Rev. Roum. Sci. Techn.– Électrotechn. et Énerg.
Vol. **64**, *4*, pp. 423–428, Bucarest, 2019

FORMAL VERIFICATION OF LOCAL AND DISTRIBUTED PYTHON SOFTWARE TRANSACTIONAL MEMORIES

MARKO POPOVIC[1], MIROSLAV POPOVIC[1], SILVIA GHILEZAN[1,2], BRANISLAV KORDIC[1]

Key words: Formal verification, Push/pull semantic model, Serializability, Python, Software transactional memory (STM).

Both local and distributed Python STMs are targeting a wide range of application domains, including critical infrastructures, such as cyber-physical systems, internet of things, etc., and formal verification of such software components is considered mandatory. Recently, the push/pull semantic model of transactions has appeared as a solution that unifies a wide range of transactional memory algorithms. In this paper, we formally prove that both local and distributed Python STM implementations are serializable by constructing their push/pull model and by showing that the push/pull model satisfies the correctness criteria for the relevant push/pull semantic rules. The main contributions of the paper are the following: (i) the PSTM and DPSTM push/pull semantic model, (ii) the proofs of the relevant push/pull semantic rules, and (iii) the way how the model and the proofs have been constructed.

1. INTRODUCTION

Transactional memory (TM), originally introduced as an architectural support for lock-free concurrent data structures [1], became a well-known paradigm that aims to replace locks with transactions. TM key advantages are: (i) it supports higher-level abstraction and composition and thus makes parallel programming easier, and (ii) it supports speculative (lock-free) transactional execution and thus provides better application performance, especially when contention among threads is lower [2]. It was not long after a software transactional memory (STM) was introduced as an implementation of TM in software [3].

Python STM (PSTM) [4] is designed as a general-purpose STM for Python targeting a wide range of application domains, from simulations [5, 6] and critical infrastructures [7], to data science and enhanced learning [8]. As a curiosity, it seems appropriate mentioning that PSTM was originally created with the aim to eliminate a barrier-like synchronization in a very large Python computational-chemistry simulation program for the protein structure prediction problem [5, 6]. Since any STM without contention management is known to be vulnerable to higher concurrency workloads, the online transaction scheduler architecture and four scheduling algorithms for scheduling transactions on PSTM were developed [9,10] in order to lessen this weakness. Additionally, common concurrent data structures [11] based on PSTM, such as a concurrent list [12], concurrent queue [13], etc., were developed to aid easier PSTM-based application development. Finally, the first version of distributed PSTM (DPSTM) for applications in distributed settings was recently developed [14].

Traditionally, the formal verification of software components of critical infrastructures is considered mandatory. Therefore, PSTM has already been formally verified using two independent and complementary approaches that are based on timed automata (TA) [15] and communicating sequential processes (CSP) [16], respectively. Of course, this research remains open for application of other formal methods. One particularly interesting formalism is the push/pull semantic model of transactions [17].

In this paper, we formally prove that both (D)PSTM (PSTM and DPSTM) implementations are serializable by applying the push/pull semantic model. The main motive for this paper was to render even better confidence in the PSTM correctness by: (i) using this third independent and complementary approach (besides TA and CSP), and (ii) by introducing (D)PSTM to a wider research community familiar with the well-known push/pull semantic model.

The main original contributions of the paper that may be utilized by other researchers and practitioners are the following: (i) the (D)PSTM push/pull semantic model, (ii) the proofs of the relevant push/pull semantic rules, and (iii) the way how the model and the proofs have been constructed. Factors that qualify novelty and utility of these contributions are the following: (i) this is the first paper that tries to rigorously apply the push/pull semantic model to a real STM not covered by [17], (ii) although the authors of the push/pull semantic model have put a respectable effort in defining their model, formalizing and verifying a real STM using their approach is far from being trivial, and (iii) there is a realistic expectation that the push/pull semantic model may be widely used for specifying transactional-based concurrent software, for example see [18].

1.1 RELATED WORK

PSTM was formalized using TA and analyzed by the model checker UPPAAL [15]. PSTM TA-based model comprises automata representing a transaction, the PSTM queue, and the TM. The three properties that were proved are: (i) safety (*i.e*. atomicity), liveness (at least one of the concurrent transactions will commit), and (iii) termination (all cyclic transactions eventually complete).

PSTM was also formalized using CSP, and analyzed by the model checker Process Analysis Toolkit (PAT) [16]. The lower-level abstraction model comprises processes representing a transaction, the Application Programming Interface (API), the server, and the system dictionary. The three properties that were proved are: (i) deadlock-freedom, (ii) ACI (Atomicity, Consistency, Isolation), and (iii) optimism (at least one of the concurrent transactions will commit).

The three particular PSTM transaction scheduling algorithms (Round Robin, Execution Time Load Balancing,

[1] University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovica 6, 21000 Novi Sad, Republic of Serbia, E-mails: {marko.popovic, miroslav.popovic, branislav.kordic}@rt-rk.uns.ac.rs, gsilvia@uns.ac.rs.
[2] Mathematical Institute SASA, Belgrade, Republic of Serbia, E-mail: gsilvia@uns.ac.rs.

and Avoid Conflicts) from [10] were formalized using the process algebra CSP, and analysed by the model checker PAT, in order to evaluate them by: (i) verifying the properties of deadlock freeness and starvation freeness, and (ii) comparing the performance of these three algorithms from the perspective of makespan, speedup, aborts, and throughput [19].

The push/pull semantic model [17] may be used to prove serializability of a given STM as follows. Since the push/pull semantic model satisfies serializability, one may prove that a given STM satisfies serializability by: (i) constructing its push/pull model, and (ii) proving that this model satisfies respective correctness criteria. Proofs of correctness criteria typically involve commutative properties of sequential programs [20].

A preliminary version of this paper appears in the Proceedings of the ECBS 2019 [21]. This paper is the extension of [21] that now covers formal verification of both PSTM and DPSTM, shortly written as (D)PSTM.

2. PUSH/PULL SEMANTIC MODEL

The push/pull model of [17] abstracts the system state with a global shared log of the operations that were pushed by all the threads, and a per-thread local log of operations that were pulled or applied by a thread. The model includes the following rules:

- APPLY(*op*): apply an operation *op* to local log.
- UNAPPLY(*op*): rewind local log to remove *op*.
- PUSH(*op*): push *op* to the global log.
- UNPUSH(*op*): recall *op* from the global log.
- PULL(*op*): pull *op* from the global log.
- UNPULL(*op*): discard *op* from the local log.
- CMT(*txn*): commit a transaction *txn*.

Note that (D)PSTM has only committed operations in the global log.

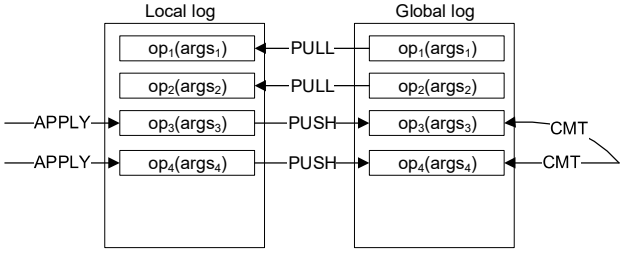

Fig. 1 – A Push/Pull transaction for money transfer.

Figure 1 illustrates a simple example of a push/pull transaction, in particular a push/pull transaction for money transfer, which will be used as a running example in this paper. The task of this transaction is to realize the payment of p euros from the account x to the account y by subtracting p euros from x and adding p euros to y. In order to realize this task, this transaction conducts the following steps: (1) pull the operations op_1 and op_2 that were used to set the current values of transactional variables (t-variables) x and y, respectively, (2) apply the operation op_3 that sets the new value of x to $(x - p)$, (3) apply the operation op_4 that sets the new value of y to $(y + p)$, (4) push the operations op_3 and op_4 from its local log to the global log, and (5) commit the transaction, *i.e.* commit the operations op_3 and op_4 in the global log.

Next, we briefly recall the rules that will be used to model (D)PSTM semantics, and these are: APPLY, PUSH, PULL, and CMT [17].

Definition 1 (Relevant push/pull rules). The (D)PSTM relevant push/pull semantic rules are defined as follows:

$$\frac{(i)\quad c_1 \downarrow_{\text{tx}} (m, c_2) \qquad (ii)\quad L_1 \text{ allows} \langle m, \sigma_1, \sigma_2 \rangle \qquad (iii)\quad \text{fresh}(id)}{\{\text{tx } c_1, \sigma_1, L_1\}, G_1 \xrightarrow{\text{fwd}} \{\text{tx } c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id \rangle, \text{unpushed } c_1]\}, G_1} \text{APPLY}$$

$$\frac{(i)\quad op \lhd \lfloor L_1 \rfloor_{\text{unpushed}} \qquad (ii)\quad \lfloor G_1 \rfloor_{\text{gUCmt}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \lhd op \qquad (iii)\quad G_1 \text{ allows } op}{\{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{unpushed } c_2] \cdot L_2\}, G_1 \xrightarrow{\text{fwd}} \{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{pushed } c_2] \cdot L_2\}, G_1 \cdot [op, \text{gUCmt}]} \text{PUSH}$$

$$\frac{(i)\quad op \notin L \qquad (ii)\quad L \text{ allows } op \qquad (iii)\quad op \lhd \lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}}{\{\text{tx } c_1, \sigma_1, L\}, G_1 \cdot [op, g] \cdot G_2 \xrightarrow{\text{fwd}} \{\text{tx } c_1, \sigma_1, L \cdot [op, \text{pulled}]\}, G_1 \cdot [op, g] \cdot G_2} \text{PULL}$$

$$\frac{(i)\quad \text{fin}(c) \qquad (ii)\quad L_1 \subseteq G_1 \qquad (iii)\quad \forall op \in \lfloor L_1 \rfloor_{\text{pulled}}, (op, g) \in G_1.g = gCmt \qquad (iv)\quad \text{cmt}(G_1, L_1, G_2)}{T_1 \cdot \{(\text{tx } c, c_1), \sigma, L_1\} \cdot T_2, G_1 \rightarrow T_1 \cdot \{c_1, \sigma, L_1\} \cdot T_2, G_2} \text{CMT}$$

$$\text{cmt}(G_1, L_1, G_2) \equiv G_2 = \text{map}(\lambda(op, g) \cdot \begin{cases} (op, \text{gCmt}) & \text{if } op \in \lfloor L_1 \rfloor_{\text{pushed}} \\ (op, g) & \text{otherwise} \end{cases}) G_1$$

In Definition 1 above, we use the symbol '$\lhd$' for the left-mover operator, because the symbol '◀' that was used in [17] is not available in MS Equation editor. Also, authors of [17] made an error in CMT in [17], which we corrected here based on their technical report, see [19] in [17].

According to authors of [17], correctness criteria for push/pull rules define under which conditions these rules may be applied; essentially they are the rules' premises, i.e. preconditions. For each rule in Definition 1, its *preconditions* or *correctness criteria* (these are synonyms) are listed above the horizontal line within the rule.

3. PSTM

PSTM-based system architecture is a typical client-server architecture written in Python [4]. Transactions (clients) request services from PSTM (server) by calling PSTM API functions. PSTM serves these requests by managing the dictionary of shared t-variables, D (like a global log in the push/pull model). Transactions' requests are sent to PSTM over the FIFO queue, q, where they get serialized.

A PSTM transaction typically gets its local t-variable copies (like a local log in the push/pull model), does some processing, including updating some local t-variable copies,

and finally commits all its operations on shared *t*-variables. Informally, PSTM is serializable because it serves all the requests atomically. The following two definitions are from [21]:

"A dictionary *item* is a pair (*ikey*, *ival*), where *ikey* and *ival* are its key and value, respectively. We write the item read operation as: *D*[*ikey*] or read(*ikey*), and the item write operation as: *D*[*ikey*] := *ival* or write(*ikey*, *ival*).

A *t*-variable is a tuple (*key*, *ver*, *val*), where *key*, *ver*, and *val* are *t*-variable key, version, and value, respectively. A *t*-variable (*key*, *ver*, *val*) is stored as the item (*key*, (*ver*, *val*)) in the system dictionary, thus item's *ikey* and *ival* are equal to *ikey* = *key* and *ival* = (*ver*, *val*)."

The main functions provided by PSTM API are the following:

- addVars(*q*, *K*) / *v*
- putVars(*q*, *W*) / *v*
- getVars(*q*, *K*) / *V*
- commitVars(*q*, *C*) / *v*

The argument *q* is the PSTM queue. The argument *K* is a list of *t*-variable keys. The argument *W* is the list of *t*-variables to be written. The argument *C* is a list [*R*, *W*], where *R* and *W* are read and write lists of *t*-variables, respectively. For more details on PSTM API, see [21].

As already mentioned, PSTM server process serves PSTM API calls sequentially, by calling its internal functions with the same name but without the first argument *q*. The pseudocodes of these internal functions are given in Algorithms 1 to 4, respectively.

Algorithm 1: The PSTM server function addVars.

```
1: addVars(K)
2:   for k in K
3:     D[k] := (0, none)
4:   return 'yes'
```

Algorithm 2: The PSTM server function putVars.

```
1: putVars(W)
2:   return commitVars([ [], W ])
```

Algorithm 3: The PSTM server function getVars.

```
1: getVars(K)
2:   V := []  // set V to the empty list
3:   for k in K
4:     vv := D[k]
5:     if vv = none then v := (false, none)
6:     else v := (true, vv)
7:     V := V + v  // append v to V
8:   return V
```

Algorithm 4: The PSTM server function commitVars.

```
1: commitVars(q, C)
2:   R, W := C
3:   if for all t in (R union W). t.ver = D[t.key].ver then
4:     for w in W
5:       write( w.key, (D[w.key].ver + 1, w.val) )
6:     return 'yes'  // a transaction got committed
7:   else
8:     return 'no'  // a transaction got aborted
```

4. DPSTM

Like PSTM-based system architecture, DPSTM-based system architecture is also a client-server type of architecture [14]. The main difference between these two architectures is that the former architecture supports centralized applications comprising transactions that execute on a single multicore machine, whereas the latter architecture supports distributed applications comprising transactions that execute on different machines in a network. Since in a DPSTM-based system, transactions and DPSTM execute on different machines, transactions use their proxies (also known as DPSTM clients) to request service from DPSTM.

A typical hardware infrastructure of a DPSTM-based system is a computer network, such as Internet, which is used to connect remote processors (computers, mobile phones, Internet of Things, *etc*.) to a server computer. The former hosts distributed application processes comprising transactions and their proxies, whereas the latter hosts DPSTM. Since DPSTM is targeting critical infrastructures, the proxies and DPSTM server process communicate over an authentication-secured connection.

DPSTM-based system architecture was designed as an extension of PSTM-based system architecture. DPSTM maintains the same system dictionary of shared *t*-variables, *D*, as PSTM. Each *t*-variable (*key*, *ver*, *val*) is again stored in *D* as a pair (*ikey*, *ival*), where *ikey* is equal to *key* and *ival* is equal to (*ver*, *val*). DPSTM supports the same set of operations on *D* (addVars, putVars, etc), and it performs these operations atomically, which is the key idea that provides serializability of both architectures. Similarly, DPSTM transactions have the same behavior, *i.e*. the same lifecycle, as PSTM transactions – they get their copies of *t*-variables at the beginning, do local processing, and update modified *t*-variables at the end.

The architectural extension is made by introducing proxies, which transparently connect transactions to DPSTM, such that transactions are not aware that they and DPSTM execute in different machines – transactions call operations on proxies, which simply delegate these operations to DPSTM.

DPSTM was designed to have the API with the same semantics as PSTM API in order to enable easy porting of already developed PSTM based software components, such as concurrent data structures, to DPSTM. The only difference between the PSTM API and the DPSTM API is their different syntax – the former is functional (*i.e.* procedural), whereas the latter is object-oriented. More precisely, PSTM API is defined as a set of stm.py module's functions, whereas DPSTM API is defined as a set of proxy object's functions. All the PSTM API functions have their equivalents within the DPSTM API.

The main syntactical difference between PSTM and DPSTM APIs is in the syntax of API function calls. Each PSTM API function has the PSTM queue as its first argument and is called as a simple (Python module) function, whereas the equivalent DPSTM API function does not have the queue argument and is called on a proxy object (that hides the network communication and the DPSTM queue). More formally, the PSTM API function call *f*(*q*, *args*) is equivalent to the DPSTM API function call *p.f*(*args*), where *q* is the PSTM queue, *args* are other arguments of the API function *f*, and *p* is the proxy object. In both cases, the function *f* returns the same return value *r*.

The DPSTM API is supported by both the transaction proxy object and the DPSTM object. However, the implementations of the DPSTM API within these two objects are different. The transaction proxy object's functions simply delegate their work to the DPSTM object's functions with the same name (by calling the

function with the same name on the DPSTM object, passing it the arguments, and returning it's return value), which in their turn perform the required operations on *D*.

In order to simplify the formalization of DPSTM-based system architecture, we assume that DPSTM API implementation within the DPSTM (object) is defined by the Algorithms 1 to 4, and that calls to DPSTM API functions on the DPSTM object map to the corresponding functions in Algorithms 1 to 4. Then, the DPSTM API implementation within the transaction proxy object is given in Algorithm 5, where *dpstm* is the DPSTM object (in Python this is a *remote customized manager* type of object).

Algorithm 5: The DPSTM API exported by a proxy object.

```
dpstm  // dpstm is the DPSTM object

1: addVars(K)
2:   return dpstm.addVars(K)

3: putVars(W)
4:   return dpstm.putVars(W)

5: getVars(K)
6:   return dpstm.getVars(K)

7: commitVars(C)
8:   return dpstm.commitVars(C)
```

5. FORMAL VERIFICATION

As explained in Section 4, DPSTM is an extension of PSTM for distributed settings, which was designed such that: (i) DPSTM and PTSM transactions have the same behavior, (ii) DPSTM and PSTM APIs have the same semantics, and (iii) DPSTM and PSTM servers provide the same operations with the same semantics. Therefore, quite naturally, DPSTM and PSTM share the common push/pull semantic model, called the (D)PSTM push/pull model.

5.1 (D)PSTM PUSH/PULL SEMANTIC MODEL

This section of the paper presents the (D)PSTM push/pull semantic model using the DPSTM API syntax. Definition adapted from [21]: "The generic (D)PSTM transactional algorithm T is defined as a block of code that starts with a getVars operation, which pulls each *t*-variable only once, performs local processing including updating local copies of *t*-variables, ends with a commitVars operation, and always terminates, see Algorithm 6."

Algorithm 6: The generic (D)PSTM transactional algorithm T.

```
1: T(K)
2:   V := dpstm.getVars(K)
3:   Do local processing and update (write) local copies of t-variables.
4:   v := dpstm.commitVars(C)
```

Note on Algorithm 6: This algorithm generalizes to an arbitrary transaction with an arbitrary number of getVars and commitVars, in an arbitrary order, but for brevity we use this simplest transactional algorithm in this paper.

Definition adapted from [21]: "Definition 2 ((D)PSTM push/pull semantic model). The (D)PSTM push/pull semantic model is a mapping of operations within the generic transactional algorithm T to the corresponding push/pull rules, denoted as function sm, which is defined in the following three sub-definitions:

Definition 2.1 (Mapping of line 2 in Algorithm 6): The getVars(*K*) call maps as follows. For each *k* in *K*, the read *D*[*k*] (line 4 in Algorithm 3) maps to the PULL rule related to the latest write operation from the global log *G* that was used to set *D*[*k*] (line 5 in Algorithm 4). So, getVars(*K*) call maps to |*K*| PULL rules, where |*K*| is the size of *K*.

Definition 2.2 (Mapping of line 3 in Algorithm 6): The updates of local *t*-variable copies (write operations) map to APPLY rules, one per each write.

Definition 2.3 (Mapping of line 4 in Algorithm 6): The commitVars(*C*) call maps as follows. For each *t*-variable *w* in *W*, the corresponding write operation (line 5 in Algorithm 4) maps to the PUSH rule related to that write operation. The end of the for-loop in line 4, in Algorithm 4, maps to the CMT rule. So, commitVars(*C*) maps to |*W*| PUSH rules and one CMT rule altogether."

In order to clarify Definition 2, let's write the algorithm T as the sequence (list) of functions: $T_i = [f_1, f_2, f_3]$, where f_1 is getVars(*K*), f_2 is update(*K*), and f_3 is commitVars(*C*); the function update updates local copies of *t*-variables in *K*. For simplicity, let's assume that *K* has a single element *k*. Then the function sm maps over T_i to produce the sequence of rules $T_o = sm(Ti) = [r_1, r_2, r_3]$, where $r_1 = sm(f_1) = PULL$, $r_2 = sm(f_2) = APPLY$, and $r_3 = sm(f_3) = PUSH, CMT$.

Definition adapted from [21]: "Definition 3 (Sequential specification of logs). Both local and global logs are logs of chronologically ordered write operations with a sequential specification defined such that getVars([*k*]) call pulls the last write to *D*[*k*] and that any read within the local processing evaluates to the value written by the last write."

Next we use the running example of a money transfer transaction as an instance T_m of Algorithm 6, shown in Algorithm 7, in order to illustrate the execution of the algorithm T.

Figure 2 illustrates the execution of Algorithm 7 within the push/pull semantic model. The left side shows the local log L_m of a transaction T_m (executing the algorithm T_m), the middle side shows the global log *G*, and the right side shows transitions. Rows show the current state and the next event. The evolution of the system proceeds downwards. Boxes in logs correspond to operations. Checked operations in the global log are committed.

The variables used in Algorithm 7 and in Figure 2 are the following. The input parameters of T_m are: *K* is the list of keys for the *t*-variables *x* and *y*, $K = [\ k_x, k_y\ ]$ and *p* is the payment (amount of money to be transferred from *x* to *y*). The variables *V*, (*e*, *vv*), *W*, and *R*, are explained in Section 3. Recall that *vv* = (*ver*, *val*). The terms *vv.ver* and *vv.val* are equal to the *vv*'s elements *ver* and *val*, respectively, So, $vv_x.val$ and $vv_y.val$ are the input values of the *t*-variables *x* and *y*, respectively. The variables nv_x and nv_y are the output (new) values of *x* and *y*, respectively (after the money transfer). The variables nvv_x and nvv_y are the output (new) *vv* tuples for *x* and *y*, respectively.

The lines 1 to 3 in Algorithm 7 n nn correspond to the first row in Fig. 2. In this section of Algorithm 7, the transaction T_m calls getVars(*K*), see line 3 in Algorithm 7, which maps to the two PULL rules (these two rules are shown as a cascade of two arrows in Fig. 2) and results in pulling the operations write(k_x, vv_x) and write(k_y, vv_y), where vv_x and vv_y are the tuples (ver_x, val_x) and (ver_y, val_y), which contain the version and the value of the *t*-variables *x* and *y* (with the keys k_x and k_y), respectively. Note that the pulled operations appear in the local log L_i in the second row in Fig. 2.

The lines 4 to 7 in Algorithm 7 do not have a corresponding part in Fig. 2. In this section of Algorithm 7, T_m does its local processing in order to determine the output values of *t*-variables *x* and *y* (nv_x and nv_y).

Algorithm 7: The instance of the algorithm T (money transfer).

```
1: Tm(K, p)   // assume K = [ kx, ky ]
2:   kx := K[0], ky := K[1]
3:   V := dpstm.getVars(K )
4:   (ex, vvx) := V[0]
5:   (ey, vvy) := V[1]
6:   nvx := vvx.val – p
7:   nvy := vvy.val + p
8:   W := R := [ (kx, vvx.ver, nvx), (ky, vvy.ver, nvy) ]
9:   v := dpstm.commitVars([R, W] )
```

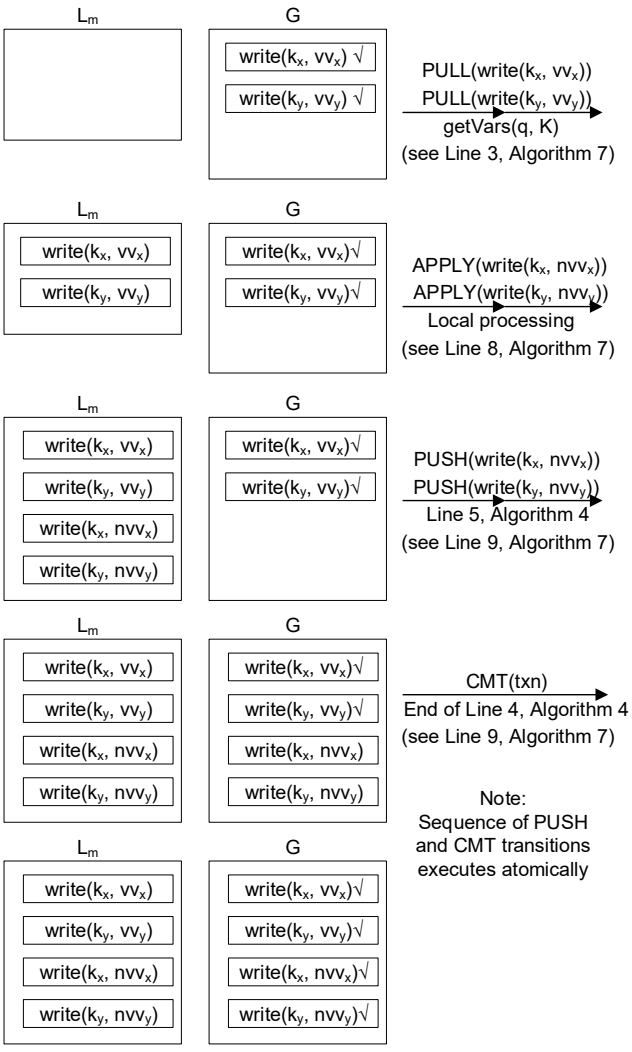

Fig. 2 – Algorithm 7 execution within the push/pull semantic model.

The line 8 in Algorithm 7 corresponds to the second row in Fig. 2. In this section of Algorithm 7, T_m updates its local copies of the *t*-variables *x* and *y*. It does this by creating the lists *R* and *W* that will be used by the function commitVars, see line 9 in Algorithm 7. As the result of these updates, the operations write(k_x, nvv_x) and write(k_y, nvv_y) are appended to T_m's local log, see the third row in Fig. 2.

The line 9 in Algorithm 7 correspond to the third and the forth rows in Fig. 2. In this section of Algorithm 7, T_m calls commitVars(*C*), which maps to the corresponding PUSH and CMT rules (two PUSH rules that are shown as a cascade of two arrows in Fig. 2, and one CMT rule), which are executed atomically, because the whole Algorithm 4 executes atomically. As the result of the two PUSH rules, two uncommitted write operations are appended to the global log *G*, see *G* in the fourth row in Fig. 2 (note that the two new operations are unchecked). Finally, as the result of the CMT rule, the last two write operations are committed, see *G* in the fifth row in Fig. 2.

5.2 (D)PSTM CORRECTNESS CRITERIA PROOFS

In this section we prove that (D)PSTM is serializable by proving that it's push/pull semantic model (Definition 2) satisfies preconditions for the relevant push/pull rules. In Lemmas 1 to 4, for each precondition, we recall (from [17]) what it requires (in the first sentence), and then prove that requirement is satisfied (in the second sentence).

Lemma 1. (D)PSTM push/pull model satisfies the preconditions for APPLY.

Proof. The precondition (i) requires existence of a path in the program code c_1 to the call of the operation *m* and the follow-up program code c_2. This precondition is satisfied by the definition of the transactional algorithm T, because there is a path in T to a write operation *m*.

The precondition (ii) requires that the operation *m* is allowed by the local log L_1. This precondition is satisfied by the definition of sequential specification (see Definition 3) of L_1 and the write operation *m*.

The precondition (iii) requires that the new operation with the unique identification *id* is added to L_1. This precondition is satisfied by the construction of L_1. Q.E.D.

Lemma 2. (D)PSTM push/pull model satisfies the preconditions for PUSH.

Proof. The precondition (i) requires that operation *op* may be moved to-the-left of all unpushed operations in L_1. This precondition is satisfied because in the (D)PSTM push/pull semantic model (see Definition 2) there are no unpushed operations in L_1.

The precondition (ii) requires that all uncommitted operations in G_1, without operations pushed by the current transaction, can be moved to-the-left (authors of [17] made a typo and wrote to-the-right) of the current operation *op*. This criterion is satisfied because in the (D)PSTM push/pull semantic model (see Definition 2) there are no uncommitted operations in G_1.

The precondition (iii) requires that the operation *op* is allowed by G_1. This precondition is satisfied by the definition of the sequential specification (see Definition 3) of G_1 and the write operation *op*. Q.E.D.

Lemma 3. (D)PSTM push/pull model satisfies the preconditions for PULL.

Proof. The precondition (i) requires that the operation op was not pulled before. This precondition is satisfied by the definition of the generic (D)PSTM transactional algorithm T, which requires that each *t*-variable is pulled just once.

The precondition (ii) requires that the operation *op* is allowed by the sequential specification of the local log *L*. This precondition is satisfied by the definition of the sequential specification (see Definition 3) of *L* and the write operation *op*.

The precondition (iii) requires that all the operations performed by the current transaction can move to-the-right of the current operation *op*. This precondition is satisfied because a transaction makes all its local operations when preparing the list *W* for the final commitVars call, and because *t*-variables in *W* are unique, local operations may

be performed in any order. *Q.E.D.*

Lemma 4. (D)PSTM push/pull model satisfies the preconditions for CMT.

Proof. The precondition (i) requires the existence of a path through a transaction **tx** *c* that reaches the stopping statement **skip**. This precondition is satisfied by the definition of the generic (D)PSTM transactional algorithm T, because it always terminates.

The precondition (ii) requires that transaction's local log L_1 must be included in the global log G_1. This precondition is satisfied by the definition of L_1 and G_1 (see Definition 3), and the construction of the (D)PSTM push/pull semantic model (see Definition 2).

The precondition (iii) requires that all operations pulled by the current transactions were pushed by transactions that meanwhile successfully committed. This precondition is satisfied because G_1 contains only committed operations (see Definition 2).

The precondition (iv) requires that the global log G_1 is updated to G_2 (see predicated cmt) by labeling all the operations pushed by the current transaction as committed. This precondition is satisfied by the mapping of the commitVars call – in case of the successful commit, all the write operations specified by *W* are labeled as committed (see Definition 2). Q.E.D.

Theorem 1 below directly follows from Lemmas 1 to 4 (because the (D)PSTM push/pull semantic model satisfies preconditions of relevant push/pull rules).

Theorem 1 ((D)PSTM serializability). *(D)PSTM is serializable*.

6. CONCLUSIONS

The main original contributions of this paper that may be utilized by other researchers and practitioners are the following: (i) the (D)PSTM push/pull semantic model, (ii) the proofs of the relevant push/pull semantic rules, and (iii) the way how the model and the proofs have been constructed.

The advantages of this paper are: (i) this is the first paper that rigorously applies the methodology based on the push/pull semantic model proposed in [17] to a real STM, (ii) this paper may be interesting to other authors because there is a realistic expectation that the push/pull semantic model may be widely used for specifying transactional-based concurrent software, e.g. see [18]. The main limitation of this paper is that it only proves serializability, and it does not address liveness properties. Another limitation of the original paper on push/pull model [17], and consequently of this paper, is that the proofs were made manually.

In our near future work, we plan to apply the approach presented in this paper to other STMs and DSTMs, as well as to study liveness properties based on the push/pull model. In the more far future work, we would like to try to use some tools, such as Maude, Ott, etc., in order to create executable push/pull semantic models, and to obtain automatic proofs made by these tools. By providing automatic proofs we could provide even higher confidence in the achieved results.

Received on October 14, 2019

REFERENCES

1. M. Herlihy, J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, In: Proc. of the 20th Annual International Symposium on Computer Architecture, pp. 289–300, ACM New York, NY, USA, 1993.
2. T. Harris, J. R. Larus, R. Rajwar, *Transactional Memory, 2nd edition*, Morgan and Claypool, 2010.
3. N. Shavit, D. Touitou, *Software transactional memory*, In: Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213, ACM New York, NY, USA, 1995.
4. M. Popovic, B. Kordic, *PSTM: Python software transactional memory*, In: Proc. of the 22nd Telecommunications Forum, pp. 1106–1109, IEEE Xplore, 2014.
5. M. Goldstein, E. Fredj, B. Gerber, *A New Hybrid Algorithm for Finding the Lowest Minima of Potential Surfaces: Approach and Application to Peptides*, Journal of Computational Chemistry, **32**, pp. 1785–1800 (2011).
6. M. Amitay, M. Goldstein, *Evaluating the peptide structure prediction capabilities of a purely ab-initio method*, Protein Engineering, Design and Selection, **30**, *10*, pp. 723–727 (2017).
7. S. Stoja, S. Vukmirovic, N. Dalcekovic, D. Capko, B. Jelacic, *Accelerating Performance in Critical Topology Analysis of Distribution Management System Process by Switching from Monolithic to Microservices*, Rev. Roum. Sci. Techn. – Électrotechn. Et Énerg., **63**, *3*, pp. 338–343 (2018).
8. R.I. Mogos, C.N. Bodea, M.I. Dascalu, O. Safonkina, E. Lazarou, E.L. Trifan, I.V. Nemoianu, *Technology Enhanced Learning for Industry 4.0 Enineering Education*, Rev. Roum. Sci. Techn. – Électrotechn. Et Énerg., **64**, *4*, pp. 429–435 (2018).
9. M. Popovic, B. Kordic, I. Basicevic: *Transaction Scheduling for Software Transactional Memory*, In: Proc. of the 2nd IEEE International Conference on Cloud Computing and Big Data Analysis, pp. 191–195, IEEE Xplore, 2017.
10. M. Popovic, B. Kordic, M. Popovic, I. Basicevic, *Online Algorithms for Scheduling Transactions on Python Software Transactional Memory*, Serbian Journal of Electrical Engineering, **16**, *1*, pp. 85–104 (2019).
11. M. Herlihy, N. Shavit, *The art of multiprocessor programming, revised printing*, Morgan Kaufmann.
12. M. Popovic, B. Kordic, M. Popovic, I. Basicevic, *A Solution of Concurrent List on PSTM*, In: Proc. of the 5th International Conference on Electrical, Electronic and Computer Engineering (IcETRAN), Article RTI2.1, pp. 1–6, ETRAN, Serbia, 2018.
13. M. Popovic, B. Kordic, M. Popovic, I. Basicevic, *A Solution of Concurrent Queue on PSTM*, In: Proc. of the 25th IEEE Telecommunications Forum, pp. 735–738, IEEE Xplore, 2018.
14. M. Popovic, B. Kordic, M. Popovic, I. Basicevic, *A Solution of Concurrent Queue on Local and Distributed Python STM*, Telfor Journal, **11**, *1*, pp. 64–69 (2019).
15. B. Kordic, M. Popovic, S. Ghilezan, I. Basicevic, *An Approach to Formal Verification of Python Software Transactional Memory*, In: Proc. of the 5th European Conference on the Engineering of Computer Based Systems, Article No. 13, pp. 1–10, ACM New York, NY, USA, 2017.
16. A. Liu, M. Popovic, H. Zhu, *Formalization and Verification of the PSTM Architecture*, In: Proc. of the 24th Asia-Pacific Software Engineering Conference, pp. 427–435, IEEE Xplore, 2017.
17. E. Koskinen, M. Parkinson, *The Push/Pull Model of Transactions*, In: Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 186–195, ACM New York, NY, USA, 2015.
18. T.D. Dickerson, P. Gazzillo, M. Herlihy, E. Koskinen, *Proust: A Design Space for Highly-Concurrent Transactional Data Structures*, Cornell University Library, arXiv:1702.04866v2 [cs.DC] 26 Jun 2017 (2017).
19. C. Xu, X. Wu, H. Zhu, M. Popovic, *Modeling and Verifying Transaction Scheduling for Software Transactional Memory using CSP*. In: Proc. of the 13th Theoretical Aspects of Software Engineering Symposium, pp. 240–247, IEEE Xplore, 2019.
20. D. Dimitrov, V. Raychev, M. Vechev, E. Koskinen, *Commutativity Race Detection*, In: Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 305–315, ACM New York, NY, USA, 2014.
21. M. Popovic, M. Popovic, S. Ghilezan, B. Kordic, *Formal Verification of Python Software Transactional Memory Serializability Based on the Push/Pull Semantic Model*, In: Proc. of the 6th European Conference on the Engineering of Computer Based Systems, Article No. 6, pp. 1–8, ACM New York, NY, USA, 2019.