Efficient Replay-based Regression Testing for Distributed Reactive Systems in the Context of Model-driven Development

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

Juergen Dingel
Queen's University, Canada
dingel@cs.queensu.ca

***Abstract*—As software evolves, regression testing techniques are typically used to ensure the new changes are not adversely affecting the existing features. Despite recent advances, regression testing for distributed systems remains challenging and extremely costly. Existing techniques often require running a failing system several time before detecting a regression. As a result, conventional approaches that use re-execution without considering the inherent non-determinism of distributed systems, and providing no (or low) control over execution are inadequate in many ways. In this paper, we present *MRegTest*, a replay-based regression testing framework in the context of model-driven development to facilitate deterministic replay of traces for detecting regressions while offering sufficient control for the purpose of testing over the execution of the changed system. The experimental results show that compared to the traditional approaches that annotate traces with timestamps and variable values *MRegTest* detects almost all regressions while reducing the size of the trace significantly and incurring similar runtime overhead.**

***Index Terms*—MDD, Distributed Systems, Regression Testing**

I. Introduction

Regression testing is defined as a type of software testing to confirm recent changes have not adversely affected existing features. But it can be extremely costly [1] and ineffective [2] especially for testing resource-constrained distributed systems (e.g., IoT) with possibly many nodes. For example, large software companies such as Google have over 100 million tests running each day on massive clusters of powerful machines which may produce large log files of collected traces and consume a lot of memory [1].

Testing by replay is an emerging technology that enables developers to execute recorded traces of a system in a repeated and deterministic manner and make diagnostic observations, e.g., [3], [4], [5], [6], [7], [8]. Despite recent progress in *replay-based regression testing* (RRT) techniques and tools, e.g., [9], [10], [11], it turns out they are inadequate for testing distributed systems [12]. As such, a typical distributed system with a few nodes can produce a large amount of (possibly out of order) traces only after a few minutes of execution [13]. Therefore, an effective RRT for distributed systems may require efficient mechanisms for replaying and regression detection [14].

In this paper, we present *MRegTest*, a replay-based regression testing tool for distributed systems at the model-level. The presented work is part of a larger research agenda that aims to identify and leverage model-driven software development concepts and techniques to facilitate the development of distributed systems. An approach to reordering traces without the use of timestamps constituted the first step in this agenda [15]. Our work on regression testing builds on our reordering approach and inherits from it a reduction in the number and size of traces required compared to standard, timestamp-based approaches. Conceptually, the work benefits from the increased level of abstraction together with strategic semantic simplifications (such as the 'run-to-completion' assumptions for state machines) that a model-based system description can offer. Concretely, we discuss how the use of communicating state machines to describe a distributed system can be leveraged for a significant reduction in the number of base model[1] executions that RRT needs to replay to detect a regression. Our approach consists of the following three steps: (1) Critical Variable Identification (*CVI*) that allows the user to adjust the level of granularity of the regression tests as appropriate; (2) Execution Selection (*ES*) that reduces the number of base model executions[2] that need to be replayed; (3) Regression Detection (*RD*) that uses an extension of MReplayer, our model-based trace replayer from [15], to determine: (*a*) whether the replayer can replay traces collected from the base model executions, (*b*) whether the value of critical variables are consistent between the base and modified version of a model. Apart from developing an effective regression testing approach for model-based distributed systems our work was guided also by the following requirements:

R1: The approach leverages existing modeling techniques.

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

There are three main architectures of regression testing for distributed systems: (1) Local testers on each nodes e.g., [16], [17], [18], (2) Single centralized tester, e.g., [19], [20], [21] or (3) Hybrid approach of local and centralized testers, e.g., [22]. The pros and cons of each architecture have been discussed in [23]. We base our work on the *centralized test architecture*, since unlike local and hybrid testers that typically introduce an additional network overhead for exchanging messages, this

[1]Referring to an original model whose state machine has not been changed
[2]Traces collected from executions of an instrumented base model

architecture maintains a centralized testing agent for possibly several nodes which is more suitable for testing resource-constrained IoT systems [24], [25].

Our approach achieves **R1** by leveraging abstraction and automation with Model-Driven Development (MDD). We use an open-source MDD tool [26] that allows us to generate code from models, such that it can be executed on different nodes in a distributed system. Also, in our approach the behaviour of components is described using communicating state machines.

To achieve **R2**, we extend the MReplayer tool described in [15], [27]. MReplayer provides an efficient deterministic replay of traces collected from a distributed system without the use of timestamps. Also, variable values are re-generated during a replay which means that they don't have to be included in the traces. Together, both characteristics allow for a significant reduction in the size and number of traces. In fact, only those variables values are added to a trace that cannot be regenerated during a replay (e.g., values depended on random number generators and local resources such as files). Our approach also takes advantage of run-to-completion (RTC), an often made atomicity assumption that greatly simplifies dealing with concurrency in distributed systems. 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 [28], but also underlies many languages (or language extensions) built on the actor model such as Akka [29] or Orleans [30]. Many of these languages have successfully been used to implement industrial reactive systems [31], [32].

To assess *MRegTest*, we conducted an empirical study based on the evaluation framework for regression testing techniques proposed by Rothermel et al. [33] in which we applied our approach to detect regressions in modified models with various levels of complexity. Our experiments show performing each step of our approach, i.e., Execution Selection (*ES*) and Regression Detection (*RD*) even on large models (i.e., models with more then 2000 states and 3000 transitions) takes less than a minute. Also, using our approach the size of traces collected from each use case is reduced significantly by an average factor of $1.56\times$ compared to traditional approaches that annotate trace with timestamps and variable values, e.g. [34], [7]. Finally, we evaluated runtime overhead of our approach against models with different complexities. The results show that the extra runtime overhead imposed by re-generating variable values by reply in our approach is within a reasonable range. We make the following contributions:

- We present a replay-based regression testing approach tailored to distributed systems designed at the model level.
- We introduce two algorithms for execution selection and regression detection that allow efficient regression testing in terms of the size of traces collected from an original system.
- We provide an open source implementation of our approach.

The rest of this paper is organized as follows. Section II presents some necessary background. Section III describes our approach. The evaluation, experimental results, and threats to validity are summarized in Section IV. Related work is reviewed in Section V.

II. Background

To illustrate and implement our approach, we use UML for Real-time (UML-RT) [35], [36], a language specifically designed for real-time embedded systems with soft real-time constraints. Over the past two decades, it has been used successfully in industry to develop several large-scale industrial projects, via, e.g., IBM RSA-RTE [37], HCL RTist [38], Protos eTrice [39] and Papyrus-RT [40]. Recently, an extension of Papyrus-RT has been developed that allows the development of distributed systems with UML-RT [26]. Our work builds on this extension. Below, the most relevant aspects of UML-RT will be described.

A. 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) [35], [41].

B. MDD for Distributed Systems: Example

Fig. 1 shows the design of a simplified ATM model with PinPad server (i.e., *PPD*), a Controller server (*CTR*) and a Bank server (*BNK*), as shown in the structure diagram in the top-left. The state machines, specifying the behaviour of the instances of a component, for *PPD*, *CTR*, and *BNK* are shown in the bottom-left, centre-right, and centre-top of Fig. 1, respectively.

In order to make either a withdrawal or a deposit request, first the component *PPD* sends the internal message *init* that initiates the operation. In state s_1 it then presents the user with options. Once a user selects one of the available options its corresponding boolean variable (i.e., `optD`, `optW`) turns to true. The component *PPD* interacts with *CTR* in order to transmit the request to *BNK*. Thus, it takes the transitions t_2, t_3 that send the message *usrReq* to the component *CTR*. Upon the reception of this message, *CTR* checks its connection to *BNK* where it responds by sending the message *ack* back to the *CTR*. This can carry either the value *accept* (i.e., transitions t_2, t_3) or *reject* (i.e., transition t_4) depending on whether the connection was established successfully or not. Once the value *accept* or *reject* is received by *CTR*, it sends the message *ok* or *nok* back to *PPD*, respectively.

Once the message *ok* has received, *PPD* sends the request to *CTR* via the message *req (WDW, amt)* where *WDW* and

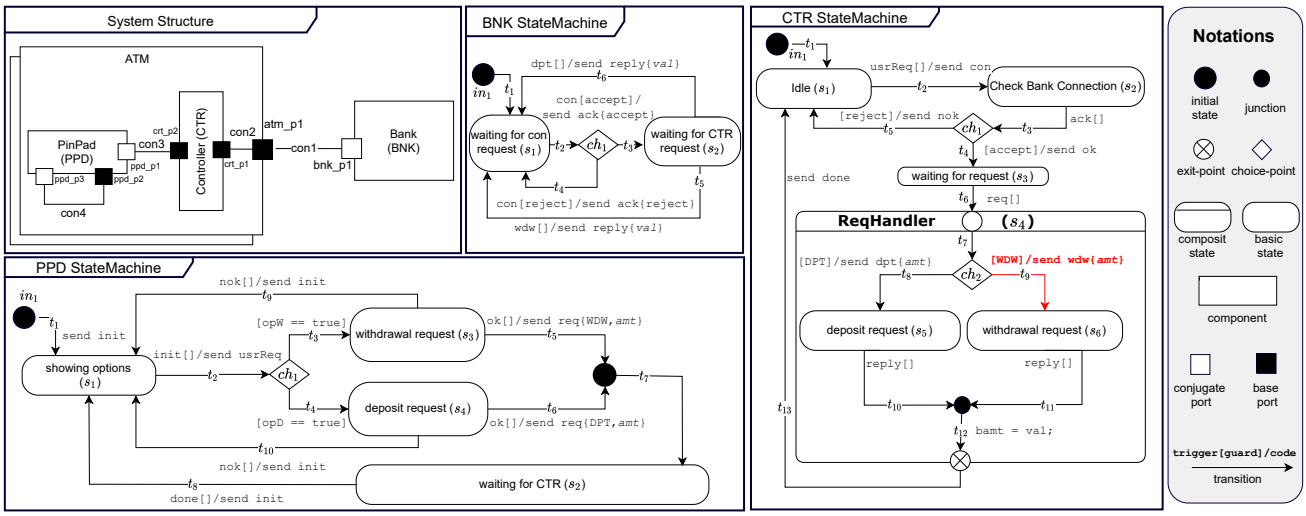

Figure 1: Models of Structure and Behaviour of Simplified Automated Teller Machine (ATM)

the variable *amt* indicate the request's type and the amount of money that the user wishes to withdraw from his bank account in *BNK*, respectively. Upon the reception of this message, the transitions t_6, t_7, t_9 are taken in *CTR* where it supplements this request with some additional information and sends the message *wdw* and the variable *amt* to *BNK*. In *BNK* the transition t_5 is taken and the value of *amt* is deducted from the user's balance amount and the result (i.e., *val*) is sent back to *CTR* via the message *reply* . Once this message is received by *CTR*, it takes transitions t_{11}, t_{12}, t_{13} where it assigns the new balance amount to the variable *bamt* and sends the message *done* to *PPD*. Then, *PPD* takes the transition t_8 and backs to the sate s_1 where it sends the internal message *init* and shows the available options to the user again.

Suppose the ATM system 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. Since nondeterminism is pervasive in distributed systems, traces might arrive at a centralized regression testing system in an arbitrary order. E.g., the arrival of a trace corresponding to the transition t_2 in *CTR* at a tester might precede one corresponding to t_2 in *PPD*. Therefore, the use of an efficient record and replay mechanism is essential. To satisfy this requirement, we employ the trace reordering method introduced by MReplayer [15] that offers a low overhead deterministic replay of traces. For example, E_1, E_2, and E_3 shown in Fig. 2 are three sample executions of the ATM system received by a centralized tester. Each execution consists of a set of traces from components of the ATM system: *BNK* (i.e., B) , *CTR* (i.e., C), and *PPD* (i.e., P). For example, t_1^P denotes the trace t_1 in the component *PPD*.

Let's consider a modification scenario in the component *CTR*. Suppose in this component the variable balance amount (i.e., *bamt*) is a critical variable and a developer modifies action code of t_9^C such that $amt+10$ is sent as argument in the wdw message to *BNK* instead of amt. This slight change will affect the expected behaviour of the system. For example, once a user with $100 initial balance amount (i.e., *bamt* = $100) wants to withdraw $10 from its account, the component *CTR* makes a withdrawal request in the transition t_5^P, and sends the value of *amt* along with the message wdw to *BNK*. According to this scenario, *BNK* is expected to send $90 as the result of this operation back to *CTR*, but because of this modification it sends $80 that is directly assigned to the variable *bamt*. We consider this modification a regression, since the value of this critical variable differs from the one produced by the base model.

C. Leveraging MReplayer

We leveraged MReplayer [15] to facilitate reordering traces collected from a distributed system. MReplayer is based on the replay of execution traces 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, MReplayer uses the replay of the state machine execution to determine consistent orderings of input traces.

When MReplayer starts it creates abstract capsules which are simplified counterparts for each of the capsules in the system. Then, they are connected to an abstract controller that acts as a communication gateway for delivering messages between abstract capsules. Unlike their real capsule counterparts, abstract capsules do not have a state machine or ports. Instead, each abstract capsule is provided with the set of rc-steps that its state machine can perform. Also, the abstract capsules keep the current execution state, as well as use a first-in-first-out queue for receiving traces, and a list that contains all incoming messages generated during a replay.

MReplayer reorders (possibly out-of-order) traces emitted from the instrumented system. They are received by the abstract 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 a replay, each abstract capsule uses its queue of incoming

messages and rc-steps to determine what transition can be triggered from the current state.

$$
\begin{array}{lllllllllll}
E_1 = & t_1^P & t_1^B & t_1^C & t_2^P & t_2^C & t_2^B & t_3^C & t_9^P & & & \\
E_2 = & t_1^P & t_1^B & t_1^C & t_2^P & t_2^C & t_2^B & t_3^C & t_5^P & t_6^C & t_5^B & t_{11}^C \\
E_3 = & t_1^P & t_1^B & t_1^C & t_2^P & t_2^C & t_2^B & t_3^C & t_6^P & t_6^C & t_6^B & t_{10}^C
\end{array}
$$

Figure 2: Sample Executions of the ATM system

Run-to-completion (RTC): It is a central concept in the definition of the execution semantics of many state machine languages including UML and UML-RT [42], [43]. It prevents the processing of an incoming message during the processing of an earlier message. RTC allows a sequence of *micro steps* to be viewed as a single execution step without loss of information. Based on the notion of RTC, an rc-step can formally be defined as a tuple $\langle \sigma, C, P, \sigma' \rangle$, where σ refers to the source state of rc, σ' refers to the destination state of rc, P is a finite sequence of transitions, and C is a set of messages representing the trigger of the first transition in P. An rc-step allows us to group all execution steps into a single RTC (i.e., macro) step. Some of the rc-steps of the state machines in the ATM system are shown in TableI. Also, rc_{E_2} contains the sequence of rc-steps corresponding to traces of E_2 in Fig. 2.

$$rc_{E_2} = rc_1^P\ rc_1^B\ rc_1^C\ rc_2^P\ rc_2^C\ rc_2^B\ rc_3^C\ rc_3^P\ rc_5^C\ rc_5^B\ rc_6^C$$

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 holds using the value map of γ, and m triggers the step (i.e., $m \in rc.C$). For example, we can execute rc_3^C only if the active state is s_2, the message *ack* exists at the head of the message queue of C, and *accept* is evaluated *true* in the value map of the configuration γ.

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, rc_1^C which contains the initial transition of capsule *CTR* be enabled when its abstract capsule is started.

III. Description of Approach

A. Overview

A graphical overview of the implementation of our approach is shown in Fig. 3. Our approach assumes that the traced system has been instrumented using MReplayer's instrumentation approach [15] and generates execution traces at runtime.

Our approach detects regressions by replaying execution traces collected from a base model on its modified version. To achieve this goal, first critical variables for each capsule are identified by the user. Critical variables determine what constitutes a regression: in the changed model, the values of critical variables must be the same as in the base model, while the values of non-critical values can differ. This allows the user to focus the analysis on the most relevant parts and abstract from changes to irrelevant parts. The selection of critical variables can thus to used as an 'information hiding' mechanism, akin to, e.g., the 'hiding' operator in classical

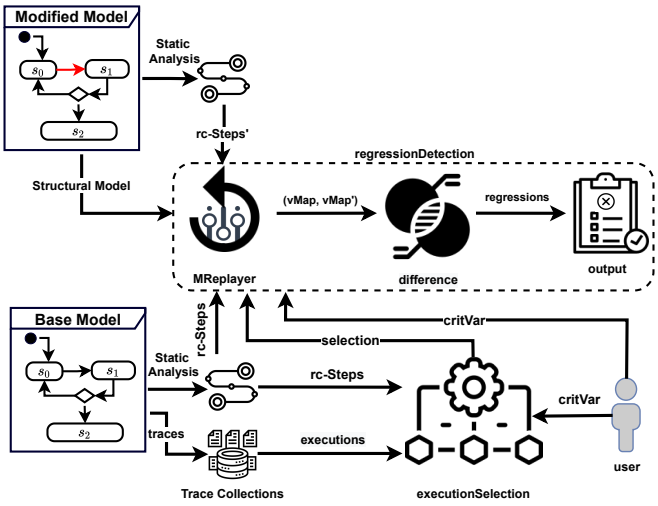

Figure 3: An overview of our approach

process algebras such as CCS and CSP [44], [45]. We suppose a set of critical variables are provided by preferably a domain expert user as an input to the functions *executionSelection* and *regressionDetection*. In the context of our running example, the variable balance amount, i.e., *bamt* is considered as the only critical variable in the capsule *CTR*. The function *executionSelection* selects only those executions of a base model that may modify any variables in a given set of critical variables. The output of this function is fed into the function *regressionDetection*. The function replays the trace on both the base and the modified model to detect steps during which the modified model (1) makes a critical variable take on a value that is different from that in the base model, or (2) is unable to complete the replay of the trace.

Our work assumes that state machine models satisfy well-formedness conditions which typically are considered in MDD. In particular, we assume the following two conditions: (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'. These conditions ensure that 1) a message arriving at a capsule enables at most one rc-step, and 2) there is a 1-to-1 correspondence between a trace observed in the execution and the rc-step it corresponds to.

B. Supported Modifications

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

Leverage the Definition of rc-steps. As discussed in MReplayer, the set of rc-steps of a capsule, extracted by performing static analysis, is fully capable of describing its possible behaviours. Therefore, we can leverage the definition of rc-steps (in Section II) to identify any regressions from the expected behaviour of a base model. In fact, any modifications of the model elements defining an rc-step (i.e., σ, C, P, and σ') can affect the behaviour of its base model. In the context

CTR	***BNK***	***PPD***
$rc_1^C = \langle in_1, \textit{startUp}, \langle t_1 \rangle, s_1 \rangle$	$rc_1^B = \langle in_1, \textit{startUp}, \langle t_1 \rangle, s_1 \rangle$	$rc_1^P = \langle in_1, \textit{startUp}, \langle t_1 \rangle, s_1 \rangle$
$rc_2^C = \langle s_1, \textit{usrReq}, \langle t_2 \rangle, s_2 \rangle$	$rc_2^B = \langle s_1, \textit{con}, \langle t_2, t_3 \rangle, s_2 \rangle$	$rc_2^P = \langle s_1, \textit{init}, \langle t_2, t_3 \rangle, s_3 \rangle$
$rc_3^C = \langle s_2, \textit{ack}, \langle t_3, t_4 \rangle, s_3 \rangle$	$rc_3^B = \langle s_2, \textit{con}, \langle t_2, t_4 \rangle, s_1 \rangle$	$rc_3^P = \langle s_3, \textit{ok}, \langle t_5, t_7 \rangle, s_2 \rangle$
$rc_4^C = \langle s_3, \textit{req}, \langle t_6, t_7, t_8 \rangle, s_5 \rangle$	$rc_4^B = \langle s_2, \textit{dpt}, \langle t_6 \rangle, s_1 \rangle$	$rc_4^P = \langle s_2, \textit{done}, \langle t_8 \rangle, s_1 \rangle$
$rc_5^C = \langle s_3, \textit{req}, \langle t_6, t_7, t_9 \rangle, s_6 \rangle$	$rc_5^B = \langle s_2, \textit{wdw}, \langle t_5 \rangle, s_1 \rangle$	
$rc_6^C = \langle s_6, \textit{reply}, \langle t_{11}, t_{12}, t_{13} \rangle, s_1 \rangle$		

Table I: Some of the rc-steps of *CRT*, *BNK*, and *PPD*

of the running example, the change in action code of the transition t_9 in the capsule *CTR*, reflects on its corresponding rc-steps (i.e., rc_5^C). When this rc-step is replayed, a critical variable may take on a different value, thus our approach can determine whether this modification may cause a regression or not.

What Modifications Are Supported. The range of possible modifications that a developer may want to apply to a model can be extensive and depends on the size and complexity of the model. However, the majority of them can be put them into two groups: (1) Modifications of the behavioural model including action code, messages and state machines (2) Modifications of the structural model including ports, protocols, and connectors. In this study, we merely consider the former group, and in our future work we will explore modifications of the structural model.

MReplayer creates a parser for *Action Code* language which is a subset of *C++* language using ANTLR [46]. This parser supports most commonly used expressions and is equipped to parse primitive operations such as accessing and updating variables, arithmetic and logical expressions, and control flow statements (e.g., if, else if). Nonetheless, the parser does not support expressions in action code that use the following C++ features: (1) object-oriented programming (OOP) principles (e.g., inheritance, encapsulation, and polymorphism); (2) user-defined data types (e.g., template, class, structure, union, and enumeration); (3) derived data types (e.g., function, array, pointer, and reference); (4) periodic timer functions (e.g., inform in, inform every, and cancel). Essentially, MReplayer does not re-generate values from such expressions by replay. Thus, any regressions due to a modification in action code of such expressions are not detected by our regression testing approach. For example, suppose a critical variable whose value is modified within a function in action code of a transition. Since replay of a function is not supported in our approach, possible regressions associated with such modification will remain undetected.

C. Critical Variable Identification (CVI)

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

Algorithm 1: *executionSelection*(*rcSteps: set of rc-steps of C*, *critVar: set of critical variables*, *executions: set of executions of C*)

```
1  Let selection be an emptyset
2  forall execution ∈ executions do
3      γ ← γ0
4      inTraces ← execution
5      msgs ← append(msgs, startUp)
6      while (inTraces not consumed) do
7          msg ← dequeue(msgs)
8          rcStep,γ1 ← getRCStep(msg, rcSteps,γ)
9          removeMatchingTrace(inTraces, rcSteps)
10         γ ← replay(rcStep, γ1)
11         actSeq ← actions(rcStep)
12         var ← variables(actSeq)
13         if {critVar ∩ var} ≠ ∅ then
14             selection ← selection ∪ {execution}
15             break
16 return selection
```

D. Execution Selection (ES)

Our approach relies on the replay of executions collected from a base model. This process can be very time consuming, especially for distributed systems with several nodes where the number of traces collected from each execution can be large. Our approach aims to mitigate this issue by using the function *executionSelection* shown in Algorithm 1. The output of this function can be used in multiple regression detection (i.e., Algorithm 2) experiments with different modified models. Consequently, with a fixed set of critical variables, as long as the base model does not change, the result produced by *ES* does not need be updated. When a modification is approved as regression-free, the current base model can be replaced with the modified model and then *ES* needs to be re-run for the new base model. As a high-level overview, Algorithm 1 reorders and replays the input execution to determine and return the executions along which at least one of the critical variables is assigned to at least once. In other words, the selection leverages the fact that executions along which no critical variable is ever assigned to cannot uncover a regression.

The function *executionSelection* uses a nested loop in which the outer loop iterates over all executions collected from a base model, and the inner loop replays rc-steps to find executions that use critical variables in their computations. In line #3, configuration is initialized with γ_0 where initial state is assigned to its active state, as well as default values are assigned to all the variables and attributes of its value map, i.e., $\gamma.E$. Then, (possibly out of order) traces of *execution* are assigned to *inTraces* and the message queue, i.e., *msg* is reset with the initial message *startUp* in line #4-5. The *while* loop checks if all traces in *inTraces* not consumed (line #6). In the line #8, the function *getRCStep*(*msg*,*rcSteps*,γ) is used to find the matching rc-step, i.e., $rcStep$ with respect to *msg*, list of rc-steps, i.e., *rcSteps* and the current configuration, i.e., γ. Well-formedness condition `C1` implies that an incoming message msg can enable at most one rc-step. Then, the function *replay* is used to replay *rcStep* and create a new configuration, i.e., γ (line #10). According to the condition `C2` the execution of an rc-step cannot get stuck. The function *variables* is used to select only the variables being assigned to in any of the action code blocks (i.e., *actSeq*). If any of the variables in *var* are critical, the execution is added into the set of selected executions (i.e., *selection*) that this function outputs.

For example, given the sample set of executions in Fig. 2, as well as the set of critical variables *critVar* (i.e., $\{$*bamt*$\}$) and rc-steps of *CTR* in Table I. The function *executionSelection* returns *selection* including the executions E_2 and E_3. In fact, it realizes that the critical variable *bamt* is modified in the rc-steps rc_4^C and rc_5^C corresponding to the traces t_8^C and t_9^C in E_2 and E_3, respectively. So, E_2 and E_3 are added into the set *selection*. Moreover, if users want to perform regression testing at a finer granularity, they can include the variable *reject* in t_5^C into the set of existing critical variables. As a result, the function *executionSelection* will output more executions (i.e., E_1, E_2, and E_3) which leads to replay of more executions in the function *regressionDetection*.

E. Regression Detection (RD)

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

Step #1: The algorithm reorders (possibly out of order) traces and replays them on the modified capsule *C′* (line #5-15). Similar to the Algorithm 1 (line #8), the function *getRCStep* outputs an rc-step which is unique according to the well-formedness condition `C1`. In the line #10, the function *removeMatchingTrace* looks for the matching trace in *inTraces*. This function blocks until a trace matching the rc-step *rcStep* appears in *inTraces* or until a timer defined by the abstract controller expires. The timer is set to a user-defined value and starts counting down if replay is deadlocked, i.e., if the execution of function *regressionDetection* in all abstract capsules is stuck at line #10. Note that the timeout value can be small, because the replay is centralized and does not involve network communication (also note that in Algorithm 1, function *removeMatchingTrace* will never block and wait, because that algorithm simply replays base model executions on the same base model and so the deadlock situation described above is impossible). Finally, the function *replay* is used to replay *rcStep′* and create a new configuration, i.e., γ' (line #13).

Step #2: The algorithm uses the same replayer mechanism to re-generate variable values based on the original capsule *C*. To this end, it identifies a unique rc-step from the set of all possible rc-steps, i.e., *rcSteps* in the original capsule *C* using the same function *getRCStep*. Then, the function *replay* is used to replay the rc-step determined in the line #16 and create a new configuration γ for the original capsule *C* (line #19). In the line #21, the algorithm uses the boolean function *different* that takes the value maps, i.e., $vMap$ and $vMap'$, and checks whether value of critical variables are consistent. If yes, this function returns false and the algorithm goes back to the line #4. If no, the function *difference* assigns changed critical variables to the *regressions*, breaks the loop and returns the *regression* (line #24).

F. Example

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

Algorithm 2: *regressionDetection*(*rcSteps: set of rc-steps of C, rcSteps′: set of rc-steps of C′, inTraces: sequence of incoming traces of C′, critVar: set of critical variables*)

```
1  Let regressions be an empty string
2  γ, γ′ ← γ0
3  msgs ← append(msgs, startUp)
4  while (inTraces not consumed) do
5      // Step#1:Reorder & replay on the modified
          capsule C′
6      msg ← dequeue(msgs)
7      rcStep′, γ1′ ← getRCStep(msg, rcSteps′, γ′)
8      if (rcStep′ = ∅) then
9          regressions ←'NoRCStep'
10         break
11     if (removeMatchingTrace(inTraces, rcStep′)) then
12         regressions ← 'NoMatchingTrace'
13         break
14     γ′ ← replay(rcStep′, γ1′)
15     vMap′ ← values(γ′.E, critVar)
16     // Step#2:Replay on the original capsule C
17     rcStep, γ1 ← getRCStep(msg, rcSteps, γ)
18     if (rcStep = ∅) then
19         regressions ←'NoRCStep'
20         break
21     γ ← replay(rcStep, γ1)
22     vMap ← values(γ.E, critVar)
23     if (different(vMap, vMap′)) then
24         regressions ← difference(vMap, vMap′)
25         break
26 return regressions
```

critical variable *bamt*. Then, in Step 16 the function *different* identifies the inconsistency between the value of *bamt* in the modified model and the base model. So, *bamt* is assigned to *regressions* and the algorithm terminates.

IV. Experimental Evaluation

In this section, we use the evaluation framework for regression test selection techniques proposed by Rothermel et al. [33]. The framework was originally designed for code-based techniques but Briand et al. [47] showed that most of the principles of Rothermel's framework can be applied to regression testing techniques at a model level.

A. Prototype Implementation

MRegTest has been developed using Java on a Linux (Ubuntu) environment running on a desktop machine equipped with a 2.7GHz Intel Core i5 and 8GB of memory. Also, we implemented our approach in Eclipse Papyrus-RT for distributed systems [26], which is an industrial-grade and open-source MDD tool. We used the Epsilon Object Language (EOL) [48] to implement the transformation rules required for instrumentation of the models. Source code of the implementation along with documentation is available at [49].

B. Evaluation Approach

Use cases. To evaluate our approach, several use cases are used. The complexity of the models used (Table II) ranges from a model with 5 capsules, 19 states, and 29 transitions (i.e., SATM) to a model with 13 capsules, 2,304 states and 3,647 transitions (i.e., RFO). The model for a simplified Automated Teller Machine (SATM) is described in Section II, the FailOver system (FO) in [50], [51], Parcel Router (PR) [52], [53], Simplified Parcel Router (SPR), Rover control system (RO) are described in [54], [55]. Automated Teller Machine (ATM), which is a more realistic version of SATM, is available in [56]. Refined FailOver (ROF) is a debuggable version of a FailOver system generated using MDebugger [54]. In the following, we discuss the metrics and detail the experiments used to evaluate our approach.

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

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

Reduction in the Number of Executions (EXP-3). In this experiment we evaluate the efficiency of the first step of our approach (i.e., Algorithm 1) to select regression-revealing executions based on a given set of critical variables. The same setup as EXP-1 is used in this experiment. We measured how many executions out of total 1,000 executions are selected by the function *executionSelection*, and then similar to [19] we calculated reduction for each use case.

Evaluating the Effectiveness of *ES* (EXP-4). This experiment measures the extent to which our approach chooses regression-revealing executions that were manually generated. We adopted the same setup as in EXP-1. Then we used Algorithm 1 to select executions based on a given set of critical variables for each use case. We evaluated the effectiveness of *ES* using the two widely used metrics, i.e., *precision* and *recall*. The former basically measures how accurate Algo-

rithm 1 selects those executions that cause regression. More specifically, precision measures the *True Positives* divided by *True Positives* + *False Positives*. The latter aims to highlight the amount of regression-revealing executions that were incorrectly not selected (i.e., *False Negative*) [57].

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

Evaluating the Effectiveness of *RD* (EXP-6). Using this experiment we evaluated the effectiveness of Algorithm 2 in terms of the number of faulty mutants it detects as regression. We generated 100 mutants from each use case including faulty and non-faulty mutants. We also ensured that for every faulty mutant, at least one execution is collected from the base model. In this experiment, the output of Algorithm 1 in EXP-3 is used to provide a set of executions for Algorithm 2. For every mutant of the use cases in Table II we ran our regression testing approach using identical deployment configurations and measured the number of faulty mutants that had been detected as regression. Finally we calculated *precision* and *recall* for each use case.

Experimental Environment. We used the development environment explained above for experiments EXP-1 to EXP-6. Also, we used Java version 1.8.0161 in configuration -Xmx12512m. The source code of the experiments and models are publicly available at [49].

C. Results and Discussions

In this paper, we propose the use of *MRegTest* to reduce the cost of replay-based regression testing for distributed systems. Since we found no previous works with the same goal (i.e., replay-based regression testing for distributed systems in the context of MDD) , we compare *MRegTest* with an implementation of our work that annotates traces with timestamps and variable values as baseline. Also, we do not compare our approach with orthogonal techniques that generate test cases (e.g., [58]) because test case generation is outside the scope of this work. We set a time-budget of 8 hours when analyzing each test case, which is in line with industrial practice (e.g., nightly-build-and-test) [19].

Time-Efficiency. It is important that processing time of the algorithms in our approach (in Section III) remains within an acceptable range even when model size grows exponentially. Based on EXP-1, the columns labelled *Time* in Table II show the time required to select relevant executions and detect regressions in Algorithm 1 (i.e., *ES*) and Algorithm 2 (i.e., *RD*), respectively. In the worst case (i.e., the largest model), *ES* takes $6{,}609$ ms and *RD* takes $34{,}041$ ms. Going from *FO* to *RFO*, the number of capsules, states, and transitions increase by factors 1.8, 74, and 84, respectively; however, *ES* and *RD* times only increase by factors 8.4 and 14.9, respectively. While processing time increases with model size, the results show that the processing times are reasonable and do not grow exponentially.

Size-Efficiency. As opposed to traditional approaches that annotate traces with timestamps and variable values, our approach re-generates variable values by replay which may lead to a significant reduction in the size of generated traces. In Table II column *avgSRed.* shows results of the first part of EXP-2. This indicates the amount of reduction in sizes of traces collected from our approach compared with sizes of traces collected from the traditional approach. The reduction ranges from $1.27\times$ to $2.13\times$, with a geometric mean of $1.56\times$. Results of the second part of EXP-2 for two use cases (i.e., *ATM* and *FO*) are illustrated in the line-chart at the bottom of Figure 4. It shows in both use cases our approach, that uses MReplayer's instrumentation technique, generates smaller sizes of traces. For example, the size of $400{,}000$ traces of the *ATM* in our approach (i.e., *ATM-Ours*) is 72.9MB. Whereas the size of traces generated from the traditional approach (i.e., *ATM-Trad.*) is 1.66 times higher, i.e., 121.3MB. The difference is getting even larger for $500{,}000$ traces (from the same model) where the size of traces generated in our approach is 1.98 times smaller than that of the traditional approach.

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

Runtime Overhead. It studies the runtime overhead of Algorithm 1 and Algorithm 2 for replay of traces in use cases with various complexity. In Table II columns *avgROver.* (under *ES* and *RD*) show average runtime overhead for both algorithms that are results of the first part of EXP-5. They show execution time increases in all use cases mainly due to the use of the replayer for re-generating variable values. Nonetheless, the imposed runtime overhead for all use cases in *ES* ranges from $1.01\times$ to $1.12\times$ and in *RD* ranges from $1.01\times$ to $1.17\times$. Also, for all use cases runtime overhead of *RD* is higher than that of *ES*. The line-chart at the top of Figure 4 only illustrates execution time of *RD* for two use cases (i.e., *ATM* and *FO*). It shows that in both use cases the difference between execution time of *RD* in our approach and that of the traditional approach is in the order of few seconds. This implies that benefit of

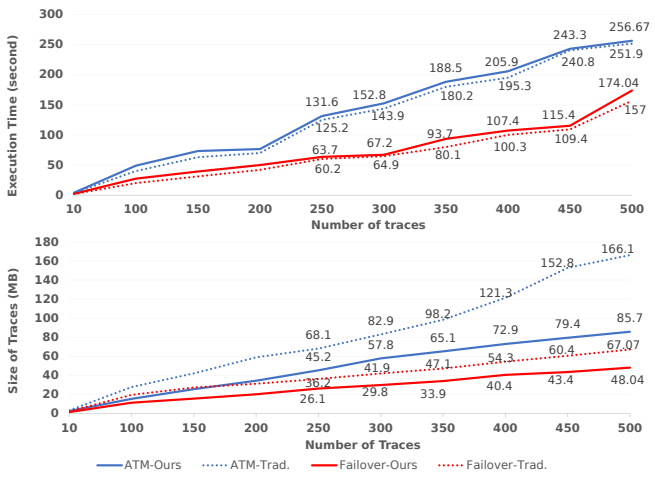

Figure 4: Cost and benefit of our approach

replaying outweighs its costs (i.e., slowing down the analysis).

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

Effectiveness of *RD*. Similar to *ES*, we evaluate the effectiveness of Algorithm 2 with respect to *precision* and *recall* and the results are shown in the columns with the same name under the column *RD* in Table II. For all use cases, precision and recall range from 0.94 to 1.00 and from 0.97 to 1.00, respectively. Since in this evaluation we used the output of the EXP-3, missing some regression-revealing executions in Algorithm 1 will reduce the capability of Algorithm 2 for detecting all regression. Nevertheless, the results indicate that precision and recall in most use cases are high enough to conclude that Algorithm 2 is accurate to detect large amount of mutants that cause regressions in use cases with different complexity.

D. Threats to validity

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

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

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

V. Related Work

At a high-level we can group existing studies based on how much control they exert over the execution of the distributed system for the purposes of regression testing. Also, one can distinguish two kinds of test methods: centralized, i.e., one tester for all nodes; decentralized, i.e., local tester at each node [59]. As controllable aspects of the system and its execution environment we consider: (1) message delivery; (2) timing of executions; (3) non-deterministic operations. This creates a kind of spectrum of regression testing techniques that can be either centralized or decentralized.

At one end, centralized approaches such as *SimRT* [20], *ReConTest* [19], and *ConTesa* [58] exert no (or low) control over an execution during the testing. Since they typically require accurate clock synchronization among nodes, deterministic re-execution is hard to achieve. Also, due to the lack of control over the executions, optimizations are harder to realize, e.g. what happens if 10K of a certain message have been exchanged between two nodes.

Replay-based approaches and tools such as *QF-Test* [9], *Selenium* [10], and *TestComplete* [11] typically lie on the other end of the spectrum on which a fair bit of control is exerted over the execution. However, certain kinds of control such as message delivery mechanism are not feasible due to technical limitations or lack of access. Also, imposing large runtime overhead by excessive instrumentation might put the accuracy of testing at risk, because replay of collected traces might not be a good representative of the execution in the original system. Our approach is a replay-based centralized tester that benefits from a low-overhead instrumentation and offers sufficient control over the execution. It leverages MReplayer's action code interpreter to select and execute the proper rc-step at any step of the execution. Our approach differs from MReplayer in that we use the reorder and replay mechanism to detect regressions.

According to the survey [59], there are many approaches that use deterministic replay of traces for testing, while they re-generate variable values via re-execution of the action code, e.g., [2], [3], [60], [5], [61]. Similar to our work, a low-overhead instrumentation approach is used in *FlowTesting* [2]. Also, it enables the user to have fine-grained control over the execution of a target system. Similarly, [3] is a decentralized

Table II: Complexity of Use-cases, Processing Time, Runtime Overhead, Size Reduction, Efficiency, and Effectiveness

Model	Complexity			Inst.	Algorithm 1 (*ES*)					Algorithm 2 (*RD*)			
					Efficiency			Effectiveness		Efficiency		Effectiveness	
	#C	*#S*	*#T*	*avgSRed.*	*Time(ms)*	*avgROver.*	*exeRed.*	*P*	*R*	*Time(ms)*	*avgROver.*	*P*	*R*
SATM	5	19	29	1.43×	115	1.01×	1.52×	1.00	1.00	663	1.04×	1.00	1.00
CDCL	5	11	15	1.32×	234	1.01×	1.84×	1.00	1.00	475	1.03×	1.00	1.00
SPR	8	12	14	1.27×	494	1.02×	3.43×	1.00	1.00	955	1.04×	1.00	1.00
PR	8	14	25	1.45×	661	1.02×	3.18×	0.98	1.00	1,088	1.03×	0.95	1.00
RO	6	16	21	1.56×	703	1.03×	4.67×	0.97	0.98	2,139	1.04×	0.96	0.98
FO	7	31	43	1.39×	783	1.05×	5.43×	0.97	1.00	2,282	1.10×	0.95	1.00
ATM	12	295	349	1.98×	2,259	1.01×	6.89×	0.95	0.98	17,582	1.01×	0.94	0.97
RFO	13	2304	3647	2.13×	6,609	1.12×	10.44×	0.95	0.97	34,041	1.17×	0.94	0.97

#C: number of capsules, *#S*: number of states, *#T*: number of transitions, *Inst.*: Instrumentation, *ms*: milliseconds, *P*: Precision, *R*: Recall

approach that aims to reduce the overhead on nodes, as well as offers an 'interpreter' to detect *expected messages*.

Moreover, the following groups of existing work seem most relevant.

Regression test selection (RTS). According to existing surveys, e.g., [62], [63], [64], most of RTS techniques can be categorized into 5 groups based on granularity of software elements in test dependencies (i.e., the software elements that can be executed during each test execution): (1) basic-block-level, e.g., [65], (2) method-level, e.g., [66], (3) file-level, e.g., [67], [68], (4) module-level, e.g., [69] and (5) hybrid e.g., [70]. Although RTS approaches working on coarser granularities may have lower overhead, techniques based on finer granularities tend to be more precise in selecting tests. Our approach works on basic-block-level granularity, because it selects executions based on critical variables which can be modified in basic elements of a system at the model level (i.e., transitions). Also, depending on how the test dependencies are collected, RTS techniques can be categorized as dynamic [67], [71], [72], [66] and static [68], [73]. Unlike dynamic RTS, static RTS requires no code instrumentation or runtime information to find impacted test cases. Furthermore, static RTS uses static analysis to over-approximate the test dependencies and thus may select more tests than necessary [70]. Our approach falls within the scope of dynamic RTS category. Nevertheless, unlike the existing studies, that typically use re-execution for detecting regressions, our approach relies on replay of execution traces collected from a base model on its modified version.

Regression testing in the context of MDE. There are several approaches using model level information for regression testing [74], [75], [76], [77], [78]. Biswas et al. [79] proposed a model-based RTS technique that constructs a graph model from a program representing characteristics that are important for RTS. Their approach is not adequate for testing distributed systems because it relies on a few assumptions (e.g., synchronous message passing) that appear unreasonable for distributed systems. Zech et al. [80], [81] introduced a generic platform for model-based regression testing based on the model versioning engine MoVE and additional Object Constraint Language (OCL) statements. It also explains the process of generation and selection of regression test cases from the UML basic behavioural models. Similarly, Honfi et al. [78] proposed RtsMoT which is a model-based RTS approach for reconfigurable, autonomous robots. However, none of this work is applicable for distributed systems. Pal et al. [61] addressed this issue by introducing a testing framework for real-time distributed systems which is applicable to model refinements with new specifications to performing model-based distributed regressing testing. Similar to our approach online monitored data is used to obtain a coherent view of a distributed system. In contrast to the architecture of our approach which is centralized, their approach essentially uses the partitioning algorithm proposed in [82] to decompose a centralized tester into a set of communicating distributed local testers. Korel et al. [83] proposed a model-based RTS method based on Extended Finite State Machines (EFSM). Their approach considers only two types of Elementary Modifications (EM) on a model (i.e., addition or deletion of a transition). Chen et al. [84] addressed this issue by considering *change in a transition* as another EM type. In contrast, we reorder execution traces before doing regression testing, because we assume data is transmitted between components through an asynchronous message passing protocol. Furthermore, they don't replay base model executions, as opposed to our approach; instead, they run the modified model multiple times to detect regressions. Also, there is some work on using UML models (i.e., class diagrams, collaboration diagrams, and statecharts) for regression testing [85], [47], [86], [87], [88].

VI. Conclusion and Future Work

In this paper we presented *MRegTest*, a replay-based regression testing approach for distributed systems at the model-level, which is capable of detecting regressions by replay of traces collected from a base model. *MRegTest* reduces the cost of regression testing by replaying only those executions that modify critical variables specified by the user. Moreover, *MRegTest* re-generates variable values by replay of the action code as opposed to the traditional approaches that read the information from traces directly. Our experimental results showed that *MRegTest* detects almost all regressions in use cases with various complexity. Also, it reduces the size of collected traces significantly while imposing negligible runtime overhead.

As future work, we intend to extend our approach to handle other sources of modifications in a behavioral model such as

some expressions in action code, as well as modifications in a structural model such as ports and protocols of a capsule.

ACKNOWLEDGMENT

This work has been supported by Zeligsoft Ltd, Malina Software Corp., Cmind Inc., and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–245.

[2] E. Gabrielova, "End-to-end regression testing for distributed systems," ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 9–12.

[3] D. Aumayr, S. Marr, C. Béra, E. G. Boix, and H. Mössenböck, "Efficient and deterministic record & replay for actor languages," ser. ManLang '18. New York, NY, USA: Association for Computing Machinery, 2018.

[4] L. Tveito, E. B. Johnsen, and R. Schlatte, "Global reproducibility through local control for distributed active objects," in *FASE*, ser. Lecture Notes in Computer Science, vol. 12076. Springer, 2020, pp. 140–160.

[5] I. Lanese, A. Palacios, and G. Vidal, "Causal-consistent replay debugging for message passing programs," in *IFIP*, ser. Lecture Notes in Computer Science, vol. 11535. Springer, 2019, pp. 167–184.

[6] K. Kazuhiro Shibanai and T. Watanabe, "Distributed functional reactive programming on actor-based runtime," in *SIGPLAN*. ACM, 2018, pp. 13–22.

[7] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," ser. ESEC/FSE. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498.

[8] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: Automating amp; sharing how-to knowledge in the enterprise," in *SIGCHI*, ser. CHI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1719–1728.

[9] "QF-Test website," https://www.qfs.de/en.html, retrieved 2021.

[10] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *AGILE'06*, 2006.

[11] Smartbear, "TestComplete," https://smartbear.com/product/testcomplete/overview/, retrieved 2021.

[12] A. Kresse and P. M. Kruse, "Development and maintenance efforts testing graphical user interfaces: A comparison," ser. A-TEST. New York, NY, USA: Association for Computing Machinery, 2016, p. 52–58.

[13] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[14] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 25–36.

[15] M. Babaei, M. Bagherzadeh, and J. Dingel, "Efficient reordering and replay of execution traces of distributed reactive systems in the context of model-driven development," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 285–296.

[16] H. Ural and D. Whittier, "Distributed testing without encountering controllability and observability problems," *Information Processing Letters*, vol. 88, no. 3, pp. 133 – 141, 2003.

[17] R. M. Hierons, M. G. Merayo, and M. Nunez, "Controllability through nondeterminism in distributed testing," in *Testing Software and Systems*, F. Wotawa, M. Nica, and N. Kushik, Eds. Cham: Springer International Publishing, 2016, pp. 89–105.

[18] L. Cacciari and O. Rafiq, "Controllability and observability in distributed testing," *Information and Software Technology*, vol. 41, no. 11, pp. 767 – 780, 1999.

[19] V. Terragni, S. Cheung, and C. Zhang, "Recontest: Effective regression testing of concurrent programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 246–256.

[20] T. Yu, W. Srisa-an, and G. Rothermel, "SimRT: An automated framework to support regression testing for data races," ser. ICSE. New York, NY, USA: Association for Computing Machinery, 2014, p. 48–59.

[21] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "SETSUDundefined: Perturbation-based testing framework for scalable distributed systems," ser. TRIOS. New York, NY, USA: Association for Computing Machinery, 2013.

[22] B. Lima, "Automated scenario-based integration testing of time-constrained distributed systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 486–488.

[23] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 747–783, 2018.

[24] C. Torens and L. Ebrecht, "RemoteTest: A framework for testing distributed systems," 2010, pp. 441–446.

[25] G. Milka and K. Rzadca, "Dfuntest: A testing framework for distributed applications." CoRR, 2018.

[26] Karim Jahed, "Papyrus-RT Distribution," https://github.com/kjahed/papyrusrt-distribution, retrieved: 2021.

[27] M. Babaei, M. Bagherzadeh, and J. Dingel, "Mreplayer: A trace replayer of distributed uml-rt models," ser. MODELS'20. New York, NY, USA: Association for Computing Machinery, 2020.

[28] The Object Management Group, "UML Superstructure Specification. Version 2.5.1," http://www.omg.org/spec/UML/2.5.1/PDF, 2017, retrieved May 15, 2020.

[29] G. Stivan, A. Peruffo, and P. Haller, "Akka.js: Towards a portable actor runtime environment," ser. AGERE. New York, NY, USA: Association for Computing Machinery, 2015, p. 57–64.

[30] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," ser. SOCC. New York, NY, USA: Association for Computing Machinery, 2011.

[31] C. McCaffrey, "Building the Halo 4 services with Orleans," 2015, presentation at QCon London.

[32] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang, "A survey of active object languages," *ACM Comput. Surv.*, vol. 50, no. 5, Oct. 2017.

[33] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[34] M. Hammoudi, "Regression testing of web applications using record/replay tools," ser. FSE. New York, NY, USA: Association for Computing Machinery, 2016, p. 1079–1081.

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

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

[37] IBM, "RSARTE," https://www.ibm.com/developerworks/community/wikis, retrieved 2020.

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

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

[40] E. Foundation, "Eclipse Papyrus for real time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, 2016, retrieved May 15, 2020.

[41] B. Selic, "Using UML for modeling complex real-time systems," in *Languages, compilers, and tools for embedded systems*. Springer, 1998, pp. 250–260.

[42] S. Esmaeilsabzali, N. Day, J. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages," *Requir. Eng.*, vol. 15, pp. 235–265, 06 2010.

[43] UML2.5.1, "Uml2.5.1," https://www.omg.org/spec/UML/2.5.1/PDF, 2017, retrieved June 5, 2019.

[44] R. Milner, *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag, 1982.

[45] C. A. R. Hoare, "Communicating sequential processes," vol. 21, no. 8, p. 666–677, Aug. 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[46] T. Parr, "Antlr (another tool for language recognition)," https://www.antlr.org/, 2017, retrieved June 5, 2019.

[47] L. Briand, Y. Labiche, and S. He, "Automating regression test selection based on uml designs," *Information and Software Technology*, vol. 51, no. 1, pp. 16 – 30, 2009.

[48] 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.

[49] blind for reviewer, "MRegTest repository," blind for reviewer, retrieved 2021.

[50] 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.

[51] N. Kahani, N. Hili, J. R. Cordy, and J. Dingel, "Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems." IEEE, 2017, pp. 12–18.

[52] 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.

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

[54] 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.

[55] M. Bagherzadeh, "Model-level debugging in the context of the model driven development," PhD dissertation, Kingston, Ontario, Canada, 2019.

[56] "ATM UML-RT model," https://github.com/MajidGitHubRepos/atm_umlrt, retrieved 2021.

[57] E. D. Ekelund and E. Engström, "Efficient regression testing based on test history: An industrial evaluation," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 449–457.

[58] T. Yu, Z. Huang, and C. Wang, "ConTesa: Directed test suite augmentation for concurrent software," *IEEE Transactions on Software Engineering*, vol. 46, no. 4, pp. 405–419, 2020.

[59] 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.

[60] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An incremental approach for repairing record-replay tests of web applications," ser. FSE. New York, NY, USA: Association for Computing Machinery, 2016, p. 751–762.

[61] D. Pal and J. Vain, "A systematic approach on modeling refinement and regression testing of real-time distributed systems," *IFAC*, vol. 52, no. 13, pp. 1091–1096, 2019.

[62] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, p. 67–120, Mar. 2012.

[63] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica (Slovenia)*, vol. 35, no. 3, pp. 289–321, 2011.

[64] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," vol. 52, no. 1, p. 14–30, Jan. 2010.

[65] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, p. 241–251, Oct. 2004.

[66] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 23–32.

[67] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 211–222.

[68] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," ser. FSE. New York, NY, USA: Association for Computing Machinery, 2016, p. 583–594.

[69] G. E. Tools, "Build in the cloud: How the build system works," http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html, 2021, retrieved Jan 30, 2021.

[70] L. Zhang, "Hybrid regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 199–209.

[71] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 312–326.

[72] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," ser. SIGSOFT'04. New York, NY, USA: Association for Computing Machinery, 2004, p. 241–251.

[73] O. Legunsen, A. Shi, and D. Marinov, "STARTS:STAtic regression test selection," 2017, pp. 949–954.

[74] N. Almasri, L. Tahat, and B. Korel, "Automatically quantifying the impact of a change in systems (journal-first abstract)," ser. ASE. New York, NY, USA: Association for Computing Machinery, 2018, p. 952.

[75] Y. Chen, R. L. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," ser. A-MOST '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 54–62.

[76] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "A model-based regression test selection approach for embedded applications," *SIGSOFT Softw. Eng. Notes*, vol. 34, no. 4, p. 1–9, Jul. 2009.

[77] C. R. Panigrahi and R. Mall, "Model-based regression test case prioritization," in *Information Systems, Technology and Management*, S. K. Prasad, H. M. Vin, S. Sahni, M. P. Jaiswal, and B. Thipakorn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 380–385.

[78] D. Honfi, G. Molnár, Z. Micskei, and I. Majzik, "Model-based regression testing of autonomous robots," in *SDL 2017: Model-Driven Engineering for Future Internet*, T. Csöndes, G. Kovács, and G. Réthy, Eds. Cham: Springer International Publishing, 2017, pp. 119–135.

[79] S. Biswas, R. Mall, and M. Satpathy, "A regression test selection technique for embedded software," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3, Dec. 2013.

[80] P. Zech, P. Kalb, M. Felderer, C. Atkinson, and R. Breu, "Model-based regression testing by OCL," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 1, p. 115–131, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s10009-015-0408-8

[81] P. Zech, M. Felderer, P. Kalb, and R. Breu, "A generic platform for model-based regression testing," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 112–126.

[82] D. Pal and J. Vain, "Model based approach for testing: Distributed real-time systems augmented with online monitors," in *Databases and Information Systems*, A. Lupeikiene, O. Vasilecas, and G. Dzemyda, Eds. Cham: Springer International Publishing, 2018, pp. 142–157.

[83] B. Korel, L. H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *International Conference on Software Maintenance, 2002. Proceedings.*, 2002, pp. 214–223.

[84] Y. Chen, R. L. Probert, and H. Ural, "Regression test suite reduction using extended dependence analysis," ser. SOQUA'07. New York, NY, USA: Association for Computing Machinery, 2007, p. 62–69.

[85] Ye Wu and J. Offutt, "Maintaining evolving component-based software with uml," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003, pp. 133–142.

[86] L. C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on uml designs," 2002, pp. 252–261.

[87] Q.-u.-a. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem, "An approach for selective state machine based regression testing," ser. A-MOST'07. New York, NY, USA: Association for Computing Machinery, 2007, p. 44–52.

[88] Y.Le Traon, T.Jeron, J.Jezequel, and P.Morel, "Efficient object-oriented integration and regression testing," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 12–25, 2000.