Efficient Reordering and Replay of Execution Traces of Distributed Reactive Systems in the Context of Model-driven Development

Majid Babaei*
Queen's University
Kingston, Canada
babaei@cs.queensu.ca

Mojtaba Bagherzadeh
Queen's University
Kingston, Canada
mojtaba@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Canada
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 or physical timestamps. However, both kinds of timestamps have their drawbacks, including increased trace size. We examine the problem of determining consistent orderings of execution traces in the context of model-driven development of reactive distributed systems, that is, systems whose code has been generated from communicating state machine models. By leveraging key concepts of state machines and existing model analysis and transformation techniques, we propose an approach to collecting and reordering execution traces that does not rely on timestamps. We describe a prototype implementation of our approach and an evaluation. The experimental results show that compared to reordering based on logical timestamps using vector time (clocks), our approach reduces the size of the trace information collected by more than half while incurring similar runtime overhead.

1 INTRODUCTION

Debugging by replay [19] is one of the most common debugging methods for software systems [11], allowing developers to execute the recorded traces of a system repeatedly and make diagnostic observations. However, supporting debugging of distributed systems via replay is challenging: First, it requires efficient mechanisms for generating useful traces and collecting them from possibly many nodes (e.g., [18]). Second, 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 [15]. Examples include high-performance web-based applications such as Microsoft's Halo [63], 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 paper, we present the first results of our ongoing work to facilitate the development of these kinds of systems in general, and debugging in particular. 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.

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., [18, 52, 78]); however, the costs associated with keeping clocks sufficiently precise and synchronized can be significant [69, 71]. (2) Logical: Traces are annotated with counter values [29, 54, 71], 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 [82].

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

Leverage abstraction and automation with MDD: MDD is a software development approach that promotes the use of models as the primary software development artifact [68, 74]. In our work code is generated from models using an open-source MDD tool [48] such that it can be executed on different nodes in a distributed system. The behaviour of components is described using communicating state machines.

Reduce number and size of traces by avoiding timestamps: To sidestep the above-mentioned costs associated with timestamps

*Mr. Babaei is the corresponding author

MODELS'20, Oct 2020, Montreal, Canada

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. 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 [81], but also underlies many languages (or language extensions) built on the actor model such as Akka or Orleans, as well as many dynamic languages relying on events and event loops such as JavaScript or E [4, 14, 24, 64]. Many of these languages have successfully been used to implement industrial reactive systems [14, 63]. 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 trace information and instrumentation required which helps keep runtime overhead low.

We have implemented our approach in the context of the UML-RT profile in a prototype called *MReplayer*. 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 ongoing efforts to provide bug detection and correction support for models of distributed systems, and realize more sophisticated debugging services as described in [10, 19, 58]. MReplayer uses existing model-to-model transformation techniques [46] 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 then 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 paper is organized as follows. Section 2 presents some necessary background. Section 3 describes our approach. The evaluation, experimental results, and threats to validity are summarized in Section 4. Related work is reviewed in Section 5.

2 BACKGROUND

To illustrate and implement our approach, we use UML for Real-time (UML-RT) [68, 73], a language specifically designed for real-time embedded systems with soft real-time constraints. UML-RT has a long and successful track record of industrial application and tool support, via, e.g., IBM RSA-RTE [43], HCL RTist [38], Protos eTrice [26] and Papyrus-RT [31]. Recently, an extension of Papyrus-RT has been developed that allows the development of distributed systems with UML-RT [48]. Our work builds on this extension. Below, the most relevant aspects of UML-RT will be described. More details can be found in, e.g., [68, 73, 85].

```
1  CLI | 14:15:23.060 | t1 | in1 | s1  |
2  AUTH| 14:15:23.061 | t1 | in1 | s1  |
3  SRV | 14:15:23.062 | t1 | in1 | s1  |
4  CLI | 14:15:23.063 | t2 | s1  | s2  | reqTicket
5  CLI | 14:15:23.067 | t3 | s2  | ch1 |
6  CLI | 14:15:23.068 | t4 | ch1 | s3  |
7  CLI | 14:15:23.069 | t6 | in2 | s4  | reqDoc
8  SRV | 14:15:23.070 | t2 | s1  | ch1 |
9  SRV | 14:15:23.071 | t4 | ch1 | s1  | srvResult(access)
10 CLI | 14:15:23.072 | t7 | s4  | ch2 |
11 CLI | 14:15:23.073 | t8 | ch2 | s5  |
12 AUTH| 14:15:23.064 | t2 | s1  | ch1 |
13 AUTH| 14:15:23.065 | t4 | ch1 | ch2 |
14 AUTH| 14:15:23.066 | t5 | ch2 | s1  | authResult(ticket)
```

Listing 1: Sample sequence of tracing information for CMS

2.1 MDD for Distributed Systems: Example

Fig. 1 shows the design of a Content Management System (CMS) with Authentication Server (i.e., *AUTH*), a Content Provider Server (*SRV*) and a Client (*CLI*), as shown in the structure diagram in the top-left. The state machines, specifying the behaviour of the instances of a component, for *AUTH*, *SRV*, and *CLI* are shown in the top-centre, bottom-left, and centre-right of Fig. 1, respectively.

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.

Suppose the CMS 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 1 shows an example sequence of traces received by a debugging application, ordered by their arrival time at the debugger. 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 *CLI* took the transition t_1 from the source state in_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 *CLI*, *AUTH*, and *SRV*, respectively. The trace in line #4 indicates that *CLI* sends an

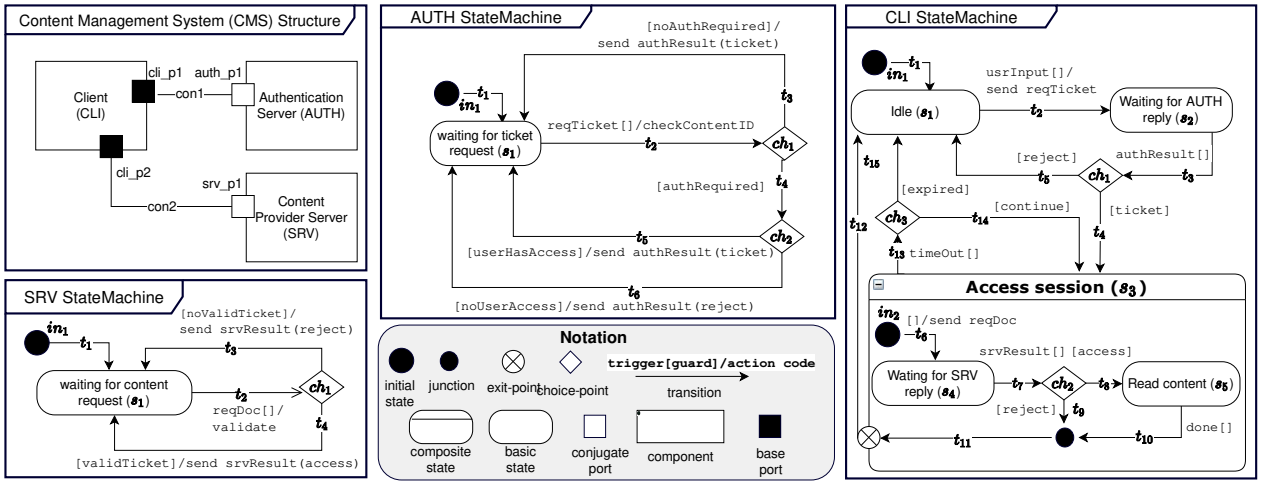


Figure 1: Models of Structure and Behaviour of Simplified Content Management System (CMS)

authentication ticket request to *AUTH*. Then, *CLI* receives the result of the authentication request (line #5), checks that it successfully received an access ticket (line #6), and sends a document request to *SRV* (line #7). Next, *SRV* validates the document request (line #8) and sends access to *CLI* (line #9). *CLI* then receives access from *SRV* (line #10) and reads the content (line #11). Finally, *AUTH* receives an authentication request from *CLI*, checks the access policy for the requested document (line #12), validates the user (line #13), and issues an access ticket (line #14).

The order in which traces arrive at the debugger shown in Listing 1 suggests that a client received content (traces in lines #10 and #11) *before* receiving a valid ticket (traces in lines #12, #13, and #14). However, this is, in this case, an incorrect conclusion and the late, out-of-order arrival of traces in lines #12-14 is due to other factors such as the delayed transmission of messages between *AUTH* and the trace collecting component. Network delay can cause traces of execution steps performed after some other steps (e.g., lines #5 to #11) to overtake these other traces (e.g., lines #12-14) and to still be received before them.

Timestamping traces can help detect such out-of-order delivery. Assuming that the time values included in the traces are correct, a debugger can use physical timestamp techniques to determine the proper order of messages and to, e.g., delay the replay of traces #5-11 until traces #12-14 have arrived. However, keeping clocks sufficiently precise and synchronized can be expensive, especially for large, heterogeneous systems with resource-constrained nodes [2, 57]. The need to postpone the replay of traces #5-11 can also be detected using logical timestamps such as vector clocks [2, 57]. 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 [82]. To satisfy Requirement 2, our approach does not use timestamps.

2.2 Modelling of Distributed Systems

In 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* (a.k.a., sub-capsules) [72, 73].

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 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. *(UML-RT state machine (USM))* The behaviour of a capsule is described with a UML-RT state machine (USM) defined as a tuple $\langle \mathcal{S}, \mathcal{T}, in \rangle$ where $\mathcal{S} = \mathcal{S}_b \cup \mathcal{S}_c \cup \mathcal{S}_p$ is a set of states, $\mathcal{T}$ is a set of transitions, and $in \subseteq \mathcal{S}_c \times (\mathcal{S} \cup \mathcal{T})$ denotes an acyclic containment relationship. States can be basic ($\mathcal{S}_b$), composite ($\mathcal{S}_c$), or pseudo-states ($\mathcal{S}_p$). 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., $\mathcal{S}_p = \mathcal{S}_{in} \cup \mathcal{S}_{ch} \cup \mathcal{S}_h \cup \mathcal{S}_j \cup \mathcal{S}_{en} \cup \mathcal{S}_{ex}$). Composite and basic states can have entry and exit actions that are coded using an action language. Action languages allow expressions (over primitive types such as boolean and integer), and primitive operations such as accessing and updating variables and sending messages.

Definition 3. *(Transition)* Let $inp(c)$ refer to the messages that can be received by capsule c. A transition t is a 5-tuple $\langle src, guard, trig, act, des \rangle$, where $src, des \in S$ refer to the source and destination of the transition, respectively. Also, *guard* is a boolean expression in the action language, $trig \subseteq inp(c)$ is a set of messages that

CLI
$rc_1^C = \langle in_1, startUp, \langle t_1 \rangle, s_1 \rangle$
$rc_2^C = \langle s_1, usrInput, \langle t_2 \rangle, s_2 \rangle$
$rc_3^C = \langle s_1, authResult, \langle t_3, t_5 \rangle, s_1 \rangle$
$rc_4^C = \langle s_2, authResult, \langle t_3, t_4, t_6 \rangle, s_4 \rangle$
$rc_5^C = \langle s_4, srvResult, \langle t_7, t_8 \rangle, s_5 \rangle$

AUTH
$rc_1^A = \langle in_1, startUp, \langle t_1 \rangle, s_1 \rangle$
$rc_2^A = \langle s_1, reqTicket, \langle t_2, t_3 \rangle, s_1 \rangle$
$rc_3^A = \langle s_1, reqTicket, \langle t_2, t_4, t_5 \rangle, s_1 \rangle$
$rc_4^A = \langle s_1, reqTicket, \langle t_2, t_4, t_6 \rangle, s_1 \rangle$

SRV
$rc_1^S = \langle in_1, startUp, \langle t_1 \rangle, s_1 \rangle$
$rc_2^S = \langle s_1, reqDoc, \langle t_2, t_3 \rangle, s_1 \rangle$
$rc_3^S = \langle s_1, reqDoc, \langle t_2, t_4 \rangle, s_1 \rangle$

Figure 2: Some of the rc-steps of the state machines in the CMS system

trigger the transition, and *act* is the transition's action code using the action language.

Example: 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 $t_1, \ldots, t_6$. Transition t_2 is triggered by message *reqTicket* and has the function *checkContentID* as action.

Run-to-completion (RTC) is a central concept in the definition of the execution semantics of many state machine languages including UML and UML-RT [28, 84]. It prevents the processing of an incoming message (possibly triggering a sequence of several transitions such as, e.g., $\langle t_4, t_5, t_6 \rangle$ and $\langle t_3, t_4, t_6 \rangle$ in the state machines of *AUTH* and *CLI*, respectively) 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 information.

Definition 4. *(RC-step)* An rc-step rc is a tuple $\langle \sigma, C, P, \sigma' \rangle$, where $\sigma \in \mathcal{S}_b \cup \mathcal{S}_{in}$ refers to the source state of rc, $\sigma' \in \mathcal{S}_b$ refers to the destination state of rc, C is a set of messages representing the trigger of t_1 (see below), and P is a finite sequence of transitions (path) $t_1 ... t_n, n \in \mathbb{N}$ that starts with transition t_1 (i.e., $t_1.src \in \sigma \cup parents^+(\sigma)$) and ends with t_n (i.e., $t_n.des \in \sigma' \cup children^+(\sigma)$). $children^1(\sigma)$ returns the states directly contained in σ, while $parents^1(\sigma)$ returns the state directly containing σ, if any. $children^+$ and $parents^+$ are the transitive closures.

Example: Some of the rc-steps of the capsules in the CMS system are shown in Figure 2.

Definition 5. (*Execution of an USM*) We define the execution of an USM using a Labelled Transition System $\langle \gamma_0, \Gamma, R \rangle$, where Γ is a set of configurations and R is a set of rc-steps. A configuration γ is a tuple $\langle \sigma, E \rangle$ where $\sigma \in \mathcal{S}$ is the active state (Definition 4) 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 γ_0 (Definition 7).

Definition 6. *(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., $exit(rc.\sigma)$), actions of transitions of rc (i.e., $act(t_1)$, ..., $act(t_n)$ where $t_1 ... t_n \in rc.P$), and entry actions of the destination state of rc (i.e., $exit(rc.\sigma')$).

Definition 7. (*Execution of an rc-step*) An rc-step rc is enabled by configuration γ and message m, when the active state ($\gamma.\sigma$) is the same as the source state of rc, the guard of the first transition in $rc.P.t_1$ holds using the value map of γ, and m triggers the step (i.e., $m \in rc.C$). Function *enabled*($\gamma \in \Gamma$, *rc, m)* returns *true* if step rc is enabled by configuration γ and message m, and *false* otherwise. When an rc-step is executed, its actions ($actions(rc)$) are executed which typically causes the value map E of the component to be updated (resulting in E'), outputs (messages) to be sent, and the current configuration of the USM to be changed to $\gamma' = \langle rc.\sigma', E' \rangle$.

Definition 8. *(Execution Trace)* Execution traces containing relevant runtime information are generated during system execution. We define a trace as a tuple $\langle L, P \rangle$, where P refers to the component id and L refers an executable element in the model (i.e., a state or transition) in the USM that is executed.

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 ensure that the trigger 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.

3 DESCRIPTION OF APPROACH

3.1 Overview

A graphical overview of the implementation of our approach is given in Figure 3 and will be discussed in Section 3.4. Our approach assumes that the traced system has been instrumented and generates execution traces at runtime. Other high-level, conceptual characteristics of the approach include:

3.1.1 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 *SRV*. 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 #3 of Listing 1 will always appear before the trace in line #8. Similarly, transition t_2 of *SRV* requires message *reqDoc* which is sent in transition t_6 of *CLI*. As a result, transition t_2 will not be allowed to occur during the replay until t_6 has occurred, and the trace in line #8 of Listing 1 will always appear after the trace in line #7.

3.1.2 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 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 γ. 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 1), it suffices to instrument the first transition of each rc-step only. So, instead of receiving the traces in Listing 1, our replayer would receive the sequence of traces T_{in} below in which, consistent with Definition 8, each trace identifies a transition in a capsule (with superscripts C, A, and S referring to capsules *CLI*, *AUTH*, and *SRV* respectively). The corresponding sequence of rc-steps recognized by the replayer is shown underneath:

$$\begin{array}{rcccccccc} T_{in} & = & t_1^C & t_1^A & t_1^S & t_2^C & t_3^C & t_2^S & t_7^C & t_2^A \\ & & rc_1^C & rc_1^A & rc_1^S & rc_2^C & rc_4^C & rc_3^S & rc_5^C & rc_3^A \end{array}$$

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 that the execution of rc will result in[1]. 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.

3.2 Static Analysis

A capsule's rc-steps are computed by function *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 *SRV*, the function *computeRCSteps* should be called as

computeRCSteps(SM, in_1, [], ∅)

[1]In the context of transition systems, this notion of equivalence between executions is typically formally captured with the notion of *confluence* as in, e.g., [83]

Algorithm 1: *computeRCSteps*(*SM* : USM, s_0 : state, P : sequence of transitions, *visited* : set of states)

```
1  T ← {t ∈ SM.T | t.src = s0}
2  forall t ∈ T do
3      P ← append(P, t)
4      if t.des ∈ Sb then
5          rcStep ← ⟨P[0].src, P[0].trig, P, P[size(P) − 1].des⟩
6          if t.des ∉ visited then
7              visited ← visited ∪ {t.des}
8              rcSteps ←
                 {rcStep} ∪ computeRCSteps(SM, t.des, ∅, visited)
9          else
10             rcSteps ← rcSteps ∪ {rcStep}
11     else if t.des ∈ Sp then
12         computeRCSteps(SM, t.des, P, visited)
13     else if t.des ∈ Sc then
14         S ← children¹(t.des)
15         forall s ∈ S do
16             visited ← visited ∪ {s}
17             computeRCSteps(SM, s, P, visited)
18         computeRCSteps(SM, t.des, ∅, visited)
19 return rcSteps
```

where *SM* and in_1 denote the state machine of *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) 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) 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).
(3) t.des is a composite state (line #13): as mentioned, 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).

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 9. *(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:

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

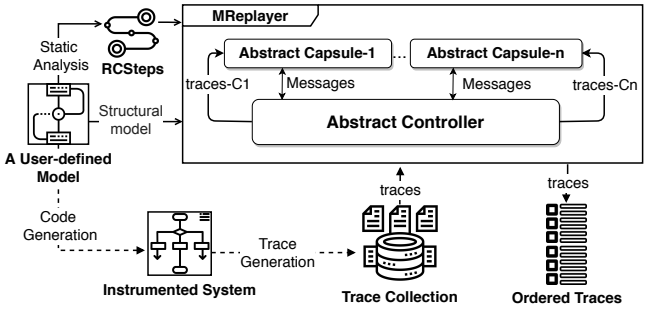


Figure 3: An overview of our approach

For *CLI* for example, $rc_1^C \rightarrow_{CFD} rc_2^C$ and for *SRV*, $rc_1^S \rightarrow_{CFD} rc_2^S$, $rc_1^S \rightarrow_{CFD} rc_3^S$, and $rc_2^S \rightarrow_{CFD} rc_2^S$. 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 10. *(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:

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

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

For example, for *SRV* and *CLI*, $rc_3^S \rightarrow_{CMD} rc_5^C$ and $rc_3^S \rightarrow_{CMD} rc_7^C$. Note that the sending and the receiving capsule in a communication dependency do not have to be different.

Both relations can be visualized using directed graphs as is standard in the literature on static analysis and program analysis [42, 61].

Definition 11. *(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 rc-steps $R_{T_2} = rc_1 \ldots rc_n$ corresponding to T_2 respects all control flow or communication dependencies, i.e.,

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

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

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

$$T_{out,1} \quad = \quad t_1^C \quad t_1^A \quad t_1^S \quad t_2^C \quad t_2^A \quad t_3^C \quad t_2^S \quad t_7^C$$

3.4 MReplayer

An overview of the approach and our replay component *MReplayer* is shown in Figure 3. 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.

3.4.1 Abstract Controller. The behaviour of the abstract controller is rather simple. It listens for traces from the deployed system, and for each traces that arrives it uses the component id P contained in the trace (Definition 8) 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.

3.4.2 Abstract Capsule. An abstract capsule[2] 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 5). As sketched above, C_a will replay every incoming trace, while postponing the replay whenever necessary.

To do so, function *replayRCSteps(s₀, rcSteps, inTraces, msgs, outTraces)* is used as shown in Algorithm 2. As a first step, *replayRCSteps* will use the boolean function *containsMatch* to find the first trace in the queue of incoming traces such that the transition t represented in the trace matches the current configuration, i.e., the source of t is active and the guards, if any, evaluate to *true* in *E*. *containsMatch* returns *true* if such a trace can be found, and *false* otherwise. For a boolean expression b, statement **await** b is assumed to block until b returns *true* (line #5). If a match has been found, the trace is stored and the rc-step corresponding to the trace is determined (lines #6-7). As mentioned, a trace can only ever match at most one rc-step. Boolean function *enabled(rcStep, msgs)* checks if the rc-step is enabled, i.e., if the queue of incoming messages *msgs* contains the trigger of the first transition of the rc-step *rcStep* (line #8). If not, the capsule will wait for the trigger to arrive. If yes, the message is retrieved and removed from the queue (line #9), and the trace and the rc-step are replayed resulting in a new configuration and, possibly, one or more messages to be sent to the controller (line #10). Then, the trace is appended to the output (line #11). We can see how the two **await** statements enforce control flow and communication dependencies respectively and postpone the replay of out-of-order traces.

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

[2] Strictly speaking, it should be called abstract capsule instance

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

```
1  Let γ range over configurations (Definition 5)
2  Let trace, rcStep, and msg range over execution traces, rc-steps, and
     messages respectively (Definitions 8,4, and 3)
3  γ ← γ0
4  while (true) do
5      await containsMatch(inTraces, rcSteps, γ)
6      trace ← getMatchingTrace(inTraces, rcSteps, γ)
7      rcStep ← getMatchingRCStep(trace, rcSteps)
8      await enabled(rcStep, msgs)
9      (msg, msgs) ← getAndRemove(trigger(rcStep), msgs)
10     γ ← replay(rcStep, msg, γ)
11     outTraces ← append(outTraces, trace)
```

3.5 Example

Figure 4 shows an execution of the replayer on the input trace T_{in} from Section 3.1.2. Initially, all abstract capsules wait for a trace to match (*status*=$wait_{cM}$). All state machines are in their initial states (in_1). Capsule *CLI* already has a *usrInp* message in its queue, but the queues of all other capsules are empty. In Step 1 input trace t_1^C is processed, causing the (abstract capsule of) *CLI* to replay rc-step rc_1^C which puts it into state s_1. Also, trace t_1^C is output. Similiarly for Steps 2 and 3. In Step 4, input t_2^C can be matched to rc_2^2 which is also found enabled and thus can be replayed, causing *CLI* to move to state s_2, message *reqTicket* be sent to *AUTH*, and t_2^C to be output. In Step 5, input t_3^C is matched to an rc-step of *CLI*, which cannot be executed because *CLI* has not received its trigger *authResult* yet. As a result, the output of t_3^C is postponed (till Step 9). In Step 6, the output of trace t_2^S is also postponed (till Step 10), because *SRV* is waiting for message *reqDoc*. In Step 7, input t_7^C is not even matched and remains in the input sequence. The arrival of t_2^A in Step 8 indicates that *AUTH* executed the rc-step starting with t_2 which can be replayed because the trigger (*reqTicket*) is in the message queue. The replay puts *authResult* into the queue of *CLI*, which enables t_3^C and the associated rc-step to be replayed (Step 9). No input is processed in this step, but *reqDoc* is sent to *SRC*. Similarly for Steps 10 and 11. The replayer has reordered the input sequence consistently and output the sequence $T_{out,1}$ given above.

3.6 Automatic Instrumentation

Our approach mostly relies on a model instrumentation process introduced in MDebugger [6, 7]. 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.

3.7 Extensions

3.7.1 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 8) that is used to track the values of variables that depend on non-deterministic statements. Assuming that the user identifies these statements, a dataflow analysis can be used to determine which variables are affected and add instrumentation code that ensures the inclusion of their values in every trace.

3.7.2 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 consistent reorderings of the input sequence T_{in}.

$$T_{out,2} = t_1^A\ t_1^C\ t_1^S\ t_2^C\ t_2^A\ t_3^C\ t_2^S\ t_7^C$$
$$T_{out,3} = t_1^A\ t_1^C\ t_2^C\ t_2^A\ t_3^C\ t_1^S\ t_2^S\ t_7^C$$

In total, T_{in} has 18 consistent reorderings. To extend our approach to output all consistent reorderings, the static analysis phase discussed in Section 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.

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 [48], which is an industrial-grade and open-source MDD tool. We used the Epsilon Object Language (EOL) [50] 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). Source code of the implementation along with documentation is available at [13, 31].

4.2 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 exponentially. We hypothesize that performing these steps on models with various complexities can be done within a reasonable amount of time.

			CLI			*AUTH*			*SRV*			
Step	*inT*	*rcStep*	*status*	*s*	*msgs*	*status*	*s*	*msgs*	*status*	*s*	*msgs*	*outT*
0			$wait_{cM}$	in_1	$\langle usrInp \rangle$	$wait_{cM}$	in_1	$\langle\rangle$	$wait_{cM}$	in_1	$\langle\rangle$	
1	t_1^C	rc_1^C	$replay(rc_1^C)$	s_1	$\langle usrInp \rangle$	$wait_{cM}$	in_1	$\langle\rangle$	$wait_{cM}$	in_1	$\langle\rangle$	t_1^C
2	t_1^A	rc_1^A	$wait_{cM}$	s_1	$\langle usrInp \rangle$	$replay(rc_1^A)$	s_1	$\langle\rangle$	$wait_{cM}$	in_1		t_1^A
3	t_1^S	rc_1^S	$wait_{cM}$	s_1	$\langle usrInp \rangle$	$wait_{cM}$	s_1	$\langle\rangle$	$replay(rc_1^S)$	s_1	$\langle\rangle$	t_1^S
4	t_2^C	rc_2^C	$replay(rc_2^C)$	s_2	$\langle\rangle$	$wait_{cM}$	s_1	$\langle reqTick \rangle$	$wait_{cM}$	s_1	$\langle\rangle$	t_2^C
5	t_3^C	rc_4^C	$wait_e(authRes)$	s_2	$\langle\rangle$	$wait_{cM}$	s_1	$\langle reqTick \rangle$	$wait_{cM}$	s_1	$\langle\rangle$	
6	t_2^S	rc_3^S	$wait_e(authRes)$	s_2	$\langle\rangle$	$wait_{cM}$	s_1	$\langle reqTick \rangle$	$wait_e(reqDoc)$	s_1	$\langle\rangle$	
7	t_7^C		$wait_e(authRes)$	s_2	$\langle\rangle$	$wait_{cM}$	s_1	$\langle reqTick \rangle$	$wait_e(reqDoc)$	s_1	$\langle\rangle$	
8	t_2^A	rc_3^A	$wait_e(authRes)$	s_2	$\langle authRes \rangle$	$replay(rc_3^A)$	s_1	$\langle\rangle$	$wait_e(reqDoc)$	s_1	$\langle\rangle$	t_2^A
9			$replay(rc_4^C)$	s_4	$\langle\rangle$	$wait_{cM}$	s_1	$\langle\rangle$	$wait_e(reqDoc)$	s_1	$\langle reqDoc \rangle$	t_3^C
10			$wait_{cM}$	s_4	$\langle srvRes \rangle$	$wait_{cM}$	s_1	$\langle\rangle$	$replay(rc_3^S)$	s_1	$\langle\rangle$	t_2^S
11		rc_5^C	$replay(rc_5^C)$	s_5	$\langle\rangle$	$wait_{cM}$	s_1	$\langle\rangle$	$wait_{cM}$	s_1	$\langle\rangle$	t_7^C

Columns *inT* and *rcStep* show the input trace processed and the rc-step matched in this step, respectively. For each abstract capsule, *status* indicates what the capsule is doing where $wait_{cM}$ = 'waiting for a match in the input traces' (line #5 in Algorithm 2), $wait_e(m)$ = 'waiting for message m' (line #8), and $replay(rc)$ = 'replaying rc-step rc' (line #11). Columns *s* and *msgs* indicate the new state and the sequence of incoming messages. *outT* shows the trace output in this step.

Figure 4: Sample Execution of MReplayer on CMS on the input trace in Section 3.1.2.

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.

4.3 Verification Approach

4.3.1 Use cases. To evaluate our approach, several use cases are used. The complexity of the models used (Table 1) ranges from a model with 3 capsules, 17 states, and 23 transitions (i.e., CMS) to a model with 13 capsules, 2,304 states and 3,647 transitions (i.e., RFO). Model for a simplified Content Management System (CMS) is described in Section 2, FailOver system (FO) [8, 47], Parcel Router (PR) [60, 79] and Rover control system (RO) are described in [5, 6]. Also, Refined FailOver (ROF) is a debuggable version of a FailOver system generated using MDebugger [6].

4.3.2 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 use case listed in Table 1 and averaged the times required for each step (i.e, instrumentation and static analysis) and each use case combination.

Collection of Execution Traces (EXP-2). We extended our prototype to implement the classical vector time approach. We then instrumented the use cases in Table 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.

Table 1: Complexity of Use-cases, Processing Time, Code Size Overhead, and Size of Collected Traces

Model	Complexity			Inst. (ms)	Stat. (ms)	LOC		SOT (MB)	
	C	S	T			Ours	VT	Ours	VT
CMS	3	17	23	1115	1967	8%	9%	5	13
CDCL	5	11	15	698	995	11%	12%	7	18
SPR	8	12	14	926	1896	11%	12%	11	30
PR	8	14	25	1023	2337	12%	16%	15	41
RO	6	16	21	980	2138	13%	17%	22	60
FO	7	31	43	1150	2625	15%	19%	35	95
RFO	13	2304	3647	2896	36033	20%	28%	53	144

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

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 use case listed in Table 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.

4.3.3 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, [53].

4.3.4 Experimental Environment. We used a computer equipped with a 2.7GHz 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 1.7GHz Intel Core i3 and 4GB. We used Java version 1.8.0161 in configuration -Xmx12512m. The source code of the experiments and models are publicly available at [13] and can be used to 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: Processing Time. Based on EXP-1, the instrumentation time (*Inst.*) and static analysis (*Stat.*)) columns of Table 1 show the time required to instrument the models and compute the *RCSteps* 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 FO to RFO, the number 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 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 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 an *RCStep*, 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 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 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 shows 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, 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 *ProcessResponse* 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 average 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.

4.5 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 [32] for implementation whenever possible. We also have tested our prototype thoroughly on several case studies.

The set of use cases used for evaluation could be an other threat to validity. In fact, some of the use cases 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.

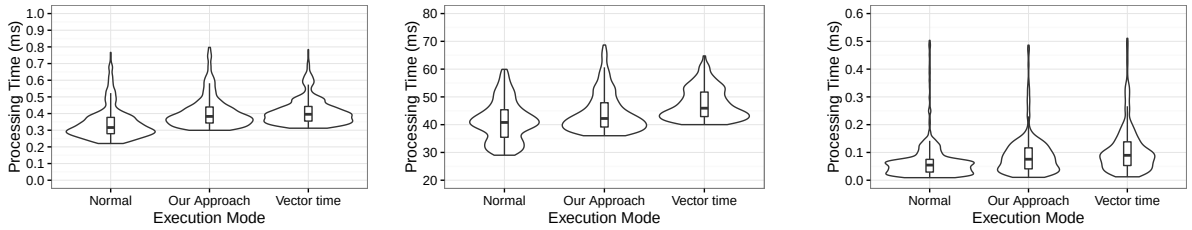
Finally, part of our evaluation is a comparison with the vector time approach. Since no implementation of vector time in the context of UML-RT state machines was available, we implemented our own. While we did our best to implement the vector time approach optimally, there might be room for further optimization.

5 RELATED WORK

The following four groups of existing work seem most relevant.

Trace reordering and debugging for distributed systems In techniques based on 'record and replay', nodes use previously recorded information to resolve non-determinism and replay executions exactly [19]. This, however, is insufficient for debugging where a user wants to, e.g., stop the replay, change variable values and then examine the new execution that follows [11]. Centralized applications such as debuggers typically use some form of timestamp to reorder traces [11, 29]. Physical and logical timestamps have been widely researched and successfully used [21, 30, 77]. E.g., many different kinds of logical timestamps have been studied [1, 54, 59, 71], including extensions to message- and event-based programs [61, 67, 70]. For event-based applications as found, e.g., in Android, Maiya et al [61] use an 'event graph' to capture communication dependencies. However, while suitable in many situations, physical and logical timestamps have disadvantages: The former require, possibly highly synchronized, clocks [27, 65, 86]; the latter 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 [33, 53, 75, 76, 82]. Our work sidesteps these challenges and leverages static analysis and replay of state machines to avoid the use of timestamps.

MDE for distributed systems The use of models to facilitate development, deployment, and analysis of distributed systems has



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

Figure 5: Performance overhead of our approach and vector time approach

been proposed before (e.g., [9, 20, 35, 36, 48, 51]). However, none of this work is concerned with trace reordering.

Tracing and replay in the context of MDE Hojaji et al [41] 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., [22, 23, 34, 37, 40, 44, 45, 66, 80]). E.g., Iyengar et al [44, 45] 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. [23] propose a configurable tracing tool based on LTTng [25]. Code instrumentation is used to produce LTTng tracepoints. Trace replay allows timing analysis and can be performed offline or live, using a remote connection to the target platform. Bousse et al [16, 17] propose an approach for supporting omniscient debugging for executable DSLs through efficient trace management. Again, none of the work in this category is concerned with reordering distributed system traces.

Trace reordering and debugging for actor languages UML-RT's execution semantics follows the actor model [3, 39], which underlies many recent highly responsive, fault-tolerant distributed applications [14, 24]. Approaches for record and replay for actor languages have been proposed by Tveito et al [83] and Aumayer et al [4]. Both approaches do not require timestamps. Similar to our work, the work in [4] 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 [55, 56] and Shibanai et at [49] does focus on debugging. In both approaches, traces are sent to a centralized component, which is similar to the work by Maiya et al on event-based programs [62] and our work. However, both also rely on timestamps. Also, their suitability for resource-constrained systems is unclear: [55] relies on a somewhat idealized language (without, e.g., mutable objects); the experimental results in [56] are inconclusive and the number and size of traces is not measured; and [49] requires every node to run a JVM.

In sum, we did not find an approach that has the same scope, i.e., that leverages existing modeling tools and techniques (Requirement R1), is suitable for systems with resource-constrained nodes (R2), and can be integrated into a centralized debugger (R3).

6 CONCLUSION

The benefits of modeling for design-level activities are well publicized. For implementation-level activities such as debugging less evidence is available (with the exception of some of the work on 'models@runtime' [12]). In this paper, we provide evidence that modeling can facilitate the ordering of traces of a distributed system, and thus debugging and replay. Concretely, we have shown 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. Our work takes advantage of existing techniques and tools for model transformation and static analysis.

The approach relies on access to the descriptions of the behaviours of each component for static analysis and replay. However, the language that these descriptions are given in also needs to satisfy the following conditions: (1) message passing and strong encapsulation: components do not share state and communicate only via messages, (2) run-to-completion: an execution triggered through the arrival of a message cannot be interrupted by the arrival of another message, and (3) deterministic message handling: the execution triggered by a message can be uniquely determined during replay.

Apart from state machines in UML-RT and UML (under certain restrictions such as the absence of orthogonal regions), several other executable modeling or programming languages may satisfy the three conditions above and thus also be amenable to our approach. Promising candidates include languages (e.g., Erlang and Orleans) and language extensions (e.g., Akka for Java and Scala, Comedy for Node.js, CAF for C++, Pykka for Python, and Actix for Rust) based on the actor model, as well as languages based on a 'communicating event loop' (e.g., E and JavaScript) [4, 14, 24, 64]. Given that some of these languages are quite popular and increasingly used to make web-based or embedded distributed applications more scalable and robust (e.g., WhatsApp is built on Erlang, Microsoft's Halo uses Orleans, and Akka powers LinkedIn), our future work

will explore the applicability and utility of our approach on some of these candidates. Moreover, the fact that the generated traces originate from state machines suggests that further optimization of the way the traces are collected and stored should be possible. Future work will also investigate this question. Finally, we will continue our work on leveraging the approach to realize sophisticated debugging capabilities.

REFERENCES

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D. Ernst. 2014. Shedding Light on Distributed System Executions. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 598–599. https://doi.org/10.1145/2591062.2591134

[2] Atul Adya and Barbara Liskov. 1997. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/259380.259425

[3] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press.

[4] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. Efficient and Deterministic Record & Replay for Actor Languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article Article 15, 14 pages. https://doi.org/10.1145/3237009.3237015

[5] Mojtaba Bagherzadeh. 2019. *Model-level debugging in the context of the model driven development*. PhD dissertation. Kingston, Ontario, Canada.

[6] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2017. 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 (ESEC/FSE 2017)*. ACM, New York, NY, USA, 419–430. https://doi.org/10.1145/3106237.3106278

[7] Mojtaba Bagherzadeh, Nicolas Hili, David Seekatz, and Juergen Dingel. 2018. MDebugger: A Model-Level Debugger for UML-RT. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 97–100. https://doi.org/10.1145/3183440.3183473

[8] Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill, and Douglas C Schmidt. 2009. Adaptive Failover for Real-Time Middleware with Passive Replication. In *15th IEEE Symposium on Real-Time and Embedded Technology and Applications*. IEEE, 118–127.

[9] Yogesh D. Barve, Prithviraj Patil, Anirban Bhattacharjee, and Aniruddha S. Gokhale. 2018. PADS: Design and Implementation of a Cloud-Based, Immersive Learning Environment for Distributed Systems Algorithms. *IEEE Trans. Emerg. Top. Comput.* 6, 1 (2018), 20–31. https://doi.org/10.1109/TETC.2017.2731984

[10] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. 2011. Karma: Scalable Deterministic Record-Replay. In *In International Conference on Supercomputing*.

[11] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging Distributed Systems. *Commun. ACM* 59, 8 (July 2016), 32–37.

[12] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@Run.Time. *Computer* 42, 10 (Oct. 2009), 22–27.

[13] Blinded for review. 2020. MReplayer. repository address blinded for review. Retrieved: 2020-5-20.

[14] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5 (Oct. 2017), 39.

[15] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto. https://www.reactivemanifesto.org/. Last accessed: May 18, 2020.

[16] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (2018), 261 – 288. https://doi.org/10.1016/j.jss.2017.11.025

[17] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. 2019. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software and Systems Modeling* (Feb. 2019), 1–37. https://doi.org/10.1007/s10270-017-0598-5

[18] Yunji Chen, Tianshi Chen, and Weiwu Hu. 2009. Global Clock, Physical Time Order and Pending Period Analysis in Multiprocessor Systems. *CoRR* abs/0903.4961 (2009). arXiv:0903.4961 http://arxiv.org/abs/0903.4961

[19] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Comput. Surv.* 48, 2, Article 17 (Sept. 2015), 47 pages. https://doi.org/10.1145/2790077

[20] Adam Childs, Jesse Greenwald, Venkatesh Prasad Ranganath, Xianghua Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Prashant Shanti, and Gurdip Singh. 2004. Cadena: An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-Based Systems. In *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004 Barcelona, Spain, March 29 - april 2, 2004, Proceedings (Lecture Notes in Computer Science)*, Vol. 2984. Springer, 160–164. https://doi.org/10.1007/978-3-540-24721-0_11

[21] Mark Christiaens and Koen De Bosschere. 2001. 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 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267847.1267862

[22] Jonathan Corley, Brian P Eddy, and Jeff Gray. 2014. Towards efficient and scalable omniscient debugging for model transformations. In *Proceedings of the 14th workshop on domain-specific modeling*. ACM, 13–18.

[23] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2016. 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, 36–43.

[24] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE'16)*. ACM, 31–40.

[25] Mathieu Desnoyers and Michel R Dagenais. 2006. The LTTNG tracer: A low impact performance and behavior monitor for GNU/Linux. In *Ottawa Linux Symposium (OLS)*, Vol. 2006. Citeseer, Linux Symposium, 209–224.

[26] Eclipse Foundation. 2020. Eclipse eTrice Real-Time Modeling Tools. https://www.eclipse.org/etrice.

[27] John C. Eidson. 2008. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (July 2008), 1–300. https://doi.org/10.1109/IEEESTD.2008.4579760

[28] Shahram Esmaeilsabzali, Nancy Day, Jo Atlee, and Jianwei Niu. 2010. Deconstructing the semantics of big-step modelling languages. *Requir. Eng.* 15 (06 2010), 235–265. https://doi.org/10.1007/s00766-010-0102-z

[29] C. Fidge. 1996. Fundamentals of distributed system observation. *IEEE Software* 13, 6 (Nov 1996), 77–83. https://doi.org/10.1109/52.542297

[30] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[31] Eclipse Foundation. 2016. Eclipse Papyrus for Real Time (Papyrus-RT). https://www.eclipse.org/papyrus-rt. Retrieved May 15, 2020.

[32] Eclipse Foundation. 2020. Eclipse Modeling Framework (EMF). https://eclipse.org/modeling/emf.

[33] Vijay K. Garg and Chakarat Skawratananond. 2001. String Realizers of Posets with Applications to Distributed Computing. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC '01)*. ACM, New York, NY, USA, 72–80. https://doi.org/10.1145/383962.383988

[34] Eran Gery, David Harel, and Eldad Palachi. 2002. Rhapsody: A Complete Life-Cycle Model-Based Development System. In *International Conference on Integrated Formal Methods*. Springer, 1–10.

[35] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. 2008. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming* 73, 1 (2008), 39 – 58. https://doi.org/10.1016/j.scico.2008.05.005

[36] Aniruddha Gokhale, Krishnakumar Balasubramanian, and Tao Lu. 2004. CoSMIC: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. ACM, 218–219. https://doi.org/10.1145/1028664.1028758

[37] Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. 2010. Model-Level Debugging of Embedded Real-Time Systems. In *10th Int. Conference on Computer and Information Technology (CIT'10)*. IEEE, 1887–1894.

[38] HCL. 2020. HCL RealTime Software Tooling (RTist). https://www.hcltech.com/software/rtist. Retrieved April 30, 2020.

[39] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Stanford, CA, USA, August 20-23, 1973*. William Kaufmann, 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

[40] Nicolas Hili, Mojtaba Bagherzadeh, Karim Jahed, and Juergen Dingel. 2020. A model-based architecture for interactive run-time monitoring. *Software and Systems Modeling* (2020), 1–23.

[41] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. 2019. Model execution tracing: a systematic mapping study.

Software & Systems Modeling (18 Feb 2019).
[42] Susan Horwitz, Thomas Reps, and Dave Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990).
[43] IBM. 2016. IBM RSARTE. https://www.ibm.com/developerworks/community/wikis. Retrieved May 15, 2020.
[44] Padma Iyenghar, Elke Pulvermuelle, Clemens Westerkamp, Michael Uelschen, and Juergen Wuebbelmann. 2011. Model-Based Debugging of Embedded Software Systems. *Gesellschaft Informatik (GI)-Softwaretechnik (SWT)* (2011).
[45] Padma Iyenghar, Clemens Westerkamp, Juergen Wuebbelmann, and Elke Pulvermueller. 2010. A Model Based Approach for Debugging Embedded Systems in Real-Time. In *10th ACM Int. Conference on Embedded Software*. 69–78.
[46] Nafiseh Kahani and James R Cordy. 2015. *Comparison and evaluation of model transformation tools*. Technical Report. Technical Report 2015-627.
[47] Nafiseh Kahani, Nicolas Hili, James R Cordy, and Juergen Dingel. 2017. 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, 12–18.
[48] Karim Jahed. 2019. Papyrus-RT Distribution. https://github.com/kjahed/papyrusrt-distribution. Retrieved: 2020-5-20.
[49] Kazuhiro Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*. ACM, 13–22. https://doi.org/10.1145/3281366.3281370
[50] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The Epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*. Springer, 46–60.
[51] Oleksandra Kulankhina. 2016. *A framework for rigorous development of distributed components: formalisation and tools*. PhD theses. Université Côte d'Azur. https://tel.archives-ouvertes.fr/tel-01419298
[52] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer International Publishing, Cham, 17–32.
[53] Sandeep S. Kulkarni and Nitin H. Vaidya. 2017. Effectiveness of Delaying Timestamp Computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/3087801.3087818
[54] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563
[55] Ivan Lanese, Adrián Palacios, and Germán Vidal. 2019. Causal-Consistent Replay Debugging for Message Passing Programs. In *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings (Lecture Notes in Computer Science)*, Vol. 11535. Springer, 167–184. https://doi.org/10.1007/978-3-030-21759-4_10
[56] Palacios A. Vidal G. Lanese, I. 2019. *Causal-consistent replay debugging for message passing programs*. Technical Report. Technical report, DSIC, Universitat Politècnica de València. http://personales.upv.es/gvidal/german/forte19tr/paper.pdf
[57] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 454–467. https://doi.org/10.1145/2934872.2934885
[58] J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. V. Kalé. 2014. Scalable replay with partial-order dependencies for message-logging fault tolerance. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 19–28. https://doi.org/10.1109/CLUSTER.2014.6968739
[59] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2020. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *Commun. ACM* 63, 3 (Feb. 2020), 94–102. https://doi.org/10.1145/3378933
[60] Jeff Magee and Jeff Kramer. 1999. *State Models and Java Programs*. Wiley.
[61] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. 2016. Partial Order Reduction for Event-Driven Multi-threaded Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 680–697.
[62] Pallavi Maiya and Aditya Kanade. 2017. Efficient Computation of Happens-before Relation for Event-driven Programs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 102–112. https://doi.org/10.1145/3092703.3092733
[63] Caitie McCaffrey. 2015. Building the Halo 4 Services with Orleans. Presentation at QCon London.
[64] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers: Progamming in E as Plan Coordination. In *Trustworthy Global Computing*, Rocco De Nicola and Davide Sangiorgi (Eds.). 195–229.
[65] D. L. Mills. 1991. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications* 39, 10 (Oct 1991), 1482–1493. https://doi.org/10.1109/26.103043
[66] Sadaf Mustafiz and Hans Vangheluwe. 2013. Explicit modelling of statechart simulation environments.. In *Summer Simulation Multiconference*. Society for Computer Simulation International (SCS), 445 – 452.
[67] R.H.B. Netzer and B.P. Miller. 1995. Optimal tracing and replay for debugging message-passing parallel programs. *Journal of Supercomputing* 8 (1995), 371–388. https://doi.org/10.1007/BF01901615
[68] Ernesto Posse and Juergen Dingel. 2016. An Executable Formal Semantics for UML-RT. *Software and Systems Modeling* 15, 1 (2016), 179–217.
[69] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. 1990. Fault-Tolerant Clock Synchronization in Distributed Systems. *Computer* 23, 10 (Oct. 1990), 33–42. https://doi.org/10.1109/2.58235
[70] M. Ronsse and D Kranzlmuller. 1998. Rolt^{mp}-replay of Lamport timestamps for message passing systems. In *Proc. 6th Euromicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE.
[71] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing* 7, 3 (01 Mar 1994), 149–174. https://doi.org/10.1007/BF02277859
[72] Bran Selic. 1998. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*. Springer, 250–260.
[73] Bran Selic, Garth Gullekson, and Paul T Ward. 1994. *Real-Time Object-Oriented Modeling*. Vol. 2. John Wiley and Sons New York.
[74] Bran Selic and Objectime Limited. 1998. Using UML for modeling complex real-time systems.
[75] M. Shen, A. Kshemkalyani, and A. Khokhar. 2013. Detecting Unstable Conjunctive Locality-Aware Predicates in Large-Scale Systems. In *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*. 127–134. https://doi.org/10.1109/ISPDC.2013.25
[76] Mukesh Singhal and Ajay Kshemkalyani. 1992. An Efficient Implementation of Vector Clocks. *Inf. Process. Lett.* 43, 1 (Aug. 1992), 47–52. https://doi.org/10.1016/0020-0190(92)90028-T
[77] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. *SIGPLAN Not.* 47, 1 (Jan. 2012), 387–400. https://doi.org/10.1145/2103621.2103702
[78] Scott D. Stoller. 2000. Detecting global predicates in distributed systems with clocks. *Distributed Computing* 13, 2 (01 Apr 2000), 85–98. https://doi.org/10.1007/s004460050069
[79] William Swartout and Robert Balzer. 1982. On the Inevitable Intertwining of Specification and Implementation. *Commun. ACM* 25, 7 (1982), 438–440.
[80] Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. 2003. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of the Int. Parallel and Distributed Processing Symposium (IPDPS'03)*. IEEE, 1–8.
[81] The Object Management Group. 2017. UML Superstructure Specification. Version 2.5.1. http://www.omg.org/spec/UML/2.5.1/PDF. Retrieved May 15, 2020.
[82] Francisco J. Torres-Rojas and Mustaque Ahamad. 1999. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing* 12, 4 (01 Sep 1999), 179–195. https://doi.org/10.1007/s004460050065
[83] Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. 2020. Global Reproducibility Through Local Control for Distributed Active Objects. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Vol. 12076. Springer, 140–160. https://doi.org/10.1007/978-3-030-45234-6_7
[84] UML2.5.1. 2017. UML2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF. Retrieved June 5, 2019.
[85] Michael von der Beeck. 2006. A Formal Semantics of UML-RT. In *Model Driven Engineering Languages and Systems*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 768–782.
[86] Yuki Yamada and Tatsuo Nakajima. 2018. *Experiments of distributed ledger technologies based on global clock mechanisms*. Springer-Verlag, 436–445. https://doi.org/10.1007/978-3-319-99626-4_38