Effective Test Suite Design for Detecting Concurrency Control Faults in Distributed Transaction Systems

Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu

School of Innovation, Design and Engineering, Mälardalen University,
Västerås, Sweden
{simin.cai,barbara.gallina,
dag.nystrom,cristina.seceleanu}@mdh.se

Abstract. Concurrency control faults may lead to unwanted interleavings, and breach data consistency in distributed transaction systems. However, due to the unpredictable delays between sites, detecting concurrency control faults in distributed transaction systems is difficult. In this paper, we propose a methodology, relying on model-based testing and mutation testing, for designing test cases in order to detect such faults. The generated test inputs are designated delays between distributed operations, while the outputs are the occurrence of unwanted interleavings that are consequences of the concurrency control faults. We mutate the distributed transaction specification with common concurrency control faults, and model them as UPPAAL timed automata, in which designated delays are encoded as stopwatches. Test cases are generated via reachability analysis using UPPAAL Model Checker, and are selected to form an effective test suite. Our methodology can reduce redundant test cases, and find the appropriate delays to detect concurrency control faults effectively.

1 Introduction

In many modern software systems, data are partitioned in several nodes across the network, and accessed by concurrent distributed transactions with read and write operations. Without proper control, concurrent transactions may interleave and access data arbitrarily, which may lead to inconsistent data. For instance, in a distributed automation system, whose configuration data are partitioned on different sites, a transaction may update the configuration data D1 on site S1 and data D2 on site S2, based on their current values. If another transaction modifies D2 exactly before the former's update, the configurations may end up inconsistent. To avoid this, the transaction manager often ensures the isolation property, that is, the absence of a specified set of transaction interleavings that cause data inconsistency [1].

To achieve this, lock-based Concurrency Control (CC) techniques are often applied by the transaction manager to prevent unwanted interleavings [4]. Such type of CC regulates transactions to acquire and release locks on data at specific times, and resolves the conflict when two transactions request the same lock simultaneously. In the previous example, one proper way to guarantee isolation could be to lock both D1 and D2 until the modification is completed, so that no other transactions can interfere. Consequently, every transaction behaves to its caller as if it was the only one executed in the system.

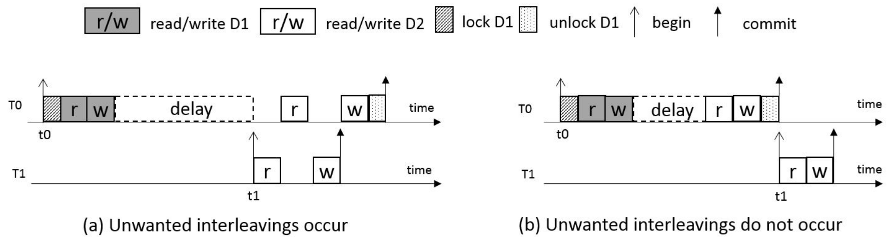

Fig. 1. Observation of isolation violation depending on delays

In this paper, we focus on detecting CC faults in a distributed transaction system with lock-based concurrency control. CC faults are commonly found and notoriously challenging to detect in software systems [25]. Common CC faults include, for instance, misplaced locks, or erroneous time of releasing locks [17, 23]. The consequence could be the violation of the isolation requirements, that is, the occurrence of undesired interleavings of the concurrent transactions. To detect the CC faults, the implemented system are usually tested by executing the distributed transactions, and monitoring the occurrence of the undesired interleavings in the transaction logs. A fault is detected when such undesired interleavings are observed. Due to the various communication delays between different nodes, however, such undesired interleavings may not be observed, even in the presence of CC faults. In order to illustrate the impact of delays, let us consider two transactions, T0 and T1, from the previous example. T0 updates D1 and D2 by reading and write D1 on site S1, and after some delay, followed by reading and write D2 on S2. T1 reads D2 and then modifies its value. The isolation property forbids the following sequence of operations: T0 reads D2, T1 reads D2, T0 writes D2, T1 writes D2, because T1 bases its write on an intermediate value of D2. While a proper CC algorithm would be to lock both D1 and D2 during the execution of T0, we assume that the developer has made a mistake by forgetting to lock D2. However, isolation violation may not be observed with this fault. As shown in Fig. 1a, the unwanted interleavings does occur that T1 reads D2 before the final modification by T0, and can be observed by the tester. However, if the delay is shorter, as shown in Fig. 1b, the unwanted interleavings do not occur, and thus the fault is not exposed to the tester.

Traditionally, a common testing technique is to insert random delays between the operations of the transactions, and test the violation of isolation [12, 16]. However, even though a large set of various delays can be used as test inputs, it is still difficult to ensure the coverage of the unwanted interleavings. Finding a relatively small set of test cases, which is manageable yet able to expose as many unwanted interleavings as possible, remains challenging for detecting CC faults of distributed transactions.

To address this challenge, in this paper we propose a methodology for finding an effective test suite that can expose the CC faults. The inputs of the test cases are designated delays between operations at design level, which can be used to configure the delays in the implemented system in a controlled testing environment. Instead of randomly chosen delay values, we propose techniques to generate a set of values for the inserted delays as inputs, such that the CC faults can be exposed. The outputs are whether or not the predefined unwanted interleavings that violate isolation have occurred, which can be examined in the logs. The techniques central to this methodology are model-based testing [26] and mutation testing [21]. We define a set of mutation operators, each altering the transaction specification by introducing a common CC fault. For instance, one mutant operator could change the specification by removing the locking of a data. We model the transaction specification, as well as its mutants, as networks of UPPAAL Timed Automata (TA) [24], in which the delays are encoded as stopwatches [8]. Test cases including the designated delays are generated from the models via reachability analysis.

The process of applying our methodology is listed as follows. (i) We specify the work units consisting of the operations on the distributed data, the delays between these operations, as well as the lock operations, in a high-level description language. In addition, we specify the requirements of desired isolation, which are the interleavings to be prevented by CC. (ii) By extending a modeling framework for concurrent transactions in our previous work [7], we model the distributed transaction system as a network of UPPAAL TA, and formalize these isolation properties in Computational Tree Logic (CTL) [9], which can be checked rigorously by the UPPAAL Model Checker (MC) [24]. (iii) The specification in (i) is mutated by applying the mutation operators, based on which we create a series of mutant TA models that model the common CC faults. (iv) We generate diagnostic traces for each mutant model via reachability analysis and obtain test cases from the traces, which are used to form an effective test suite that can kill all the mutants with a minimal number of test cases.

The remainder of the paper is organized as follows. Section 2 recalls the background knowledge of this paper, including model-based and mutation testing, UPPAAL TA and the UPPAAL tools. Section 3 presents the details of our methodology. In Section 4, we apply our methodology to a case study, followed by a comparison with related work in Section 5. In Section 6, we conclude the paper and outline our future work.

2 Background

In this subsection, we present the background knowledge of this paper, that is, a brief overview of model-based and mutation testing in Section 2.1, as well as UPPAAL timed automata and the UPPAAL tools in Section 2.2.

2.1 Model-based Testing and Mutation Testing

Model-based testing [26,33] encompasses the processes and techniques to perform testing based on behavioral or architectural models of the System Under Test (SUT). A generic process of model-based testing consists of the following major steps. (1) Create a model of the SUT. (2) Decide the test selection criteria to guide test generation. (3) Define the high-level test case specifications. (4) Generate the test cases that satisfy the test case specifications. (5) Run the generated test cases, manually or automatically.

Mutation testing [21] is a fault-based technique to provide criteria for selecting the effective test cases. Mutants, which represent common faults, are created, upon which the candidate test cases are executed. Combined with model-based testing, the mutants

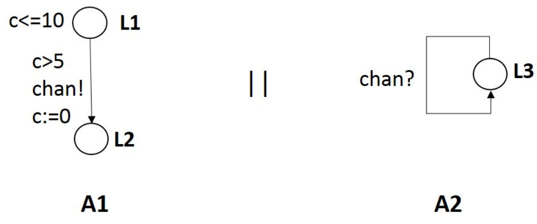

Fig. 2. A network of UPPAAL TA

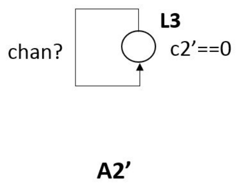

Fig. 3. An UPPAAL TA with stopwatch

are also models generated from the original system model using pre-defined mutation operators. A good test case should be able to distinguish the mutated behaviors from the original behavior. An effective test suite, selected from the candidate test cases, should be able to kill as many mutants as possible, with a minimal number of test cases.

2.2 UPPAAL Timed Automata and the UPPAAL Tools

In this paper we use Timed Automata (TA) [2] to model the distributed transaction system. A Timed Automaton (TA) is a finite-state automaton extended with clock variables with real-type values. These clock variables progress synchronously, and are used to model the progress of time in a system. UPPAAL [24] is the state-of-art model checker for verifying TA-based models of real-time systems.

Fig. 2 shows two TA, A1 and A2, in UPPAAL. A1 has two locations, *L1* and *L2*, and has defined a clock variable c to keep track of the elapsed time. An directed edge connects these two locations, meaning the it is possible to transit from *L1* and *L2*. When a TA reaches a location, it can non-deterministically choose to delay at the same location, or take a transition to another location following an edge. An invariant, which is a propositional formula over clock variables, may be associated with a location to set an upper-bound on the delay. In Fig. 2, the invariant $c <= 10$ on *L1* indicates that A1 may delay at *L1* at most until c equals 10 time units. A guard, which is also a predicate of variables, may be associated with an edge as the required condition to take the transition. In this example, the guard $c > 5$ ensures that the transition from *L1* and *L2* can be taken only if the value of c is bigger than 5. During a transition, TA can take actions, as associated with the edge, to update the values of the variables.

The automaton A1 and A2 form a network of TA via parallel composition ("||"). The two TA can perform handshake synchronization via the channel *chan*. The "!" denotes the sender of the signal in the synchronization, and the "?" denotes receiver. In the example, A1 sends a signal via *chan* when it transits from *L1* to *L2*. Meanwhile, A2 receives the signal and takes the transition from location *L3* back to *L3*.

The requirements to be satisfied can be formalized in the UPPAAL query language, which is a decidable subset of TCTL (Timed Computational Tree Logic) [24]. These formalized requirements can then be verified exhaustively and automatically by the UPPAAL model checker. In this paper we will use the following queries:

– $A\ [\,]\ P$: P always holds for all possible execution paths (invariance property).

– $E <> P$: There exists a path in which P eventually holds (reachability property).

P is a logic expression that may contain clock constraints, and logical operators such as “and”, “or”, “not” and “imply”. In case an invariance property fails, UPPAAL can provide a trace leading to the violation as a counter-example of the property. If a reachability property is satisfied, UPPAAL also returns a trace that leads to the state where P holds. Such traces contain the state transitions as well as the bounds of clock variables in each state.

In this paper, we also use an extended version of UPPAAL TA, augmented with stopwatches [8]. Stopwatches allow clocks to be stopped at locations, so that the values of the clock variables do not progress. Fig. 3 shows an UPPAAL TA with stopwatch. The invariant $c2' == 0$ assigns the rate of the clock variable *c2* to be 0. By doing this, as long as automaton A2 stays at location *L3*, the value of *c2* is never changed.

UPPAAL has been used for generating test cases [19]. The system under test is modeled in UPPAAL TA, in which test inputs and outputs are encoded in the model. By executing UPPAAL queries that formalize the testing goal, testers can utilize the diagnostic traces returned by UPPAAL to form test sequences, which may consist of synchronizations, discreet transitions and time delays.

3 A Model-based Testing Methodology for Isolation Violation

In this section we present our proposed methodology for testing isolation violation in a distributed transaction system with lock-based CC. We first give an overview of our methodology, after which we explain the major steps and techniques in details.

3.1 Overview

We assume that a distributed transaction is a sequence of atomic operations that may access data located in different sites, and as a whole should satisfy data consistency requirements. Among them, the isolation requirements claim the avoidance of a particular set of interleavings of concurrency transactions. The transaction manager implements a CC algorithm with locking and unlocking mechanisms over the distributed data partitions. The delays between nodes, while their actual values unpredictable, are bounded with maximum and minimum values, which is reasonable for many distributed systems, such as automotive and factory automation systems. The maximum and minimum response times of the operations on each data item is also known a priori. While the system is tested, designated delays can be deliberately inserted between operations, and the interleavings can be examined by checking the transaction logs.

Our model-based testing methodology consists of four major steps, which are presented in Fig. 4, and listed as follows:

1. Specify the work units with bounded delays, the CC operations, as well as the isolation requirements, in a high-level description language.
2. Construct a network of UPPAAL timed automata for the work units as well as the concurrency control algorithm. Formalize the specified isolation properties in UPPAAL queries. Verify that the isolation properties are satisfied by the current design, using UPPAAL MC.

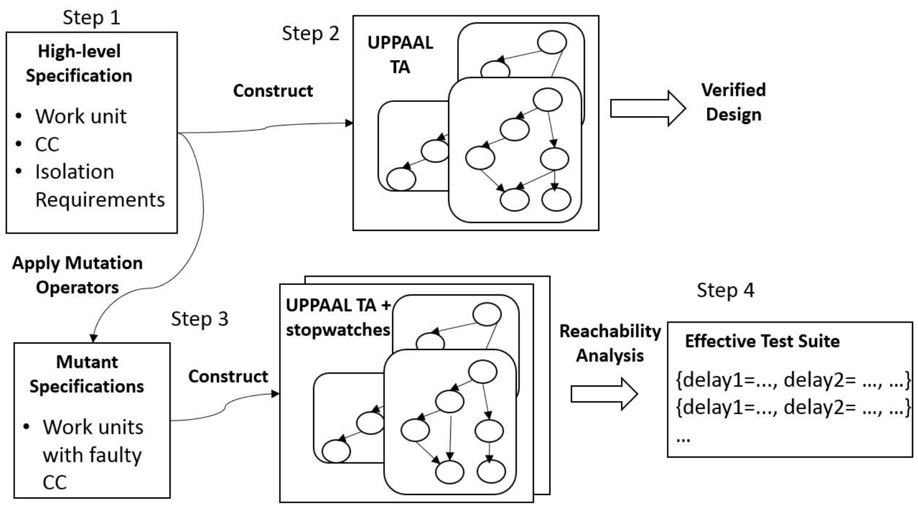

Fig. 4. Our proposed methodology

3. Mutate the transaction specification using the mutation operators presented in Section 3.4, and construct TA models with stopwatches for the mutants.
4. Generate an effective test suite from the mutant models.

The details of the steps are presented in the following subsections.

3.2 Step 1: Specification of Transactions and Isolation

Our methodology starts with the specification of the distributed transactions and the isolation properties. We assume a high level language is selected, which is able to specify the sequence of operations in the work unit of the transaction, the nodes where the data are located, the response times of the operations, as well as the lock and unlock operations. The language can also specify the delay bounds between operations, which is calculated by the designer based on the communication delays between nodes. For instance, Listing 1.1 and 1.2 specify transaction T0 and T1, respectively, in a generic exemplary high-level language. T0 updates configuration D1 and D2 on site S1 and S2, respectively. T1 reads configuration D2, and then modifies it after a calculation based on the read value. In this description language, "delay" specifies the bounded delay value between operations, with the lower and upper bounds in brackets. "Read" and "write" specify the data operations, while "calculate" specifies calculations in the client. The minimum and maximum response time of the operations are specified in the brackets as well. "On" specifies the sites. "Lock" and "unlock" specifies the CC operations. "Begin" and "commit" specify the boundary of the transaction, respectively.

The isolation property is specified as the avoidance of a set of unwanted interleavings. Such interleavings could be described as a sequence of operations, such as: T0 reads D2, T1 reads D2, T0 writes D2, T1 writes D2.

Listing 1.1. Specification of T0

```
begin
delay [1, 10]
lock D1 on S1
read D1 on S1 [1, 1]
delay [1, 10]
write D1 on S1 [1, 2]
delay [1, 8]
lock D2 on S2
read D2 on S2 [1, 1]
delay [1, 8]
write D2 on S2 [1, 2]
delay [1, 6]
unlock D1 on S1
unlock D2 on S2
commit
```

Listing 1.2. Specification of T1

```
begin
delay [1, 8]
lock D2 on S2
read D2 on S2 [1, 1]
delay [1, 8]
calculate [2, 3]
delay [1, 4]
write D2 on S2 [1, 2]
delay [1, 6]
unlock D2 on S2
commit
```

3.3 Step 2: Construction of TA Models for the Correct Specification

Our modeling framework for the distributed transaction system is adapted from an existing modeling framework for real-time concurrent transaction systems [7]. We extend the original model with the possibility to model distributed data partitions as well as delays between operations introduced by the distribution.

Assuming a distributed transaction system intended to achieve the isolation that avoids k unwanted interleavings, we model the system as a network of UPPAAL TA, denoted as N, defined as follows:

$$N ::= A_0 \,||\, ... \,||\, A_{n-1} \,||\, A_{CCManager} \,||\, O_0 \,||\, ... \,||\, O_{k-1},$$

where A_0, ..., A_{n-1} are the work units TA of transactions T_0, ..., T_{n-1}, $A_{CCManager}$ is the CCManager automaton, and O_0, ..., O_{k-1} are the TA of IsolationOservers that observe the unwanted interleavings, respectively. A work unit automaton models the work unit, that is, the operations of a transaction, as well as the delays between them. An IsolationObserver is an automaton that monitors the occurrence of transaction executions that lead to violation of the isolation property. The CCManager automaton models the selected lock-based concurrency control algorithm.

Fig. 5 shows the work unit automaton skeleton. The *begin* location represents that the transaction actually starts, while *commit_trans* represents the end of the transaction. After the start, the work unit performs a set of read, write or calculation operations, modeled by the instantiated operation patterns (Fig. 6). We extend the original pattern [7] for distributed transactions in two aspects. First, each read/write operation is performed atomically on a partition. This is modeled by a shared variable *cs[partition]* for each partition. When an operation is performed, it sets *cs[partition]* to 1, so that other operations on this partition are blocked until the current one has finished. The other extension is the modeling of delays between operations caused by the distributed communication. The time of delay for operation_k is modeled by the clock variable *Delay_k*. The operation may delay for at most *MAX_DELAY_k* time units, and must delay for at least *MIN_DELAY_k* time units. Each operation starts from the *start_operation* location, and moves to *operation*. Before moving to *operation_done*, it may stay at *operation* for at most *WCRT* time units, representing its worst-case response time; and for

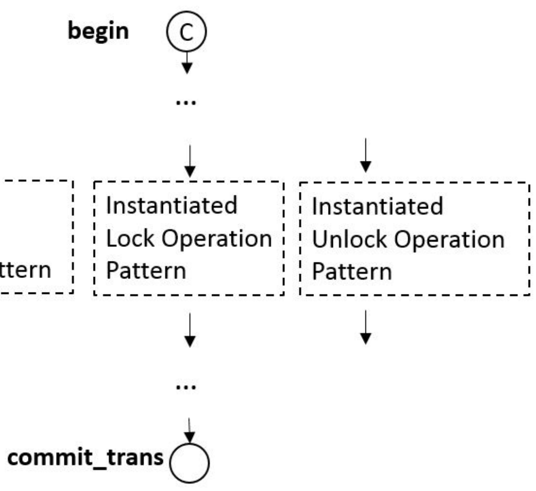
begin
C
...
Instantiated Read/Write Operation Pattern
Instantiated Lock Operation Pattern
Instantiated Unlock Operation Pattern
...
commit_trans

Fig. 5. The UPPAAL automaton skeleton of a work unit

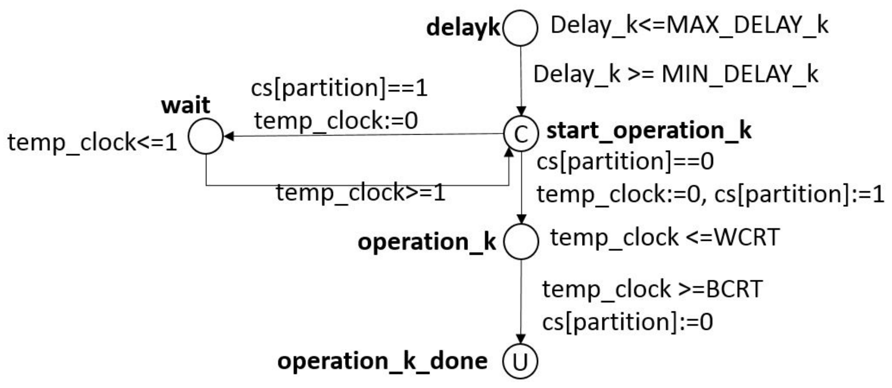

Fig. 6. Operation pattern extended with delay

at least *BCRT* time units, representing its best-case response time. The work unit may also interact with the CCManager in order to acquire and release locks, which is modeled by the instantiated locking and unlocking patterns (Fig. 7 and Fig. 8). The synchronization channel *lock[ti][di]* models the locking request for data D_j sent by transaction T_i. The channel *unlock[ti][di]* and *grant[ti][di]* represent the unlocking and granting messages, respectively. These patterns are similar to the ones in [7].

The CCManager is modeled using the CCManager skeleton proposed previously [7] (Fig. 9). It models the behavior of the concurrency control manager in managing lock requests and releases, following a selected algorithm. When a locking request is received via the *lock[ti][di]* channel, the CCManager evaluates the situation using a user-defined function *satisfyPolicy()*, and decides if it sends a grant signal, or refuses

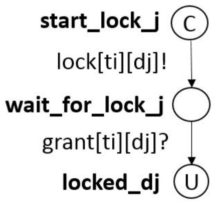

Fig. 7. Locking pattern

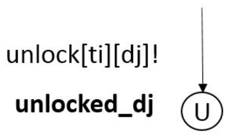

Fig. 8. Unlocking pattern

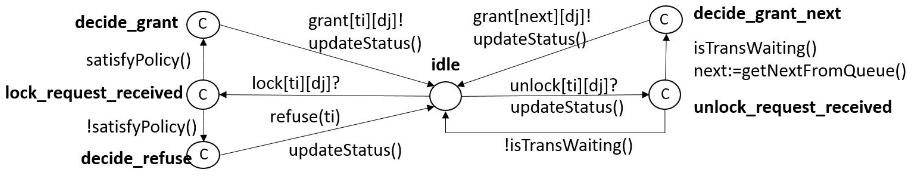

Fig. 9. The UPPAAL automaton skeleton of CCManager

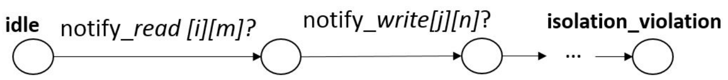

Fig. 10. The UPPAAL automaton skeleton of IsolationObserver

the request. When the CCManager receives a unlock signal, it picks the next transaction in the waiting queue (if any), and grants it the lock.

A violation of isolation can be seen as the occurrence of a series of inappropriate events, and is monitored by an IsolationObserver, which is modeled using the skeleton presented in Fig. 10. When the events occur in the specified order, the model eventually reaches the *isolation_violation* location.

As the violations of isolation are modeled using IsolationObservers, the verification of properties is equal to checking the *isolation_violation* locations are not reachable. This can be specified in UPPAAL query language and checked by the UPPAAL model checker. The following query specifies that the isolation property, that is, the violations encoded by the IsolationObservers will never occur:

$A[]\,(not\,IsolationObserver1.isolation\_violation)\,...$
$and\,(not\,IsolationObserverk.isolation\_violation)$.

The outcome of this step is a design of distributed transactions, including the work units and the CC manager, that are proven to satisfy the specified isolation properties.

3.4 Step 3: Generation of Mutant Specifications and Models

Inspired by the existing work on mutation testing for concurrency bugs [6, 17, 23, 31], which have proposed mutation operators that represent common CC faults in various implementation platforms and programming languages, we define a set of mutation operators for the high-level description language, listed in Table 1. Among them, "Remove

Table 1. Mutation operators for created specifications with CC faults

Mutation operator	Change in the correct specification
Remove lock	Remove a line with “lock” operation for a shared data
Add lock	Add a line with unnecessary “lock” operation for a data
Remove unlock	Remove a line with necessary “unlock” operation for a locked data
Change lock type	Change the type of lock
Change lock position	Move the line with a lock operation to another position
Change unlock position	Move the line with an unlock operation to another position

lock”, “Add lock” and “Remove unlock” models the common errors that developers may forget to lock/unlock data, or put on unnecessary locks. “Change lock type” is a useful operator for CC algorithms applying more than one type of locks. “Change lock position” and “Change unlock position” adjust the duration when the data are locked, which captures the error of erroneous length of critical sections.

As examples, Listings 1.3, 1.4 and 1.5 present three mutant specifications, respectively. Mutant 1 applies the “Remove lock” operator, as shown in Listing 1.3, which removes the lock before reading D2 in T0. Mutant 2 applies the “Change unlock position” operator, moves the unlocking of D2 in T0 to before writing D2 (Listing 1.4). We apply “Change lock position” to create mutant 3, which moves the locking of D2 in T1 to before the writing of D2, but after the calculation (Listing 1.5).

Listing 1.3. Mutant 1

```
begin
delay [1, 10]
lock D1 on S1
read D1 on S1 [1, 1]
delay [1, 10]
write D1 on S1 [1, 2]
delay [1, 8]
--Remove lock
//lock D2 on S2
read D2 on S2 [1, 1]
delay [1, 8]
write D2 on S2 [1, 2]
delay [1, 6]
unlock D1 on S1
unlock D2 on S2
commit
```

Listing 1.4. Mutant 2

```
begin
delay [1, 10]
lock D1 on S1
read D1 on S1 [1, 1]
delay [1, 10]
write D1 on S1 [1, 2]
delay [1, 8]
lock D2 on S2
read D2 on S2 [1, 1]
delay [1, 8]
--Change unlock position
unlock D1 on S1
write D2 on S2 [1, 2]
delay [1, 6]
--Change unlock position
//unlock D1 on S1
unlock D2 on S2
commit
```

Listing 1.5. Mutant 3

```
begin
delay [1, 8]
--Change lock position
//lock D2 on S2
read D2 on S2 [1, 1]
delay [1, 8]
calculate [2, 3]
delay [1, 4]
--Change lock position
lock D2 on S2
write D2 on S2 [1, 2]
delay [1, 6]
unlock D2 on S2
commit
```

The TA models of the mutant specifications are constructed using the same technique as presented in Section 3.3. However, in order to generate test cases with delays, the actual delay values need to be captured in the trace. We achieve this by adapting the models with stopwatches, as shown in Fig. 11. The rule is that, for each *delayk* location, only the kth clock is allowed to progress, while all other clock variables *Delay_n* ($n \neq k$) is set to stop, which is done by *Delay_n'==0* in the invariants of this location. In other locations, all clocks are set to stop using stopwatches in the same way. The purpose is that, a clock variable for delay only progresses during its delay period, so

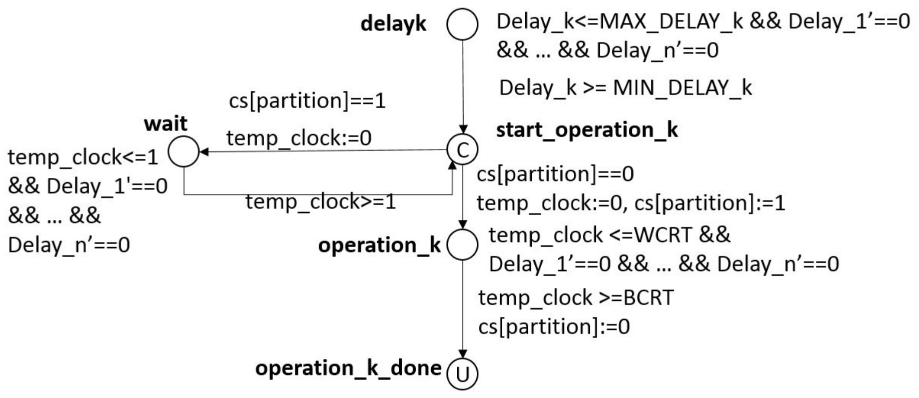

Fig. 11. Modified operation pattern for mutant models

that the delay values are preserved in the diagnostic trace. The output of this step is a set of mutant models, each representing a mutant specification with CC faults.

3.5 Step 4: Generation of Effective Test Suite

For each mutant model, we use UPPAAL tool to verify the following query:

$E <> IsolationObserver.isolation\_violation.$

UPPAAL then generates a trace that leads to the location *isolation_violation*, if this location is reachable. At the end of the trace, the clock constraints at location *isolation_violation* encode the bounds of values of the clock variables, which represent the feasible ranges of delays that can lead to the violation. Since the clock constraints are simple linear inequalities between variables and integers, it is easy to solve them using modern tools, such as Microsoft Z3 SAT solver [11].

We propose an algorithm to generate the effective test suite. If we use CS_i to denote the set of clock constraints of mutant M_i, the delay values as the solution set of CS_i are then the test inputs that can lead to the violation, therefore killing M_i. Ideally, the effective test suite contains only one test case, which is the set of delay values that satisfy $CS_1 \wedge ... \wedge CS_k$. Therefore, our algorithm starts with looking for one solution for $CS_1 \wedge ... \wedge CS_k$ (line 3, when *i* equals 1). If such a solution cannot be found, we try to find a larger test suite, by dividing the constraint set into smaller groups, and finding a solution for each group. For instance, let us consider that *i* equals 2, meaning that we try to find a solution containing two sets of delay values, one set satisfying $CS_1 \wedge ... \wedge CS_m$, the other satisfying $CS_{m+1} \wedge ... \wedge CS_k$. From line 12 to line 23, the algorithm iterates all possible two-group partitions, and tries to find a solution set for each. For each group in the partition, the algorithm tries to solve the constraints using Z3 (line 15), and records the solution (line 17). If every group in the current two-group partition is solvable, the algorithm returns the solutions as the minimum set of delays that forms the test suite (line 24, in this case the suite contains exactly two test cases). If any group is found unsolvable, the algorithm skips the current two-group partition, and tries another one (line 20). If no two-group partition is solvable, which means there does not exist a two-case test suite, the algorithm continues with a possible three-group partition (increment i in line 3), trying to find a test suite that contains three test cases. Using this algorithm, the effective test suite will have at most m test cases, in which each for one of the mutants; and at least 1 test case, which is able to kill all mutants.

Algorithm 1 Algorithm to find the effective test suite

```
1:  function $find\_effective\_suite()$:
2:  The constraint set of $m$ mutants $CS := \{CS_1, ..., CS_m\}$
3:  for $i \in [1, m]$ do
4:      SOLUTIONS := $solve\_k\_partition\_of\_CS(i)$
5:      if SOLUTIONS $\neq$ NULL then
6:          return SOLUTIONS
7:      end if
8:  end for
9:  return NULL
10:
11: function $solve\_k\_partition\_of\_CS(k)$:
12: for each k-partition $S_k$ of $CS$ do
13:     SOLUTIONS := { }, solved := true
14:     for each constraints group $S_k^i$ in $S_k$ do
15:         try solving $S_k^i$ using Z3
16:         if $S_k^i$ is solvable and solution $SOLUTION_i$ is found then
17:             SOLUTION := SOLUTION $\cup$ $SOLUTION_i$
18:         else
19:             solved := false
20:             break
21:         end if
22:     end for
23:     if solved == true then
24:         return SOLUTIONS
25:     end if
26: end for
27: return NULL
```

4 Illustrative Example

To illustrate our proposed methodology, we present an running example, based on the aforementioned transactions in Listings 1.1 and 1.2 in an unmanned loading system on a construction site [22]. This system consists of an autonomous wheel loader, an autonomous dump truck, and a controller, each equipped with a computer, and connected by real-time industrial wireless communication. In a typical scenario, the controller configures the planned job information (paths, locations, etc.) for the truck and wheel loader, which are stored in their respective databases. The wheel loader autonomously discovers and scoops the piles in its surroundings, and dumps them into the truck. The wheel loader may also adjust its working path, in order to avoid obstacles in its working surroundings. All data are protected by CC that requires the corresponding locks in order to access the data.

In this scenario, we consider two transactions. Started by the controller, transaction T0 updates the next location of the truck (denoted as D1) in the truck's local system (denoted as S1), followed by updating the planned location and path (D2) in the wheel loader (S2). Transaction T1, started by the wheel loader, reads D2 from S2, calculates a new value, and then modifies D2. The isolation requirement demands the avoidance of the following sequence: T0 reads D2, T1 reads D2, T0 writes D2, T1 writes D2, which indicates that, the wheel loader using old data computes a new path, which overwrites the planned path. Consequently, the truck goes to the newly planned location while the wheel loader may goes to the old one and miss the truck. We apply our proposed methodology to generate a test suite for testing the CC faults that can lead to the violation of isolation.

4.1 Specification of transactions and isolation

The first step is to specify the transactions and the isolation requirement. T0 and T1 are specified in Listing 1.1 and 1.2, respectively. The isolation requirement is formulated as follows: the following sequence of operations, T0 reads D2, T1 reads D2, T0 writes D2, T1 writes D2, should never occur.

4.2 Construction of TA Models for the Correct Specification

We model the transactions and CC in UPPAAL timed automata. The automaton of T_0, as an example, is presented in Fig. 12. It is constructed by composing the instantiated operation, locking and unlocking patterns with the work unit skeletons. An IsolationObserver is created to monitor the occurrence of the unwanted interleavings, as presented in Fig. 13. Isolation is verified by UPPAAL using the following query:

$A[]\ (not\, IsolationObserver.violation)$.

The time and memory consumed by UPPAAL for the verification are 0.015s and 28316KB, respectively.

4.3 Generation of Mutant Specifications and Models

We use the mutation operators proposed in Table 1 to manually generate mutant models. In this case study, we create three mutants, which are specified as Listings 1.3, 1.4 and 1.5 in Section 3.4. The TA models of these mutant specifications are also constructed.

4.4 Generation of Effective Test Suite

We use the query in Section 3.5 to generate diagnostic traces for the mutant models, and obtain the constraints on the clock variables representing the delays. The time and memory consumed for the checking of each mutant, respectively, are as follows: M1 (0.014s and 28488KB), M2 (0.007s and 28388KB), M3 (0.015s and 28492KB). The

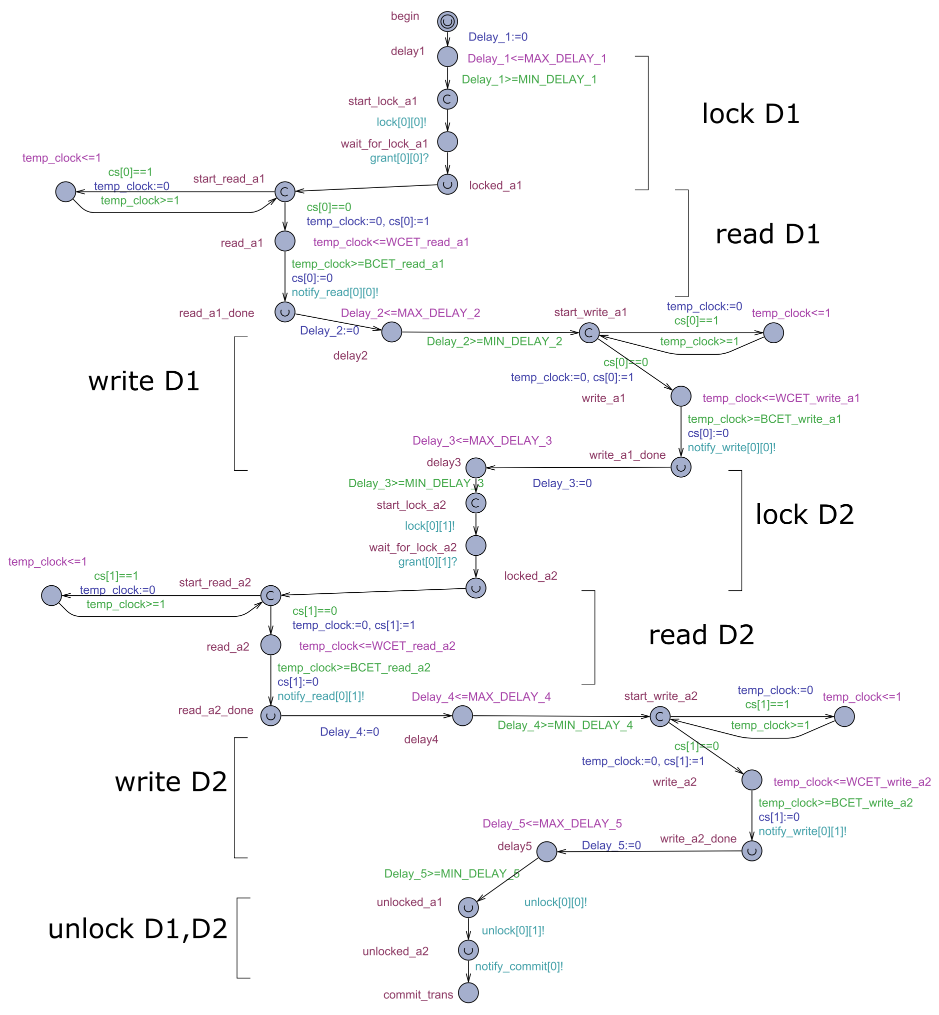

Fig. 12. The UPPAAL model of T_0

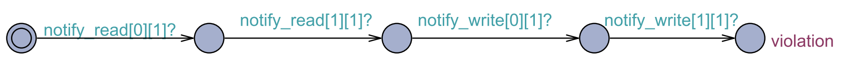

Fig. 13. The UPPAAL model of IsolationObserver that monitors the unwanted interleavings

constraint sets for mutants M1, M2 and M3 are listed in Listings 1.6, 1.7, and 1.8, respectively.

Table 2. Test inputs of the test case in the effective test suite

Test input variable	Value
T0.Delay_1	3
T0.Delay_2	3
T0.Delay_3	1
T0.Delay_4	1
T0.Delay_5	1
T1.Delay_1	10
T1.Delay_2	1
T1.Delay_3	1
T1.Delay_4	1

Listing 1.6. CS_1

```
3≤T0.Delay_1≤20
3≤T0.Delay_2≤15
1≤T0.Delay_3≤8
1≤T0.Delay_4≤8
1≤T0.Delay_5≤6
10≤T1.Delay_1≤20
1≤T1.Delay_2≤4
1≤T1.Delay_3≤4
1≤T1.Delay_4≤4
T0.Delay_1-T1.Delay_1≥-17
T0.Delay_1-T1.Delay_1≤0
T1.Delay_1-T0.Delay_2≥4
T1.Delay_1-T0.Delay_2≤17
T1.Delay_1-T0.Delay_3≥8
T1.Delay_1-T0.Delay_3≤19
T1.Delay_3-T0.Delay_5≥-5
T1.Delay_3-T0.Delay_5≤2
```

Listing 1.7. CS_2

```
3≤T0.Delay_1≤20
3≤T0.Delay_2≤6
1≤T0.Delay_3≤2
1≤T0.Delay_4≤5
1≤T0.Delay_5≤6
T1.Delay_1==10
1≤T1.Delay_2≤4
1≤T1.Delay_3≤4
1≤T1.Delay_4≤4
T0.Delay_4-T1.Delay_2≥-3
T0.Delay_4-T1.Delay_2≤1
T0.Delay_5-T1.Delay_3≥-3
T0.Delay_5-T1.Delay_3≤3
```

Listing 1.8. CS_3

```
3≤T0.Delay_1≤20
3≤T0.Delay_2≤15
1≤T0.Delay_3≤8
1≤T0.Delay_4≤8
1≤T0.Delay_5≤6
10≤T1.Delay_1≤20
1≤T1.Delay_2≤4
1≤T1.Delay_3≤4
1≤T1.Delay_4≤4
T0.Delay_1-T1.Delay_1≥-17
T0.Delay_1-T1.Delay_1≤0
T1.Delay_1-T0.Delay_2≥4
T1.Delay_1-T0.Delay_2≤17
T1.Delay_1-T0.Delay_3≥8
T1.Delay_1-T0.Delay_3≤19
T0.Delay_5-T1.Delay_3≥0
T0.Delay_5-T1.Delay_3≤3
```

Algorithm 1 is applied to find the effective test suite. The combined constraint set $CS_1 \wedge CS_2 \wedge CS_3$ is solvable using Z3, which returns the solution, which is the designated delays as inputs of one test case that kills all mutants [1]. Therefore, the effective test suite contains one test case, as shown in Table 2.

4.5 Discussion

Our methodology generates a set of delays between operations, which can be utilized to create concrete test cases for a particular implementation, considering its architecture and testability characteristics. For instance, if the tested system is instrumented with configurable delayed operations on each individual site and synchronized clocks across all sites, a concrete test sequence with designated delays from the abstract test case can be achieved.

Scalability is one important issue for the verification of distributed transactions, which may involve data items over many partitions. As a result, the model may in-

[1] We used the online Z3 tool provided by Microsoft (https://rise4fun.com/z3) in this running example. The time to resolve the constraints was less than one second.

clude large numbers of operations, delays and synchronization signals that lead to state explosion for exact model checking. Another characteristic of distributed system is the probabilistic behavior, for which we may only know the probabilistic distribution, rather than exact bounds of the delays in the system. In both cases, we need to adapt the modeling language from UPPAAL TA to UPPAAL stochastic timed automata, in which the probability of delays can be encoded and analyzed by statistical model checking using UPPAAL SMC [10]. More scalable than exact model checking, statistical model checking can estimate the probability of the violation of isolation with a given confidence level, via stochastic simulation. Values of designated delays can be generated automatically using statistical simulation provided by UPPAAL SMC.

In this example, we manually generate UPPAAL models for the distributed transaction system specification, as well as its three mutants by applying three mutant operations directly. In practice, induced from domain knowledge in concurrency faults and distributed transaction systems, selected mutation operators can be combined and applied multiple times on a transaction. For instance, the "Change lock type" operator may be applied together with the "Change lock position" operator in one or many locking operations in a transaction. This may result in a large number of mutants and requires considerable modeling effort. Therefore, tool automation is important for the application of our methodology in practice, which should assist the testers to specify the mutated specifications easily, and generate UPPAAL models automatically. We consider it feasible to automate the generation of models by a tool, thanks to the modularization of our modeling framework. The tool should also automate the extraction of clock constraints from the diagnostic traces, as well as the resolution of the constraints with Z3, which is also possible. Tool support for our methodology is considered as our future work.

5 Related Work

Testing for concurrency design faults in database systems, and in software systems in general, has attracted considerable attention in recent years. Much of the effort has been dedicated in detecting the interleavings that could lead to consistency violations, and executing such interleavings in a controlled way. Deng et al. [12] have proposed techniques for testing isolation violations of concurrent transactions in database systems. They propose to use dataflow analysis techniques to identify the schedules, which are ordered operations, that can lead to violations. These schedules are then executed as test cases by the target database system. The delays between operations are neglected in this work. In our work, on the contrary, we generate not only the schedules, but also the delays between the operations, which are an important factor for isolation in a distributed transaction system. We also propose to use mutation testing to find an effective test suite. Park et al have proposed the CTrigger method for testing atomicity violation bugs [30]. In their work, unserializable interleavings that lead to concurrency bugs are identified by a set of profiling runs of the target program. Such interleavings are ranked by their probabilities of occurrence, used as a selection metric for pruning the test suite. Similar to CTrigger, the CHESS tool proposed by Musuvathi et al [28] also identify race conditions by a small number of runs. In the race-directed random testing (RACE-

FUZZER) framework proposed by Sen [32], the author has applied dynamic data race detection algorithms [29] to compute the interleavings with potential data races, and used a scheduler to execute and evaluate these interleavings at run time. Blum and Gibson [5] have proposed a stateless model checking framework called QUICKSAND to examine the interleavings at runtime. Contrary to these work, our framework proposes the identification and selection of delays, which is a common issue for testing distributed system, that can lead to unwanted interleavings. Our methodology is based on abstract models of the system, which can be automatically processed by existing model checking tools.

Mutation testing has been recognized as a promising technique for pruning test cases for concurrent systems. Deniz et al. [13] have proposed a mutation library for Multicore Communication API (MCAPI), targeting multicore embedded systems. Mutation operators are introduced for concurrent communication messages. Gligoric et al. [17] have created mutants for multithreaded code in their MutMut tool. Mutation operators have been proposed for various design and implementation languages, such as SystemC [31], C/C++ [23] and JAVA [6]. Our work shares the same objective and concepts of mutation testing with these mentioned work, and gets inspired by the existing classification of concurrency control errors. However, as we focus on the mutation at the model level, our mutation operators are more abstract, rather than concrete operations in specific programming languages as in most of the existing work,.

6 Conclusions and Future Work

In this paper, we proposed a model-based testing methodology for designing test cases for concurrency control faults in distributed transaction systems. The test inputs are delays between distributed operations, while the outputs are the occurrence of isolation violations, which are consequences of CC faults. We model the transaction system, as well as the isolation violations, in UPPAAL TA. We create a set of mutated models that model the common CC faults, based on which we generate test cases via reachability analysis. A minimal set of test cases that can kill most mutants is generated, which forms the effective test suite.

One of our future work is to develop tool automation for our methodology, including automated mutation and model generation, and test case generation and selection. We also consider to investigate the application of statistical model checking, in order to improve the scalability and deal with probabilistic behaviors of distributed systems. The extension of the methodology for other types of concurrency bugs, such as deadlock, and other properties of distributed transactions, such as atomicity, is another interesting direction.

Acknowledgment The Swedish Research Council (VR) is gratefully acknowledged for supporting this research by the project “Adequacy-based Testing of Extra-Functional Properties of Embedded Systems”.

References

1. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: Proceedings of the 16th ICDE. pp. 67–78 (2000)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science 126(2), 183–235 (1994)
3. Bartholomew, R., Collins, R.: Using combinatorial testing to reduce software rework. CrossTalk 23, 23–26 (2014)
4. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Computing Surveys (CSUR) 13(2), 185–221 (1981)
5. Blum, B., Gibson, G.: Stateless model checking with data-race preemption points. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 477–493. ACM (2016)
6. Bradbury, J.S., Cordy, J.R., Dingel, J.: Mutation operators for concurrent java (j2se 5.0). In: Mutation Analysis, 2006. Second Workshop on. pp. 11–11. IEEE (2006)
7. Cai, S., Gallina, B., Nyström, D., Seceleanu, C.: A formal approach for flexible modeling and analysis of transaction timeliness and isolation. In: Proceedings of the 24th International Conference on Real-Time Networks and Systems. pp. 3–12. ACM (2016)
8. Cassez, F., Larsen, K.: The impressive power of stopwatches. In: International Conference on Concurrency Theory. pp. 138–152. Springer (2000)
9. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2), 244–263 (1986)
10. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal smc tutorial. International Journal on Software Tools for Technology Transfer 17(4), 397–415 (2015)
11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
12. Deng, Y., Frankl, P., Chen, Z.: Testing database transaction concurrency. In: Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. pp. 184–193. IEEE (2003)
13. Deniz, E., Sen, A., Holt, J.: Verification coverage of embedded multicore applications. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012. pp. 252–255. IEEE (2012)
14. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. IBM systems journal 41(1), 111–125 (2002)
15. Elmasri, R.A., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley Longman Publishing Co., Inc. (2004)
16. Fu, H., Wang, Z., Chen, X., Fan, X.: A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. Software Quality Journal pp. 1–35 (2017)
17. Gligoric, M., Jagannath, V., Marinov, D.: Mutmut: Efficient exploration for mutation testing of multithreaded code. In: Software testing, verification and validation (ICST), 2010 third international conference on. pp. 55–64. IEEE (2010)
18. Grottke, M., Trivedi, K.S.: A classification of software faults. Journal of Reliability Engineering Association of Japan 27(7), 425–438 (2005)
19. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In: Formal methods and testing, pp. 77–117. Springer (2008)
20. ISO/IEC 9075:1992 Database Language SQL. Standard, International Organization for Standardization

21. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE transactions on software engineering 37(5), 649–678 (2011)
22. Koyachi, N., Sarata, S.: Unmanned loading operation by autonomous wheel loader. In: ICCAS-SICE, 2009. pp. 2221–2225. IEEE (2009)
23. Kusano, M., Wang, C.: Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. pp. 722–725. IEEE Press (2013)
24. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
25. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM Sigplan Notices. vol. 43, pp. 329–339. ACM (2008)
26. Marinescu, R., Seceleanu, C., Le Guen, H., Pettersson, P.: A research overview of tool-supported model-based testing of requirements-based designs. Advances in Computers 98, 89–140 (2015)
27. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: ACM Sigplan Notices. vol. 42, pp. 446–455. ACM (2007)
28. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI. vol. 8, pp. 267–280 (2008)
29. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Acm Sigplan Notices. vol. 38, pp. 167–178. ACM (2003)
30. Park, S., Lu, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: ACM SIGARCH Computer Architecture News. vol. 37, pp. 25–36. ACM (2009)
31. Sen, A., Abadir, M.S.: Coverage metrics for verification of concurrent systemc designs using mutation testing. In: High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International. pp. 75–81. IEEE (2010)
32. Sen, K.: Race directed random testing of concurrent programs. ACM SIGPLAN Notices 43(6), 11–21 (2008)
33. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability 22(5), 297–312 (2012)