothesis 1 is verified.

Hypothesis 2 (Cost). The LOC column of Table 1 shows the size of the generated generated code for original and instrumented models collected using EXP-4. The percentage of overhead ranged between 16% and 23%. Also, for the FailOver system, the overhead of our approach is less than the versions of the vector-time approaches (6486 LOC) due to the optimization we introduced for the generation of traces (Ref. Sec. 3). From this, we can conclude that the size overhead of our approach is reasonable and that the first part of Hypothesis 2 is verified.

Fig. 5 shows violin plots of computation times for three transitions (i.e., $RequestReply$, $ProcessingResponse$, $SendKeepAlive$). Computation times are recorded until 10,000 messages are processed. The wideness bars show the density of computation time in the specific range. As shown in Fig. 5, for all three transitions the system performance is impeded by the use of our approach. Using our approach, the majority of the *ProcessResponse* messages are processed within 0.3 ms to 0.48 ms, with an average time of 0.4 ms and a median time of 0.38 ms, which is close to the processing time when the system is in normal mode (average and median times of 0.34 and 0.31 ms respectively). The overhead for the *ProcessResponse* transition is within the range of microseconds and therefore negligible. The overhead is similar for *RequestReply* and *SendKeepAlive* messages. While the median and average of computation time for *RequestReply* is 44.38 ms and 42.22 ms, respectively, using our approach, the median and average in normal mode are 40.79 ms and 40.38 ms. For the *SendKeepAlive* transitions, the median and average using our approach are 0.075 ms and 0.89 ms, respectively, and in normal mode are 0.54 ms and 0.38 ms, respectively. In summary, we can argue that for each transition, the overhead of our approach is small, which is quite acceptable for many distributed systems. This verifies the second part of Hypothesis 2.

Hypothesis 3 (Benefits). Fig. 6 shows the size of the traces generated according to EXP-2. The size of generated traces significantly smaller, i.e., size of generated traces in the vector time approach is around 2.5 times the size of the traces generated by our approach. E.g., the size of 500,0000 generated traces annotated with vector time is 96MB, while it is 37MB with our approach. This verifies the first part of Hypothesis 3.

Concerning the performance overhead, as we discussed Fig. 5 also shows violin plots of the computation times for the three transitions in the vector time approach. As shown, the differences between our approach and vector time are not significant enough to derive any conclusion. This means that the performance overhead of both approaches is almost the same and therefore second part of Hypothesis 3 is not verified.

5 Related Work

To the best of our knowledge, we are the first to address the ordering/replaying of traces of distributed systems in the context of MDD. Despite this, there is a significant amount of related work in the areas of distributed systems, abstract interpretation, and debugging by replay. This Section discusses the most relevant work, organized into three groups. (1) abstract interpretation (2) ordering of traces of distributed systems (3) debugging by replay.

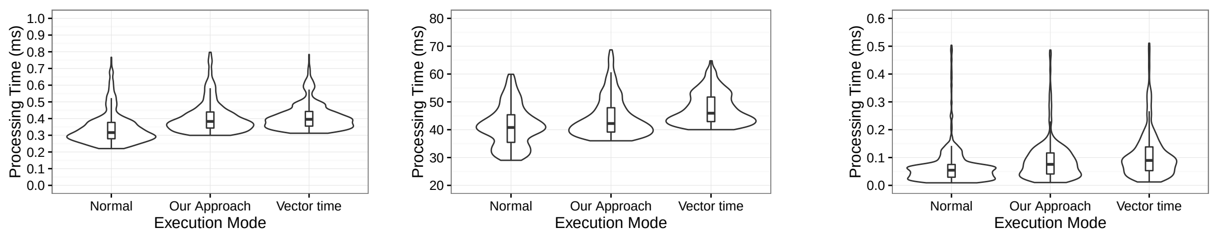
Abstract Interpretation. This group consists of research that applies abstract interpretation which is concerned with the approximation of software semantics [38]. E.g, (1) *Astrée*[39] and *Thesee* [40] use abstract interpretation to prove the absence of run-time errors for synchronous embedded C programs. (2) Zakeryfar. et.al [41] uses abstract interpretation to order synchronous events in the context of process-oriented programming languages such as Erasmus [42], [43]. This work is similar to ours, but our work addresses asynchronous communicating components in the context of MDD.

Ordering of Traces of Distributed Systems. We further classify work in this group into physical time, logical time and hybrid, based on the type of clock used.

(1) Physical-time approaches. The majority of physical time 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 synchronized clocks [44]. For example, Adya et al. proposed a concurrency control method based on multipart physical-time timestamps [26], [45]. Also, *ORDO*, proposed by Kashyap et al. [46], relies on invariant hardware clocks to provide the notion of a global hardware 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 [47].

(b) Tightly synchronized clocks. Network Time Protocol *(NTP)* and IEEE 1588 Precise Time Protocol *(PTP)* are among the most commonly used protocols for clock synchronization between computer systems [48], [49]. However, both methods are bounded by the limitation 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 [25].



(a) ProcessResponse (b) RequestReply (c) SendKeepAlive

Figure 5: Performance Overhead of Our Approach and Vector Time

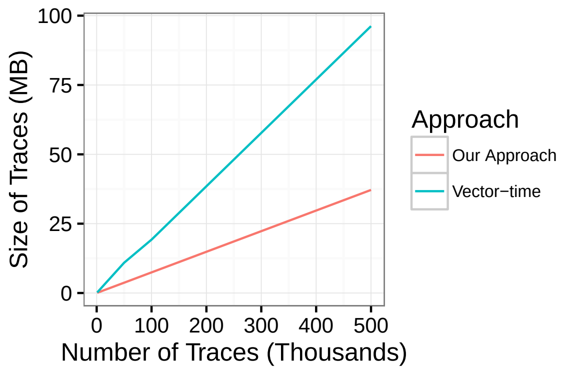


Figure 6: Size of Execution Traces

(2) Logical-time approaches We classify work in this group into the following classes based on the relation they use for the ordering of traces.

(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 [8]. Since then, several applications have resolved conflicting operations using the HB relation [50], [51], [52], [53]. For example, *Virtual-time*, proposed by Jefferson [54], implemented Lamport's conditions using a *Time Warp* mechanism. Since the HB relation is transitively closed, its computation time can be expensive [55]. Smaragdakis et al. [51] 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* [50], *DroidRacer* [56] and *SparseRacer* [57], 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. [58] 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. [59], introduced a data structure to compute the set of candidate events.

(b) Vector time. The concept of vector time was introduced by Mattern [60], [7] and Fidge [9] as a data structure for representing causality relationships among events. Charron-Bost [37] showed that the size of vector-time must be as great as the number of parallel processes. Since then, several methods have been proposed, intended to reduce the size of vector-time [61], [62], [63]. For example, Singhal and Kshemkalyani [64] introduced an enhanced version of vector-time which reduces the message-passing overhead at the expense of memory consumption. Also, Rodrigues and Verissimo [65] compressed causal information using extracted knowledge from the underlying network. This approach was leveraged in the $inline$ timestamps method proposed by Kulkarni et al. [24] in which vector-time computation is delayed in order to reduce the size of vector-time. Shen et al. [66] presented an encoded vector-time technique for large-scale distributed systems which optimizes the time and space complexity of vector-time.

(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 [67], [68], [69]. 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 [70] two protocols that provide scalable causal consistency using two-dimensional dependency matrices without relying on the transitivity of causality.

(3) 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 [71]. Kulkarni et al. [72] 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 [73]. Recently, Yingchareonthawornchai et al. [74] proposed a bounded size hybrid clock solution using a combination of *HLC* [72] and Hybrid Vector Clocks *(HVC)* [75]. They showed that their method is able to generate all possible snapshots from a distributed system at any given time [74]. Recently, *Retroscope* was proposed by Charapko et al. [44] in order to generate lightweight and consistent snapshots.

Debugging by Replay in the Context of MDD There are some works that address debugging by replay specifically for RTE systems in the context of MDD. For instance, Iyengar et al. [76], [77], [78] propose an optimized model-based debugging technique for RTE systems with limited memory. They use a monitor on the target platform to collect the generated traces and a debugger (executed on a host with sufficient memory) to replay the traces off-line and to display results on the model elements. Das et al. [79] propose a configurable tracing tool based on LTTng. They rely on code instrumentation to produce tracepoints useful for LTTng.

6 Conclusion

In this paper, we presented an approach for ordering and replaying of execution traces of distributed systems in the context of MDD. In contrast to existing approaches that rely on physical/logical timestamps, our approach relies on the abstract interpretation of models. To do so, (1) we instrument models of distributed systems using model transformation techniques to create instrumented models. The system generated by the instrumented models then produces execution traces. The structure and quantity of traces are optimized to minimize both the performance overhead on the distributed system and the size of the generated traces. (2) We use static analysis techniques on models of the distributed system to extract all possible execution steps of models annotated with causality information, and we create a flat structure which is suitable for abstract interpretation. This information is used by the abstract interpreter to clone a distributed system that accepts execution traces and will order and replay the traces based on the causality relation between them. We also developed a prototype and conducted experiments to assess the cost and benefit of our approach compared to approaches using vector-time (a popular timestamp approach). The experiments showed that our approach decreases the size of the traces significantly and imposes performance overhead similar to that using vector-time. The performance of ordering and replaying traces using our approach is slightly slower than the vector-time approach due to the abstract interpretation.

References

[1] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: A survey," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 17:1–17:47, Sep. 2015.

[2] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[3] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems*, M. K. Aguilera, L. Querzoni, and M. Shapiro, Eds. Cham: Springer International Publishing, 2014, pp. 17–32.

[4] S. D. Stoller, "Detecting global predicates in distributed systems with clocks," *Distributed Computing*, vol. 13, no. 2, pp. 85–98, Apr 2000.

[5] Y. Chen, T. Chen, and W. Hu, "Global clock, physical time order and pending period analysis in multiprocessor systems," *CoRR*, vol. abs/0903.4961, 2009. [Online]. Available: http://arxiv.org/abs/0903.4961

[6] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *Computer*, vol. 23, no. 10, pp. 33–42, Oct. 1990. [Online]. Available: http://dx.doi.org/10.1109/2.58235

[7] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 7, no. 3, pp. 149–174, Mar 1994.

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

[9] C. Fidge, "Fundamentals of distributed system observation," *IEEE Software*, vol. 13, no. 6, pp. 77–83, Nov 1996.

[10] B. Selic and O. Limited, "Using uml for modeling complex real-time systems," 1998.

[11] E. Posse and J. Dingel, "An executable formal semantics for UML-RT," *Software and Systems Modeling*, vol. 15, no. 1, pp. 179–217, 2016.

[12] N. Kahani and J. R. Cordy, "Comparison and evaluation of model transformation tools," Technical Report 2015-627, Tech. Rep., 2015.

[13] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 419–430.

[14] M. Bagherzadeh, N. Hili, D. Seekatz, and J. Dingel, "Mdebugger: A model-level debugger for uml-rt," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 97–100.

[15] J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. V. Kalé, "Scalable replay with partial-order dependencies for message-logging fault tolerance," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2014, pp. 19–28.

[16] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable deterministic record-replay," in *In International Conference on Supercomputing*, 2011.

[17] B. Selic, "Using UML for modeling complex real-time systems," in *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, 1998, pp. 250–260.

[18] Karim Jahed, "Papyrus-RT Distribution," https://github.com/kjahed/papyrusrt-distribution, 2019, retrieved: 2019-02-10.

[19] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[20] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. John Wiley and Sons New York, 1994, vol. 2.

[21] IBM, "IBM RSARTE," https://www.ibm.com/developerworks/community/wi 2016, retrieved July 19, 2016.

[22] E. Foundation, "Eclipse Papyrus for real time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, 2016, retrieved June 5, 2019.

[23] M. von der Beeck, "A formal semantics of uml-rt," in *Model Driven Engineering Languages and Systems*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 768–782.

[24] S. S. Kulkarni and N. H. Vaidya, "Effectiveness of delaying timestamp computation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 263–272.

[25] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, "Globally synchronized time via datacenter networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 454–467.

[26] A. Adya and B. Liskov, "Lazy consistency using loosely synchronized clocks," in *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '97. New York, NY, USA: ACM, 1997, pp. 73–82.

[27] Object Management Group, "About the Action Language for foundational UML Specification Version 1.1," https://www.omg.org/spec/ALF/About-ALF/, 2019, retrieved February 5, 2019.

[28] ITEMIS AG, "Yakindu StateChart Tool," https://www.itemis.com/en/yakindu/statechart-tools, 2016, retrieved July 19, 2016.

[29] M. Bagherzadeh, N. Hili, and J. Dingel, "Mdebugger repository," https://github.com/moji1/MDebugger, 2017, retrieved June 5, 2017.

[30] Blind For Review, "Model-Aware Abstract Interpreter," , 2019, retrieved: 2019-02-10.

[31] W. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Communications of the ACM*, vol. 25, no. 7, pp. 438–440, 1982.

[32] J. Magee and J. Kramer, *State Models and Java Programs*. Wiley, 1999.

[33] R. Ahmadi, N. Hili, L. Jweda, N. Das, S. Ganesan, and J. Dingel, "Run-time Monitoring of a Rover: MDE Research with Open Source Software and Low-cost Hardware," in *Workshop on Open Source for Model-Driven Engineering (OSS4MDE'16)*, 2016.

[34] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive failover for real-time middleware with

passive replication," in *15th IEEE Symposium on Real-Time and Embedded Technology and Applications*. IEEE, 2009, pp. 118–127.

[35] N. Kahani, N. Hili, J. R. Cordy, and J. Dingel, "Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems," in *Proceedings of the 9th International Workshop on Modelling in Software Engineering*. IEEE Press, 2017, pp. 12–18.

[36] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[37] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Information Processing Letters*, vol. 39, no. 1, pp. 11 – 16, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002001909190055M

[38] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the 4th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), Los Angeles, CA*. ACM, 1977, pp. 238–252.

[39] D. Monniaux, "The parallel implementation of the astrée static analyzer," in *Programming Languages and Systems*, K. Yi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 86–96.

[40] A. Miné, "Static analysis of run-time errors in embedded critical parallel c programs," in *Programming Languages and Systems*, G. Barthe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 398–418.

[41] M. Zakeryfar, "Static analysis of a concurrent programming language by abstract interpretation," Ph.D. dissertation, Montreal, Quebec, Canada, 2014.

[42] M. Zhu, P. Grogono, and O. Ormandjieva, "Exploring relationships between syntax and semantics of a process-oriented language by category theory," *Procedia Computer Science*, vol. 109, pp. 241 – 248, 2017, 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917310116

[43] Peter Grogono, "The Erasmus Project," URL: https://users.encs.concordia.ca/ grogono/Erasmus/erasmus.html, 2019, retrieved: 2019-02-10.

[44] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni, "Retrospective lightweight distributed snapshots using loosely synchronized clocks," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 2061–2066.

[45] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," *SIGMOD Rec.*, vol. 24, no. 2, pp. 23–34, May 1995.

[46] S. Kashyap, C. Min, K. Kim, and T. Kim, "A scalable ordering primitive for multicore machines," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 34:1–34:15.

[47] Y. Yamada and T. Nakajima, *Experiments of distributed ledger technologies based on global clock mechanisms*, ser. Studies in Computational Intelligence. Springer-Verlag, 1 2018, pp. 436–445.

[48] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, Oct 1991.

[49] "Ieee standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–300, July 2008.

[50] P. Maiya and A. Kanade, "Efficient computation of happens-before relation for event-driven programs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 102–112.

[51] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," *SIGPLAN Not.*, vol. 47, no. 1, pp. 387–400, Jan. 2012.

[52] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 121–133.

[53] M. Christiaens and K. De Bosschere, "Trade, a topological approach to on-the-fly race detection in java programs," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267847.1267862

[54] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[55] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 369–384.

[56] A. K. Pallavi Maiya and R. Majumdar, "DroidRacer," http://www.iisc-seal.net/droidracer, 2014, retrieved: 2019-01-17.

[57] A. Santhiar, S. Kaleeswaran, and A. Kanade, "Efficient race detection in the presence of programmatic event loops," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 366–376.

[58] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," *SIGPLAN Not.*, vol. 50, no. 10, pp. 332–348, Oct. 2015.

[59] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 193–205, Apr. 2017.

[60] F. Mattern, "Virtual time and global states of distributed systems," in *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 1988, pp. 215–226.

[61] S. Meldal, S. Sankar, and J. Vera, "Exploiting locality in maintaining potential causality," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '91. New York, NY, USA: ACM, 1991, pp. 231–239.

[62] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: constant size logical clocks for distributed systems," *Distributed Computing*, vol. 12, no. 4, pp. 179–195, Sep 1999. [Online]. Available: https://doi.org/10.1007/s004460050065

[63] V. K. Garg and C. Skawratananond, "String realizers of posets with applications to distributed computing," in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '01. New York, NY, USA: ACM, 2001, pp. 72–80.

[64] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Inf. Process. Lett.*, vol. 43, no. 1, pp. 47–52, Aug. 1992. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(92)90028-T

[65] L. E. T. Rodrigues and P. Verissimo, "Causal separators for large-scale multicast communication," in *Proceedings of 15th International Conference on Distributed Computing Systems*, May 1995, pp. 83–91.

[66] M. Shen, A. Kshemkalyani, and A. Khokhar, "Detecting unstable conjunctive locality-aware predicates in large-scale systems," in *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, June 2013, pp. 127–134.

[67] S. K. Sarin and N. A. Lynch, "Discarding obsolete information in a replicated database system," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 39–47, Jan 1987.

[68] G. T. Wuu and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '84. New York, NY, USA: ACM, 1984, pp. 233–242.

[69] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. T. Rodrigues, "On the use of clocks to enforce consistency in the cloud," *IEEE Data Eng. Bull.*, vol. 38, pp. 18–31, 2015.

[70] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:14.

[71] D. N. Nguyen, A. Charapko, S. S. Kulkarni, and M. Demirbas, "Optimistic execution in key-value store," *CoRR*, vol. abs/1801.07319, 2018.

[72] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Principles of Distributed Systems*, M. K. Aguilera, L. Querzoni, and M. Shapiro, Eds. Cham: Springer International Publishing, 2014, pp. 17–32.

[73] M. Demirbas and S. Kulkarni, "Highly auditable distributed systems," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/demirbas

[74] S. Yingchareonthawornchai, D. N. Nguyen, S. S. Kulkarni, and M. Demirbas, "Analysis of bounds on hybrid vector clocks," *IEEE*

Transactions on Parallel and Distributed Systems, vol. 29, no. 9, pp. 1947–1960, Sep. 2018.

[75] M. Demirbas and S. Kulkarni, “Beyond truetime : Using augmentedtime for improving spanner,” in *7th Workshop Large-Scale Distributed System Middleware*, 2013, pp. 1–5.

[76] P. Iyenghar, C. Westerkamp, J. Wuebbelmann, and E. Pulvermueller, “A model based approach for debugging embedded systems in real-time,” in *10th ACM Int. Conference on Embedded Software*, 2010, pp. 69–78.

[77] P. Iyenghar, E. Pulvermuelle, C. Westerkamp, M. Uelschen, and J. Wuebbelmann, “Model-based debugging of embedded software systems,” *Gesellschaft Informatik (GI)-Softwaretechnik (SWT)*, 2011.

[78] P. Graf and K. D. Muller-Glaser, “Dynamic mapping of runtime information models for debugging embedded software,” in *17th IEEE International Workshop on Rapid System Prototyping, 2006*. IEEE, 2006, pp. 3–9.

[79] N. Das, S. Ganesan, L. Jweda, M. Bagherzadeh, N. Hili, and J. Dingel, “Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation,” in *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 36–43.