Formal Verification of Python Software Transactional Memory Based on Timed Automata

Branislav Kordic, Miroslav Popovic, Silvia Ghilezan

University of Novi Sad, Faculty of Technical Sciences
Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia

branislav.kordic@rt-rk.uns.ac.rs, miroslav.popovic@rt-rk.uns.ac.rs, gsilvia@uns.ac.rs

Abstract: Nowadays Software Transactional Memories (STMs) are used in safety-critical software, such as computational-chemistry simulation programs. To the best of our knowledge, the existing STMs were not developed using rigorous model-driven development process, on the contrary, the majority of proposed STMs are directly implemented in a target programming language and formally verified STMs are proven against more general models. This may result in some key aspects of implementation being omitted or interpreted incorrectly. In this paper, we demonstrate an approach to the formal verification of one particular STM, for the Python language, named Python Software Transactional Memory (PSTM), which is based on a STM design and implementation details. Based on these details, faithful models of a PSTM based system, are developed and verified. The PSTM system components are modeled as timed automata utilizing UPPAAL tool. Finally, it is verified that PSTM satisfies deadlock-freeness, safety, liveness, and reachability properties.

Keywords: formal verification; transactional memory; model checking; correctness, timed automata

1 Introduction

Transactional Memory (TM) is a programing paradigm [1, 2] which offers an alternative to traditional lock mechanisms based on mutual exclusion by replacing them with lock-free mechanism in order to harvest more performances on multicore architectures. It is considered to be a paradigm that simplifies writing and maintaining parallel programs as well. Due to the lack of hardware support Software Transactional Memory (STM) was born [3]. For a long time, STMs have been a playground for research in this area. Even today, it seems that hardware support is still not a standard feature in commercial architectures.

The correctness of a transactional memory plays a key role in a transactional based system. The common properties and correctness criteria with small variations of basic ideas [4, 5, 6, 7, 8] such as serializability, atomicity, deadlock etc., can be defined. Serializability and opacity are assumed as prevailing correctness criteria for the safety property, whereas different levels of progressiveness are commonly used for the liveness property. Predominantly, (S)TM verifications are applied on an abstract model which is drawn from specification and/or captured from transactional semantics rather than being developed directly from an implementation itself – a design and a source code details may be omitted despite the fact that a verification model is desired to be a faithful counterpart of the verified system. On the other hand, most of the formal verification models and approaches targeting STMs are general. They were created with the intention to be used as general frameworks, and not to target real implementations. In this paper we tried to overcome these shortcomings by using an approach that could be applied in an agile software development and which uses existing STM's program code as its input. In our previous work [9] we tackle this problem and presented preliminary verification results.

A motivating example which initially inspired us to search for a Python STM solution and to verify its correctness is the performance optimization of a Python application in the area of chemical and pharmacy calculation [10, 11]. The authors of these papers describe a computational-chemistry simulation program for the Protein Structure Prediction model. The aim of PSTM is to replace the existing barrier-based process synchronization in order to gain more performances. Although Python is one of the most widely used programming languages, it still lacks an applicable and reliable STM implementation. Some announcements for PyPy have been made [12], but until today, no final solution has been published.

In this paper, a formal verification of Python Software Transactional Memory (PSTM) [13] using UPPAAL tool [14] is presented. The main aims are (1) to apply a formal verification process to a real STM solution in order to derive a faithful STM model based on a particular PSTM design and implementation rather than making a general model, and (2) to use the developed PSTM model for automated machine-checked formal verification of core system properties which ensures PSTM correctness, namely deadlock-freeness, safety, liveness, and reachability properties. In the contrast to general models, fine grained parameterized automata models are developed. As a type of transactions, aligned and drifted (time shifted) read-write transactions which share a common variable are considered. For the verification purpose, a formula for calculating commit time for a given arbitrary number of transactions was derived.

This paper contributes to the related aspects of STM formal verification, in the following areas: (i) to the best of our knowledge, this is the first formal verification of an STM solution for Python language, (ii) it introduces an approach to modeling a real STM implementation by a tool based on a timed finite state machine model rather than modeling a high-level STM abstraction model, (iii) the

templates of fine grained automata models can be used as a starting point for verification of any PSTM based system, (iv) it develops a framework for calculating transactions execution times as a means for verifying system temporal behavior, and lastly, (v) it formally verifies that the STM solution for Python language, namely PSTM, conforms deadlock-freeness, safety, liveness, and reachability properties, and hence it is eligible to be a part of a real-world application.

The paper is organized as follows. In Section 2 PSTM architecture is introduced. Formalization of PSTM using UPPAAL tool is presented in Section 3. In Section 4 a framework for temporal behavior analysis is introduced. Verification properties and results are provided in Section 5 and Section 6, respectively.

2 Python Software Transactional Memory

In general, a transactional memory system accommodates transactions and a component responsible to handle transactions requests, i.e. a (S)TM. In this paper, that component is PSTM [13]. A transaction may be considered as a sequence of instructions that are executed atomically over a given set of transactional variables (*t-vars*). The common behavior steps for a transaction are the following: (i) get a local set of t-vars from PSTM, also called a *snapshot*, (ii) perform a processing based on t-vars, and (iii) commit new values, if any.

PSTM architecture is shown in Fig. 1. It consists of the two main components, a *set of transactions* and *PSTM*. Transactions are executed in the context of a transactional application while the PSTM is comprised of an API provided to the transactions, and a server which implements the API functionality.

PSTM public API captures all requirements defined by the common transactions behavior. The API functions are accessible via Remote Procedure Call (RPC) interface. The RPC interface is a key part of PSTM which provides concurrent access to PSTM – it ensures transactional requests serialization. The RPC interface is implemented using Python Queue class. Transactions use a singleton queue to send requests towards PSTM. PSTM API is the following:

- `AddVars(q, keys)`
- `GetVars(q, keys)`
- `CommitVars(q, rw_sets)`
- `PutVars(q, vars)`
- `CmpVars(q, vars)`

Let us first introduce a *dictionary* and a *t-var*. A transactional variable, or t-var, denotes a variable stored in PSTM which can be accessed only through the API

functions. A t-var is uniquely determined by three attributes, namely a *key*, a *version*, and a *value*. A t-var's *key* is an accessing identifier of a t-var, a *version* holds the t-var's version, which is the most recent one at the read time, and a *value* represents data. The t-vars are kept in a dictionary. Only the PSTM server is allowed to *read* and *write* directly to the dictionary. The dictionary and t-vars are implemented as Python dictionary and tuple data structures, respectively.

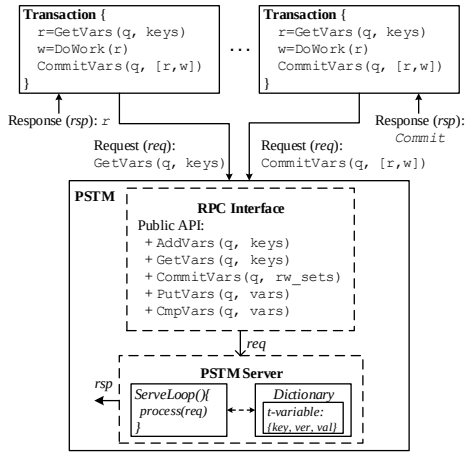

Figure 1
Overview of PSTM architecture

PSTM API functions share a mutual argument queue `q`. The queue is used as a communication channel between transactions and PSTM. The function `AddVars` introduces new t-vars to PSTM. The function `GetVars` returns the most recent version of t-vars which are currently stored in the dictionary. For both `AddVars` and `GetVars` functions a set of t-vars' keys are expected as `keys` argument. The function `CmpVars` is a helper function used to compare (or validate) a set of t-vars' versions against the current versions of corresponding t-vars in the dictionary. The argument `vars` denotes a set of t-vars with all attributes included. The function `CommitVars` tries to commit a transaction to PSTM, i.e. tries to write (update) new value to a t-var. The function `CommitVars` takes the two sets of t-vars, a *read set* and a *write set*, as the argument `rw_sets`. The read set comprises of t-vars that were previously read, i.e. a local snapshot of transactions, while the write set carries changes (a set of t-vars values) which have to be applied to the t-vars in the dictionary. The function commits only if all t-vars' versions in the local snapshot are equal to the t-vars' versions in the dictionary, which means that the particular transaction has the most recent versions of t-vars. When a transaction successfully commits, a t-var version is changed. The function `PutVars` gives another way to commit. Its attempt will be successful only if all the t-vars are up to date. Usually, it is used for t-vars initialization. For the context of this paper `GetVars` and `CommitVars` functions are the most important.

A PSTM server is used to process transactions requests. It provides functionalities behind PSTM API. The PSTM server takes a request from the queue, executes it, and sends a response message to a client (i.e. a transaction). The backbone of PSTM architecture is based on the conventional client-server architecture and it relies on multipoint-to-point and point-to-point communication. Transactional requests are sent in multipoint-to-point fashion, while a request response is sent from the PSTM server directly to a transaction.

In a PSTM execution model, a transaction starts with a *read operation*, then follows a *processing operation*, and finally, the transaction ends with a *write operation*. The mapping between PSTM API functions and transactional operations is illustrated in Fig. 2. The functions `GetVars` and `CommitVars` correspond to the operations read and write, respectively. The function `DoWork` is the processing operation and it is not a member of PSTM API.

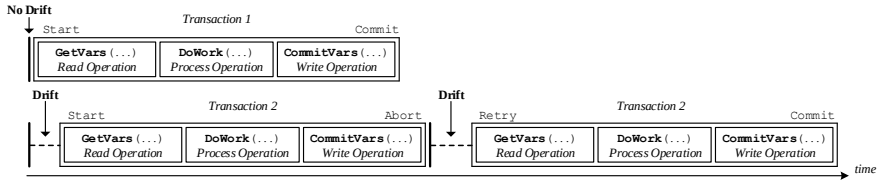

Figure 2

An example execution of aligned (*Transaction 1*) and drifted (*Transaction 2*) transactions

Let's suppose that in the example (Fig. 2) both transactions share a common t-var. The start time of the second transaction is drifted (time shifted) to the start time of the first transaction. Because of the conflict only one of them commits. Specifically, the first transaction commits while the second aborts and retires. An execution set can be comprised of *aligned* (not shifted) and *drifted* transactions. Within a set of aligned transactions, all the transactions start at the same time and they retry immediately after an abort, while within a set of drifted transactions, all the transactions may start and retry in a non-deterministic fashion.

3 PSTM Formalization

In this section we describe PSTM modeling approach, sketch up building entities of a UPPAAL PSTM system model, and introduce its timed automata models.

3.1 Modeling Approach

Due to its expressiveness and convenience, finite state machine based formalisms such as Petri nets and Timed Automata (TA), are often used in the praxis for

modeling and machine verification of complex problems like scheduling [15] or synchronization problems [16]. PSTM is formalized and verified using UPPAAL tool. The UPPAAL tool [17, 18, 14] is based on TA and it is widely used by the researchers and in the industry too. In addition to the expressiveness inherited from TA, it provides powerful and user-friendly model checker tool.

A model of a PSTM verification system is made in a compositional way from a set of nondeterministic finite state automata which are coupled through communication channels and shared variables. Three major automata are:

- *Transaction*
- *Remote Procedure Call Queue*
- *Transactional Memory*

A UPPAAL PSTM system design is shown in Fig. 3. The automata are depicted as functional blocks interconnected with channels. The automata are implemented as a template with local variables and functions. The channels and template attributes are defined relying on UPPAAL native data types and C-like features.

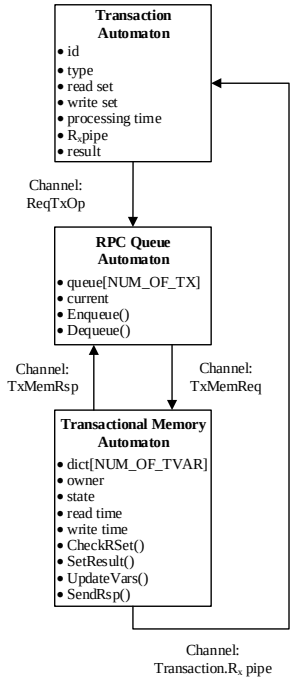

Figure 3
UPPAAL PSTM system design

3.2 UPPAAL PSTM Model

The automata *Transaction*, *RPC Queue*, and *Transactional Memory* are modeled as the templates `Transaction`, `RPCQueue`, and `TxMemory`, respectively.

3.2.1 The Automaton Transaction

An instance of the automaton `Transaction` (Fig. 4) starts with a transition from the state `go` to the state `start_tx`. Immediately after the start, the transaction instance requests a set of shared t-vars from the transactional memory.

The read request is sent during the transition from the state `start_tx` to the state `wait_rsp_TX_R` issuing a send operation to the channel *ReqTxOp*. As a part of the channel operation, transaction data (or context data) are updated: First an operation type is set (TX_R), secondly the shared t-var *SharedTvar* is defined, and finally, the data are passed to the `RPCQueue` through a global variable *RPCQueueMsg*. The *tx_id* is defined as an input argument of each `Transaction` instance. When the request in sent, the instance moves to the state `wait_rsp_TX_R` and waits until the read operation is processed.

The corresponding response is received during the transition from the state `wait_rsp_TX_R` to the state `update_tvar`. It is received by issuing a receive operation on the channel *TxPipe*[*tx_id*]. As in the case of a sending request, the response data are sent through a global variable *TxMemRspMsg*.

In the state `update_tvar` the transaction's read set *Tx*[*tx_id*].*read* contains the most recent version of the shared t-var. The transaction processing time is modeled using a time invariant. The time invariant associated to the state `update_tvar` defines the transaction processing duration. It ensures that the automaton holds in the state `update_tvar` exactly *Tx_proc* time units. For that purpose, a clock variable *c* is introduced. The clock variable *c* measures the time progress and it is local for each transaction instance. Using the time invariant and the clock variable *c* the channel *ReqTxOp* is invoked only when *c* is equal to *Tx_proc*. The variable *Tr_proc*'s value is defined as a template's input argument.

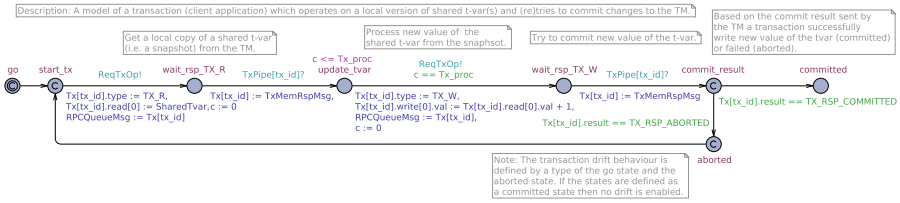

Figure 4
UPPAAL model of a transaction, i.e. the automaton `Transaction`

Each transaction process the shared t-var in the same way – it increments the t-var's current data value. Once the shared t-var is processed, it is ready for a commit. The commit operation is performed in the same two-steps as the read operation, except that the write operation is applied. When the corresponding request is ready, the transaction gets the response, and moves to the state `commit_result`. In that state, a result of the commit operation is known, so the transaction finally ends in `committed` or `aborted` state.

For the verification purposes, the two variations of the automaton `Transaction` are defined. The automaton depicted in Fig. 4 is also named *cyclic* transaction. After the abort, the cyclic transaction models the *retry* operation. A transaction model which does not retry is named a *linear* transaction. The difference between the two transaction models is in a single transition connecting the state `aborted` and the state `start_tx`, which is removed in the linear transaction.

Both the cyclic and the linear transactions may be aligned or drifted. A transaction's drift behavior is modeled with different type of the two states, namely the state `go` and the state `aborted`. The aligned transactions are modeled with a pair of committed states (time delay is not allowed), thus, the transactions start execution immediately. The drifted transactions are modeled with a pair of normal states (time can progress), which enables transactions to drift.

Based on the former descriptions, in a PSTM system verification, the four types of transactions may be used: (i) cyclic drifted transaction, (ii) cyclic aligned transaction, (iii) linear drifted transaction, and (iv) linear aligned transaction.

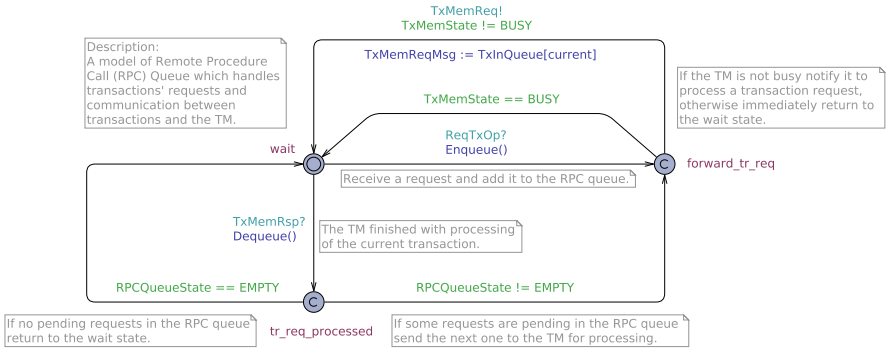

Figure 5
UPPAAL model of RPC Queue, i.e. the automaton `RPCQueue`

3.2.2 The Automaton RPC Queue

An instance of the automaton `RPCQueue` (Fig. 5) starts from the state `wait`. It waits to be notified by some of the transactions or by the transactional memory. At the beginning, the `RPCQueue` is empty, and no pending request exists. The new

incoming request is received during the transition from the state `wait` to the state `forward_tr_req`. The request is stored in a local array of pending requests using the function *Enqueue*(). There are the two possibilities and both of them depends on the current state of the automaton `TxMemory`. If the `TxMemory` is not busy then the `RPCQueue` forwards to it the *current* pending request, which has to be processed. After that, the `RPCQueue` advances to the state `wait` where it waits for new incoming requests or to be notified by the `TxMemory`. If the `TxMemory` is busy than the `RPCQueue` immediately advances to the state `wait`. The received requests wait until the `TxMemory` is available.

By moving from the state `wait` to the state `tr_req_processed` the `TxMemory` notifies the `RPCQueue` that the current request is served. In that state the two possibilities exist too, but in this case, both of them depend on the number of pending requests. If the `RPCQueue` is empty, i.e. no pending requests exist, it moves to state `wait`. If the `RPCQueue` holds any pending request, it forwards the *current* request to the `TxMemory` by moving to the state `forward_tr_req`.

3.2.3 The Automaton Transactional Memory

An instance of the automaton `TxMemory` (Fig. 6) may be either available or busy. The `TxMemory` instance starts from the state `wait`. As long as it resides in the state `wait` it is available, otherwise it is busy and it serves a particular request.

In the state `wait`, it may be notified by the `RPCQueue`. Right after being notified, it moves from the state `wait` to the state `processing`. The variable *TxMemState* denotes whether the `TxMemory` is available or busy.

When the state `processing` is reached, the type of requested operation needs to be checked. Based on the operation type, the `TxMemory` switches either to a path that models the read or to a path for the write functionality.

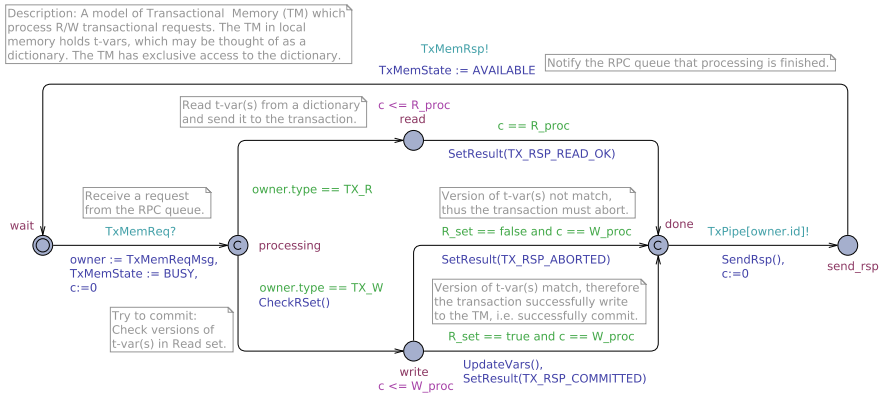

Figure 6
UPPAAL model of transactional memory, i.e. the automaton `TxMemory`

The read operation is served by moving from the state `process` to the state `read`. An additional time invariant is stitched to the state `read` defining the operation duration. Applying the same modeling approach as in the case of transaction processing duration, a clock variable *c* is introduced. The clock variable *c* ensures that a transition is taken immediately as it becomes enabled. Particularly, read and write operations are enabled after *R_proc* and *W_proc* time units, respectively. The variables *R_proc* and *W_proc* denote transactional memory operation duration and they are defined as a template argument, as well.

The transaction response is generated on the transition from the state `read` to the state `done`. The function *SetResult*() updates the transaction's read set with the specified t-vars from the dictionary. Also, it updates the result of the requested operation. The result of the read operation is always successful – the internal value *TX_RSP_READ_OK* sets an execution result value to *TX_RSP_COMMITTED*.

Once the response data are prepared, the transaction and the `RPCQueue` have to be notified. The particular transaction and the transactional memory are synchronized on the transition from the state `done` to the state `send_rsp`. Along the transition, *SendRsp*() function is executed. The function *SendRsp*() sends the response to the transaction by moving the response data to the variable *TxMemRspMbx*. Finally, on the transition from the state `send_rsp` to the state `wait`, the `RPCQueue` is notified that the transactional memory is now available.

The write operation is served in two steps. The first step is to check the t-vars in the read set. For this purpose, the function *CheckRSet*() is used. It loops through the t-vars in the read set and checks if each t-var from the set matches the most recent version of the corresponding t-var in the dictionary. Like the state `read`, the state `write` models the write operation. An additional time invariant ensures that the automaton stays in the state `write` exactly *W_proc* time units simulating the processing duration.

The second step of the write operation is to try to commit the transaction. A commit decision is made, based on the return value of the function *CheckRSet*(). Intuitively, if the result is *true*, the transaction commits, otherwise it aborts. Both paths are modeled on the transitions from the state `write` to the state `done`. The actual t-var(s) update is done by the function *UpdateVars*(). In the case of an abort, the t-vars in the dictionary are unchanged.

4. Transactions Execution Time Analysis

Driven by the desire to make a more faithful transaction model, the drift behavior is modeled as a normal (non-deterministic) state without a time invariant. The fact that a system can behave non-deterministically prevents transactions execution to

be handled in a unified way. This significantly impacts the temporal behavior analysis of a PSTM system and brings us closer to the area of transaction scheduling, which, however, is not the focus of this work.

In order to verify the temporal behavior of the PSTM system for the worst case execution scenario, i.e. the scenario in which all the transactions are started simultaneously, we refine the assumptions regarding the transaction processing duration and the durations of read and write operations. The aim of this theoretical analysis is to make a framework for PSTM temporal behavior verification in the worst case scenario.

Particularly, three assumptions are made. The first assumption is that the processing operation duration t_{p_i} takes the same amount of time, $t_{p_1} = t_{p_2} = \cdots = t_{p_N} = t_p$ for all transactions. An index i denotes i–th transaction in the set of N transactions. This assumption is based on the fact that all transactions in a set, typically execute rather short functions that take similar amounts of time – for instance we can consider a banking system, some kind of an online system such as airline ticketing system, etc. For example, ATM (Automated Teller Machine) functions, such as cash withdrawals, deposits, transfer funds, or obtaining account information, all take similar amounts of time, i.e. approximately the same time t_p. The second assumption is that the transactions processing operations take the same time as the transactional read and write operations, $t_p = t_r = t_w$. In hardware transactional memories, t_r and t_w may differ, but since in PSTM t-var read and write operations perform on Python dictionary data structure it is realistic to assume that $t_r = t_w$. Further on, we assumed $t_p = t_r = t_w$ because processing operations usually are a lightweight task (money transfer, ticket reservation, etc.) rather than being compute intensive. The third assumption is that an aborted transaction immediately retries to commit, until it succeeds (aligned cyclic transaction behavior). This way a run-time execution overhead needed to restart a transaction in a real system is neglected. For the comparable overhead values, the results of the analysis would remain unchanged, otherwise the analysis would be unnecessary complicated and unrealistic for a genuine system. Hence, these assumptions enable us to handle the transactions execution times in a unified way without loss of generality of the results of the analysis.

For $N > 2$ a transactions set execution may evolve through three characteristic phases. These execution phases are used to derive a formula for the transaction commit times. Before the analysis of the execution phases, let us first introduce preliminaries.

Lemma 1. An execution time of the single transaction set takes exactly $t = t_r + t_p + t_w$.

Proof. The single transaction is executed sequentially, thus its execution time is equal to the duration of all executed operations. □

Lemma 2. In each execution attempt the maximum queue delay $d_{q_{MAX}}$ is equal to the number of pending transactions p.

Proof. The proof is rather intuitive. If all pending transactions p request a transactional memory operation at the same time, then the lastly enqueued request, is actually the p^{th} transactional request. Further, if a transactional memory operation takes some time t_p, then p^{th} transactional request in the queue will be processed after exactly the p times of the time t_p. □

Claim 1. The first transaction which requests the read operation commits at time $t_c = p \cdot t_r + t_w$.

Proof. Obviously, the transaction which requested the read operation first is ahead of all the others, thus it will also be the first to request the write operation. The write request will be executed just after the last read request in the queue, i.e. after reaching the maximum queue delay $d_{q_{MAX}}$. Since the queue delay $d_{q_{MAX}}$ depends on the number of pending transactions p, the commit time t_c is equal to p read operation times t_r and one write operation time t_w.

Claim 2. A transactions commit attempt terminates at time $t_{ter} = p \cdot (t_r + t_w)$.

Proof. All read operation requests are served at time $p \cdot t_r$. The number of pending transaction in the current attempt is unchanged, therefore $p \cdot t_w$ time is needed for all write operation requests to be served. A commit attempt is terminated at the time when the last write operation is served, $t_{ter} = p \cdot t_r + p \cdot t_w$.

Claim 3. In a commit attempt *at*, where $at \geq 2$, a transaction commits at time $t_c = \sum_{i=1}^{at-1} t_{ter_i} + p \cdot t_r + t_w$

Proof. A new commit attempt starts when a read operation request from the first transaction in a set of aborted transactions is received by the transactional memory. The aborted transactions immediately retry, i.e. they send a new read operation request, but it can be served only after the termination of the previous commit attempt(s) – all p write operation requests from the previous attempt are served first. A set comprised of aborted transactions is the new pending transactions set, therefore the first transaction from the set commits at time $t_c = t_{ter_{at-1}} + p \cdot t_r + t_w$

Claim 4. The last transaction commits at time $t_c = \sum_{i=1}^{at-1} t_{ter_i} + t_r + t_p + t_w$

Proof. In the last commit attempt ($at = N$), only one transaction is running, thus it is executed sequentially (*Lemma 1*). Like to the former case, the last commit attempt starts execution after the previous commit attempt is terminated.

Based on the conducted analysis, the following formula for calculating commit times for a set of N transactions in the worst case scenario is derived:

$$t_c(c) = \begin{cases} N \cdot t_r + t_w, & | \; c = 1 \\ \sum_{i=1}^{at-1} t_{ter_i} + p \cdot t_r + t_w, & | \; c \neq 1 \; and \; c \neq N \\ \sum_{i=1}^{at-1} t_{ter_i} + t_r + t_p + t_w. & | \; c = N \end{cases} \quad (1)$$

The formula gives the commit time t_c as the function of the number of occurred commits. The number of transactions in a set is denoted as *N*. The number of committed and pending transactions are denoted as c and p, respectively. For $c = 0$ no committed transaction exists. A linear function $N = c + p$ models the relation between the total, committed, and pending number of transactions in the system. The termination time of i^{th} commit attempt t_{ter_i} is given by *Claim 1*. The formula (1) defines a framework for transactions temporal analysis.

5 PSTM Verification

As mandatory properties of almost any system, the correctness criteria include *safety*, *liveness*, *deadlock freeness*, and *reachability* properties. PSTM system is not the exception, therefore, all the former properties are checked.

An execution scenario used to verify a property may vary for each property. Namely, some properties may be verified using all types of transactions while for the others an execution scenario restricted to only one type of transactions may be more suitable. However, the property statements are formulated in such manner that they could be applied to any PSTM-based system with arbitrary number of transactions, t-vars, and read, write, and processing duration values. The given properties are generalized and formulated as statements accompanied with the equivalent CTL formulas expressed in UPPAAL query language [17].

5.1 Deadlock Freeness

The property deadlock freeness can be verified with the following query:

```
A[] not deadlock
```

Let us name the former query as *Deadlock Freenees*. It verifies that a system is deadlock free. A remark about this property is related to terminal states and how UPPAAL tool defines a deadlock state. In UPPAAL tool, a state is a deadlock state if there are no outgoing action transitions either from the state itself or any of its delay successors [17]. This is important for both linear and cyclic transactions. Namely, the linear transactions finally end up either in the state `aborted` or the state `committed` whereas the cyclic transactions end up in the state `committed`. Due to the lack of terminal states this may be considered as a deadlock. In order adapt the transaction automation to the deadlock definition, a self-loop transition is added to the final state(s) of an automaton – `aborted` and `committed` states in the case of linear transactions, and only `committed` state in the case of cyclic transactions.

5.2 Safety

Commonly, in a transactional memory environment, safety property is reduced to atomicity property. In order to get a clear picture of the problem, we need to consider a set of transactions in the PSTM system and analyze a relevant verification scenario. The safety property is verified from the two complementary angles, the perspective of transactional memory operations execution and the perspective of transactions execution in the case of the highest concurrency.

The first safety property, named *Safety I*, claims that in any execution scenario a transactional memory operation is executed atomically, i.e. the transactional memory always serves only *one* transaction's request at the time, and, more precisely, the currently *front* request in the RPC queue. The property can be verified with the following CTL query:

```
A[] TxMemory.processing imply queue[current].id ==
                              owner.id
```

It brings to the focus the components which are fundamental for safety. It verifies that always when the automaton `TxMemory` is in the state `processing`, the `current` request in the `queue` is actually the same request which is the `owner` of the transactional memory.

The second, stronger safety property, named *Safety II*, claims that in the case of the highest concurrency, in which all the transactions are in the conflict, only one transaction may commit at the time. It is stated as follows: from a set of *N* conflicted transactions, only *one* transaction may commit – a transaction whose

commit request is received *first* – while remaining $N-1$ transactions aborts: $Tr_{commit} = Tr_{id=first}$; $Tr_{abort} \in \{Tr_{id \neq first}\} = \{Tr_{id=0}, Tr_{id=1}, \cdots, Tr_{id=N-1}\} - \{Tr_{id=first}\}$. The execution scenario for this property is composed of aligned linear transactions only. The property can be verified with the following CTL query:

```
A[] forall (i:IDs) ((i == first imply S1) and (i != first imply S2))
S1 := Tx[i].result != TX_RSP_ABORTED
S2 := Tx[i].result != TX_RSP_COMMITTED
```

The auxiliary statements `S1` and `S2` do not affect the logic behind the query, they are used to relax a query expression. The statement `S1` limits a transaction result to the two possible values, `TX_RSP_NONE` and `TX_RSP_COMMITTED`. The statement `S2` limits a transaction result to `TX_RSP_NONE` and `TX_RSP_ABORTED`. The value `TX_RSP_NONE` is used for initialization purposes, which means that the transaction did not request any operation yet.

5.3 Liveness

Liveness property is used as a warrant of a system progress. It guaranties that all transactions will finish eventually. The three liveness properties are introduced: *Liveness I*, *Liveness II*, and *Liveness III*. Each definition of the liveness property aims to verify a system progress. The difference between them is the number of details which they comprehend. These properties are defined in the increasing order of their respective power (from the weakest to the strongest). The strongest property *Liveness III*, introduces the timings of the system evolution. The properties are defined in a context of *N* cyclic transactions.

The liveness property *Liveness I* is used as a basic (sanity) functional correctness test. It checks if it is possible for a system to reach a state in which all the transactions are committed. The liveness property *Liveness I* ensures the following: for a set of *N* cyclic transactions, there is a path to a state in which all the transactions will commit eventually, in any transactions schedule. The property can be verified with the following CTL query:

```
E<> forall (i:N) TxW(i).committed
```

The liveness property *Liveness II* is stronger than the former in the sense that it includes pending transactions. It verifies that a specific relation between the committed, pending, and total number of transactions holds in any state. This relation defines the total number of transactions in a system as the sum of already committed and still running (pending) transactions. The property can be verified with the following CTL query:

```
A[] forall (i:N) ((i == pending) imply (L >= (NUM_OF_TX - i))
```

L := Tx(0).committed + ⋯ + Tx(N-1).committed

An auxiliary statement L defines the total number of transactions instances Tx(i), which are in the state committed. Clearly, if a transaction is not in the state committed, then it is still running (trying to commit).

Liveness property named *Liveness III*, as a warrant of a system progress enforces the strict timing of the system. It utilizes the formula (1) to define an upper bound of a time window along which the system advances. For the kind of property such like liveness, the timings of the system progress are critical. The example of a negative and undesired timing is a system *livelock*. In order to verify that a system progress is positive, a number of committed transactions must be analyzed, too.

A system progresses in a positive way as long as some of the transactions commits. This can be interpreted as the definition of the property *Liveness I*. Indeed, the property *Liveness III* may be viewed as a timed version of the property *Liveness I*. It is defined in the similar way, but with an additional criterion which defines an upper bound of time up to which a particular number of transactions must commit. Each time window with higher lower and upper bounds, increases the number of committed transactions – it increases the number of commits. With this approach, the property *Liveness III* can be considered as a proof that the PSTM system is livelock free as well. The property can be verified with the following CTL query:

E<> now == $t_{committ}$ and (L == *num_of_committed*)

L := Tx(0).committed + ⋯ + Tx(N-1).committed

num_of_committed ∈ {1,⋯,*N*}

t_{commit} = commit_time(*num_of_committed*, *N*)

An auxiliary statement L defines a number of transaction instances Tx(i) that are in the state committed while the value of *num_of_committed* is the expected number of committed transactions. In order to verify the commits of a set of *N* transactions, the same number *N* of verification queries has to be defined. The value of *num_of_committed* is the parameter which denotes a particular number of transaction commits which is the subject of verification. The value $t_{committ}$ is calculated by function commit_time() which actually is an implementation of the formula (1).

5.4 Reachability

The reachability property is included as additional proof of the liveness property. It is exploited to verify the system termination, i.e. the system state after all the transactions have been finished. We use the reachability to verify that the system terminates correctly in any execution scenario. The property *Reachability* can be verified with the following CTL query:

```
R --> ((TxMemory.wait and TxMemState == AVAILABLE) and
(RPCQueue.wait and RPCQueueState == EMPTY))
R := (Tx(0).committed or Tx(0).aborted) and … and
(Tx(N-1).committed or Tx(N-1).aborted
```

An auxiliary statement *R* claims that *N* `Transaction` instances are either in the state `committed` or in the state `aborted`, i.e. that all transactions are finished.

6 Verification Results

The property verification queries can be applied to a system with an arbitrary number of transactions and arbitrary duration of read, write and processing time. Despite of model's generic nature, it was not feasible to conduct verificaion for all arbitrary sets of parameters, so for all verification properties the operations read, write and processing take one time unit, $t_p = t_r = t_w = 1$. These particular values make the verification process viable, without losing anything from the generality of the applied method.

Increasing the number of transactions influences the verification time required by a model checker to explore a state space and make a verdict about the property. Considering PSTM architecture design, the correctness properties can be verified using only two transactions, which are required for minimal level of contingency that may provoke undesired system behavior. A system can be verified against an arbitrary number of transactions *N*, if it is necessary. In the conducted verification, the number of transaction instances is increased as long as the verification execution time was reasonable. Actually, at some point, due to the expanded state space, the model checker may consume all of the operating memory of the host machine. In such a case, the tool is capable of utilizing more memory by using swapping mechanism, although this causes very long verification time. However, this bottleneck is related to the host's hardware capabilities.

The summary results are given in Table 1. The results confirmed that PSTM satisfies all the previously defined properties. Although the correctness property results are the main objective of the conducted verification they are accompanied with additional data, namely the number of transactions, the verification execution time, and the number of explored states, which may be beneficial for other researchers who want to get insight into verification process statistics.

The system complexity is elevated by increasing the number of transaction instances. In addition to the number of transaction instances, transactions type is relevant as well. For example, property verification for a set of cyclic transactions is more demanding than the verification of the same property against a set of linear transactions. Further, verifying a property against drifted cyclic transactions

is certainly more demanding than verifying the same property against aligned cyclic transactions. The same can be concluded for linear transactions. The reason is rooted in the automata structure – firstly, the model of linear transaction is lighter than the model of cyclic transaction, and secondly, for the model checker it is easier to deal with a committed state (aligned transaction) than with a non-deterministic state (drifted transactions).

Unquestionably, the number of time invariants, i.e. clock variables, significantly impacts a state space which has to be explored. UPPAAL tool is very sensitive to clock variables which causes state space to expand faster. Generally, relaxing the number of clock variables would reduce the state space size.

Table 1

Summary of verification results and performance statistics

Property	Number of Transactions	Type of Transactions	Time	States Explored
Deadlock Freeness	6	Linear Drift	1m 2s 110ms	9 045 757
	5	Cyclic Drift	1m 2s 440ms	10 140 401
	8	Linear Aligned	1m 2s 90ms	8 014 336
	7	Cyclic Aligned	24s 930ms	3 597 232
Safety I	6	Linear Drift	32s 180ms	9 045 757
	5	Cyclic Drift	34s 670ms	10 140 401
	8	Linear Aligned	34s 790ms	8 014 336
	7	Cyclic Aligned	12s 950ms	3 597 232
Safety II	8	Linear Aligned	38s 740ms	8 014 336
Liveness I	5	Cyclic Drift	1m 0s 20ms	17 489 881
	7	Cyclic Aligned	13s 450ms	3 597 232
Liveness II	5	Cyclic Drift	5s 510ms	1 690 633
	7	Cyclic Aligned	12s 420ms	3 577 073
Liveness III	7	Cyclic Aligned	13s 680ms	3 577 073
Reachability	6	Linear Drift	32s 500ms	9 186 157
	5	Cyclic Drift	52s 620ms	14 008 901
	8	Linear Aligned	34s 650ms	8 094 976
	7	Cyclic Aligned	13s 500ms	3 662 752

The experiments are conducted using Ubuntu 14.04 64bit OS, which is running on Intel i7-3770 CPU with 16 GB of RAM, and UPPAAL 64-4.1.19 (rev. 5648).

Conclusions

In this paper we tried to overcome the shortcomings of existing STM formal verification approaches by introducing an approach based on timed automata formalism which uses existing STM's program code as its input. Our verification approach respects a STM solution implementation details aiming to make verification models as faithful counterparts of the implementation rather than

developing a generalized verification framework to cover more transaction execution models and semantics. Particularly, we demonstrated our approach to formal verification of one STM, written in the Python language, named Python Software Transactional Memory (PSTM) [13], utilizing UPPAAL tool [17, 14]. Based on the PSTM architecture and implementation details, we derived a model of a PSTM system which is formally verified by the UPPAAL model checker.

The verification of the system correctness includes checking deadlock-freeness, safety, liveness, and reachability properties. We analyzed the system execution against the types of aligned and drifted read-write transactions which share a common transactional variable. The system temporal behavior is analyzed, too. For the purpose of the system temporal behavior verification a framework for calculating transactions execution times in the worst case scenario is developed. By applying generalized property queries to a verification system based on a different number and type of transactions, we successfully verified that our PSTM system model satisfies all the formerly mentioned properties. The results presented in the paper may be useful for the academia and the industry researches as well.

The direction of future work is oriented towards development and formal verification of a distributed (P)STM for the Internet of Things.

Acknowledgement

This work was supported by the Ministry of Education, Science, and Technology Development of Republic of Serbia under grants III-44009, ON174026, III044006.

References

[1] M. Herlihy, J. E. B. Moss: Transactional memory: Architectural support for lock-free data structures, *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pp. 289-300, 1993

[2] T. Harris, J. R. Larus, R. Rajwar: Transactional Memory, 2nd edition, Morgan and Claypool Publishers, 2010

[3] N. Shavit, D. Touitou: Software transactional memory, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pp. 204-213, 1995

[4] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, L. D. Zuck: Verifying correctness of transactional memories, *Proceedings of the 7th International Conference on Formal Methods in Computer - Aided Design (FMCAD 2007)*, pp. 37-44, 2007

[5] M. Emmi, R. Majumdar, R. Manevich: Parameterized verification of transactional memories, *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI'10)*, pp. 134-145, 2010

[6] R. Guerraoui, M. Kapałka: On the correctness of transactional memory, *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pp. 175-184, 2008

[7] R. Guerraoui, T. A. Henzinger, V. Singh: Model checking transactional memories, *Distributed computing*, Vol. 22 (3), pp. 129-145, 2010

[8] S. Doherty, L. Groves, V. Luchangco, M. Moir: Towards formally specifying and verifying transactional memory, *Formal Aspects of Computing (FAOC)*, Vol. 25 (5), pp. 769-799, 2013

[9] B. Kordic, M. Popovic, S. Ghilezan, I. Basicevic: An approach to formal verification of python software transactional memory, *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems (ECBS'17)*, pp. 1-10, 2017

[10] M. Amitay, M. Goldstein: Evaluating the peptide structure prediction capabilities of a purely ab-initio method, *Protein Engineering, Design and Selection*, Vol. 30 (10), pp. 723-727, 2017

[11] M. Goldstein, E. Fredj, R. B. Gerber: A new hybrid algorithm for finding the lowest minima of potential surfaces: approach and application to peptides, *Journal of Computational Chemistry*, Vol. 32, pp. 1785-1800, 2011

[12] PyPy Software Transactional Memory: http://doc.pypy.org/en/latest/introduction.html (accessed 15 October 2018)

[13] M. Popovic, B. Kordic: PSTM: Python software transactional memory, *22nd Telecommunications Forum Telfor (TELFOR 2014)*, pp. 1106-1109, 2014

[14] UPPAAL tool home page: http://www.uppaal.org (accessed 15 October 2018)

[15] C. Belwal, A. M. K. Cheng: Schedulability analysis of transactions in software transactional memory using timed automata. *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1091-1098, 2011

[16] J. Perhac, D. Mihalyi, V. Novitzka: Modeling synchronization problems: from composed Petri nets to provable linear sequents. *Acta Polytechnica Hungarica*, Vol. 14 (8), pp. 165-182, 2017

[17] G. Behrmann, A. David, K. G. Larsen: A tutorial on Uppaal, *Lecture Notes in Computer Science*, Vol. 3185, pp. 200-236, 2004

[18] R. Alur, D. L. Dill: A theory of timed automata, *Theoretical Computer Science*, Vol. 126 (2), pp. 183-235, 1994