The Journal of Systems and Software 165 (2020) 110559

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Formal analysis and verification of the PSTM architecture using CSP

Ailun Liu[a], Huibiao Zhu[a,*], Miroslav Popovic[b], Shuangqing Xiang[a], Lei Zhang[c]

[a] Shanghai Key Laboratory of Trustworthy Computing, MOE International Joint Laboratory of Trustworthy Software, International Research Center of Trustworthy Software, East China Normal University, Shanghai 200062, China
[b] The University of Novi Sad, Serbia
[c] Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University, Shanghai 200062, China

ARTICLE INFO

Article history:
Received 17 May 2018
Revised 12 January 2020
Accepted 20 February 2020
Available online 22 February 2020

Keywords:
Transactional memory
The PSTM arhcitecture
Formal analysis and verification
Shared counter

ABSTRACT

Starting with the analysis of the source codes of the Python Software Transactional Memory (PSTM) architecture, this paper applies process algebra CSP to formally verify the architecture at a fine-grained level. We analyze the communication process and components of the architecture from multiple perspectives and establish models describing the communication behaviors of the PSTM architecture. We use model checker PAT to automatically simulate and verify the established model. After adapting the traditional transactional properties to the PSTM architecture, we analyze and verify five properties for the PSTM architecture, including deadlock freeness, atomicity, isolation, consistency and optimism. The verification results indicate that all the properties are valid. Based on the judgement of the execution logic of the communication procedure in the PSTM architecture, we can conclude that the architecture can have a proper communication and can guarantee atomicity, isolation, consistency and optimism. Besides, we also provide a case study with an application scenario and propose a corollary that the value of the shared counter is equal to the number of parallel processes. We verify whether the case study system can satisfy all the conditions of corollary from both positive and negative perspectives. The results show that the corollary is tenable.

1. Introduction

With the advent of multicore processors, parallel programming emerged. One of the mainstream solutions for parallel programming is locking. However, with the increasing popularity of high scalable applications, the cost of the locking mechanism is getting higher and higher. An alternative solution is to use atomic primitive, such as *compareAndSet*(). Most of the atomic primitives can provide good services, and multicore processor programmers have gradually constructed a variety of practical and sophisticated data structures. However, almost all synchronous atomic primitives only operate on a single word (e.g. one object, or one variable), and this restriction often results in the need to introduce an overly complex data structure in the algorithm of the program to extend the atomic operations on multiple words (e.g. an array of objects or variables). Herlihy and Shavit gave a great example of the complexity and unnaturalness when using *compareAndSet*(), see section 18.1.2 named "What is Wrong with *compareAndSet*()?" in M. Herlihy and N. Shavit (2013) (pp. 418–420). Moreover, both locks and atomic primitives like *compareAndSet*() cannot easily be composed into atomic units, see section 18.1.3 named "What is Wrong with Compositionality?" in (M. Herlihy and N. Shavit, 2013) (pp. 420–421).

At the 1993 ISCA conference, for the first time Transactional Memory (TM) was proposed in (Herlihy et al., 1993) as an extended Cache Coherence Protocol, in particular Goodman's "snoopy" protocol for a shared bus Goodman (1983). By adopting transactions instead of locks, TM solves the problems existed in the traditional parallel programming, such as priority inversion, convoying or the deadlock. By enriching the Cache Coherence Protocol, developers are able to define read or write operations that apply to multiple individual memory words, which makes atomic updates easier and provides better performance, and also improves functionalities. Moreover, transactions can be nested. Nested transactions for conditional synchronization facilitates atomic compositions across multiple calls on different objects. Transaction synchronization can be implemented by software, hardware, or both. Shavit and Touitou (1997) first proposed the concept of software transactional memory and Herlihy et al. (1993) proposed hardware transactional memory as a general multiprocessor programming model for the first time.

* Corresponding author.
E-mail addresses: ellenliu_biz@sina.com (A. Liu), hbzhu@sei.ecnu.edu.cn (H. Zhu).

https://doi.org/10.1016/j.jss.2020.110559
0164-1212/

After two decades of research and development in the area of Software Transactional Memory (STM), there have been various solutions and implementations of STM based on object-oriented programming or functional programming paradigms and languages, including C, C++, Java, Haskell etc. Many of these solutions and implementations have been widely used in the research community for emerging technologies, benchmarks and experimental evaluations such as persistent storage, instruction set extensions, etc Kordic et al. (2017). However, as one of the most popular programming languages and the basis of various parallel and distributed computing frameworks, Python still lacks an applicable and reliable software transactional memory framework.

On the other hand, the applications of TM on the market now are still database-centric, and mainly across domains of traditional trading systems and services, such as banking, trading and stock exchange. In addition to some well-known benchmarks, such as STMBench7, STAMP Minh et al. (2008), etc., there are very few practical applications in non-traditional fields, such as the complex chemical applications in pharmacy (Goldstein et al., 2011). Based on our past experience, we think that the reasons why the TM paradigm has not spread into new applications are as follows: (1) Multicore devices supporting TM have not been widely used either in academia nor in industry, and the popular programming languages such as Python do not support TMs; (2) Not all kinds of applications are suitable for such a programming model. Since delivered performance largely depends on the workload of the applications, potential users are reluctant to try new programming paradigms; (3) The system analysis and evaluation is not perfect enough, because it is difficult to effectively characterize the performance of TM applications based on limited number of benchmarks; (4) Most of the TMs are not formally verified. Therefore, potential users will question whether the TM paradigm should be used.

In order to solve the problems mentioned above, Popovic et al. proposed a novel STM of Python, Python Software Transactional Memory Popovic and Kordic (2014). They pay attention to the PSTM Popovic and Kordic (2014) architecture from the concrete implementation. We apply CSP method Hoare (1985) to analyze and model the communication behaviors of the architecture from multiple perspectives. This paper extends our conference paper Liu et al. (2017) to formally analyze and verify the architecture at a fine-grained level. By using various primitives in CSP method, the communication procedure is described precisely. We also use the model checker PAT PAT (0000) to simulate and verify the achieved model automatically. By covering all the possible traces, PAT exhausts its space to assert the validity of the given properties. We have verified five properties for the architecture, including deadlock freeness, atomicity, consistency, isolation and optimism. Another new contribution we offered in this paper is that we also give a case study with a specific scenario. We make a formal analysis and modeling of the case study, in order to verify whether the PSTM architecture could still guarantee the security for the same dictionary item. The validity of these properties shows that this architecture could guarantee some security of transactional memory. To the best of our knowledge, there is no formal verification of PSTM architecture from the perspective of above five properties.

The remainder of this paper is organized as follows. Section 2 introduces preliminary knowledge about CSP method with PAT and the PSTM architecture including its client, server, remote procedure call interface and the patterns of communication between them. In Section 3, we analyze and propose a formal model for the PSTM architecture. We implement and verify the achieved model in the model checker PAT in Section 4. Giving a case study, we apply our model in modeling and reasoning about a corollary of an application scenario of shared counter in Section 5. We discuss the related work about parallel programming, transactional memory and the PSTM architecture in Section 6 and Section 7 concludes the paper.

2. Preliminaries

In this section, we give a brief introduction to the PSTM architecture and an overview of CSP method, as well as its model checker PAT.

2.1. Overview of the PSTM architecture

The PSTM architecture is a typical client-server framework, as shown in Fig. 1. It is mainly divided into three parts - the client, the server, and the RPC interface. As an STM, the PSTM architecture consists of a set of transactions and its transactional variables (i.e. t-variables or t-vars). A single transaction can be viewed as an atomic set of instructions executing on transactional variables, requesting some services from the STM. During the execution of the program, the transactions try to add, read or perform other actions on certain items of the system dictionary by calling the API. The API includes six defined public functions which finally request services from the server through the RPC interface.

The client consists of requested transactions and API public functions. The API functions which are actually exported functions of the PSTM module accessed by the RPC interface can cover all the requirements (i.e. CRUD) raised by transactions. The API provides five various functions to update the system dictionary, including *addVars*(*q, keys*), *getVars*(*q, keys*), *cmpVars*(*q, vars*), *putVars*(*q, vars*) and *commitVars*(*q*, *read_write*). It also provides another function *delete*(*q*) to delete the system process. Among all the functions mentioned above, the parameter *q* is a queue, also is a mutex shared by the six functions, and other parameters are related to the specific functions. The parameter *keys* records the keys of the items which are requested by the transaction, and the parameter *vars* records the triples to update or commit the system dictionary. The parameter *read_write* records a read list and a write list, where the read list records the variables that the transaction can read, and the write list records the variables that the transaction can both read and write.

The PSTM_ Server component can implement the functionality offered by the exported functions of the PSTM_API component. The PSTM_Server component performs its own actions cyclically - waits for a request from the queue, executes the request, and sends a response message to the client. It is in charge of managing the system dictionary entry, which means that the system dictionary can only be accessed by the PSTM_Server. The basic unit stored in the system dictionary is an item which has the format of a triple like (*key, version, value*), where *key* is used to map the item, and *version* and *value* are used to record the current version and value of the item respectively.

Fig. 2 shows the specific architecture of the RPC interface. It handles the asynchronous requests of multiple parallel transactions in the sequence of their arriving time to guarantee the atomicity of the transactions execution. This is an extremely important role of the RPC interface. The RPC interface is abstracted from the concepts of Python 3.0 queue and pipe. With queue, it implements a many-to-one connection structure that serializes all requests sent from the clients to a server. While the server responds to the requests and returns the execution results, each client receives its own response through its own pipe, so each reply is sent back through a point-to-point connection. Therefore, every reply returned by the server is sent to the corresponding clients via these pipes. Queue ensures the correctness of parallel processing of transactions, and multiple pipelines ensure the correctness and completeness of the procedure of requesting communications.

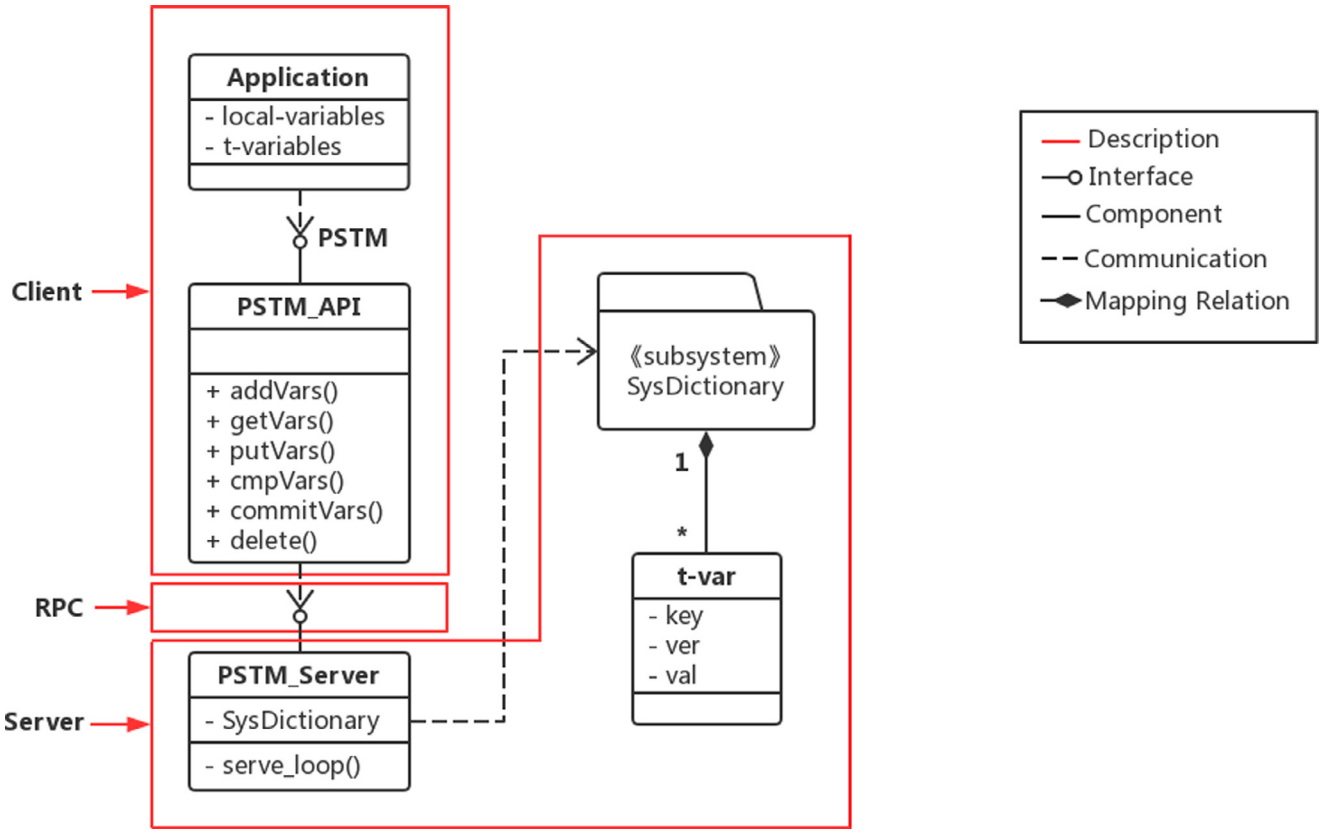

Fig. 1. The PSTM architecture.

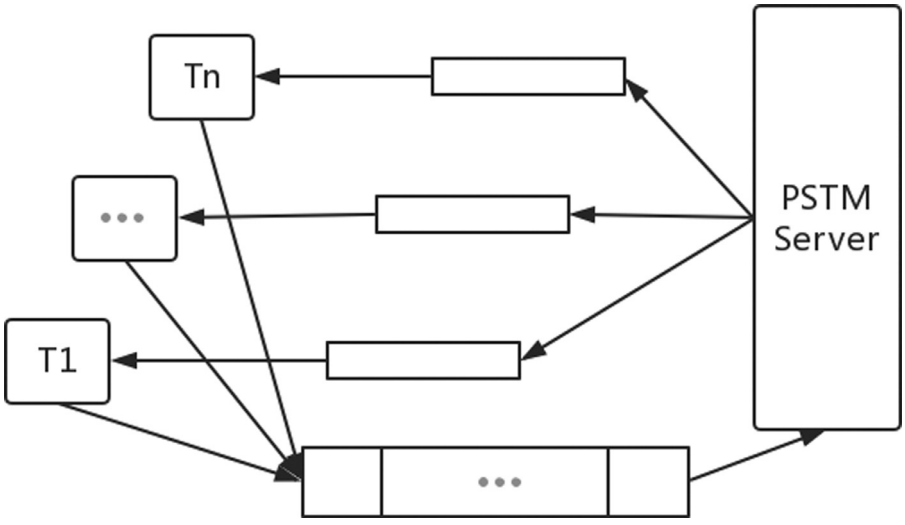

Fig. 2. The RPC interface.

2.2. CSP method and PAT

2.2.1. CSP method

As a representative of the formal methods, process algebra uses mathematical methods to study communications of the concurrent systems. By giving a visible model of a concurrent system and observing its communication procedure, we characterize the behaviours. The core thoughts behind the process algebra are to abstract a whole system into an element, and use semantic and grammatical rules to reason the behaviours of this abstract element.

In order to cope with the confusion and uncertainty brought about by the concurrency issue, Hoare proposed a type of research method, namely CSP, which enriches the theory of process algebraic calculus Hoare (1985). Compared to the other formal methods, CSP adopts mathematical theory to describe process interactions of concurrent systems and to avoid the traditional problems encountered in concurrent programming. It introduces advanced structural ideas and provides a rigorous mathematical foundation for avoidance of errors such as divergence, deadlock and nontermination, and for achievement of the provable correctness in the design and implementation of computer systems. Due to these powerful expressive abilities, CSP method has been successfully applied in many fields, such as the security protocol or system, and the web service (Lowe and Roscoe, 1997; Roscoe and Huang, 2013; Yuan et al., 2014). Since PSTM architecture is designed for parallel

processing, there will be many procedure calls at the same time. In this paper, we apply CSP method to model the communication procedure of the PSTM architecture.

CSP method uses the process as the basic unit which could involve rich operators to describe, including sending and receiving actions, specific events, or child processes. We give some frequent examples of its syntax as below, where P and Q are two processes, a and b are two events, c is a channel. For more syntax introduction, please refer to (Hoare, 1985).

$$P, Q ::= Skip|Stop|a \rightarrow P|c?x \rightarrow P|c!v \rightarrow Q|P \triangleleft b \triangleright Q$$
$$|P; Q|P \Box Q|P||Q|P|||Q|P \backslash S|P[|X|]Q$$

- *Skip* denotes that a process does nothing but terminates successfully.
- *Stop* denotes the process is in the state of deadlock and does nothing.
- $a \rightarrow P$ the process first engages in action a, then the subsequent behaviour is like P.
- $c?x \rightarrow P$ the process gets a message through the channel c and assigns it to variable x, then behaves like P.
- $c!v \rightarrow Q$ the process sends a message v using the channel c, then the behaviour is like Q.
- $P \triangleleft b \triangleright Q$ if the condition b is true, the behaviour is like P, otherwise, like Q.
- $P; Q$ the process performs P and Q sequentially.
- $P \Box Q$ the process behaves like either P or Q and the choice is made by the environment.
- $P||Q$ denotes that P runs in parallel with Q.
- $P|||Q$ indicates that P interleaves Q which means P and Q run concurrently without barrier synchronization.
- $P[|X|]Q$ indicates that processes P and Q perform the concurrent events on the set X of channels.

2.2.2. PAT

PAT is designed as an extensible and modularized framework for automatic system analysis based on CSP (Hoare, 1985). It supports specifying and verifying systems in many different modeling languages. There are already various systems such as concurrent real-time systems, probabilistic systems that have been verified in PAT (Sun et al., 2008; 0000; Si et al., 2014).

PAT can be applied in verifying various properties such as deadlock-freeness, reachability, determinateness and LTL properties with fairness assumptions in distributed systems. No matter it is a software system, a hardware system, or a hybrid system, PAT can always be used to reason and verify it. Based on the codes, PAT performs automatic simulation and provides executive outcomes of each step, which makes it easy to find problems and solve problems. It exhausts its space to find the reachable traces, by which to detect whether the achieved model satisfies certain properties. Here we list some notations.

- $\#define\ N\ 0$ defines a global constant N with the initial value 0.
- $channel\ c\ 1$ defines a channel which has the channel name c and the buffer size 1.
- $var\ cond\ =\ false$ defines a boolean condition with the initial value *false*.
- $[cond]P$ defines a guarded process, which only executes when its guard condition is satisfied.
- $\#define\ goal\ n > 0; \#assert\ P\ reaches\ goal;$ defines an assertion that checks whether process P can reach a state where the condition goal is satisfied or not.
- $|||i : \{0..N\}@P(i)$ defines N processes, including $P(1)$, $P(2)$, ..., $P(N)$, run by interleaving with each other.
- $\#assert\ P() \mid = F;$ defines an assertion that checks whether process P satisfies the formula F.

3. Formal analysis of the PSTM architecture

In this section, we try to abstract and model the architecture as detailed as possible to fully describe its characteristic. Since the RPC interface consists of queue and pipes, we also model them respectively which ensures its functionality of dealing with parallel incoming requests asynchronously. Unlike some horizontal modelings in other architectures, we model the architecture vertically, which is also the challenge of this paper. We model all the methods of the API including their internal calculation process. Based on these calculation process modelings and their results, we could verify if the PSTM architecture satisfies some properties.

We formalize the PSTM architecture in CSP. We present the modeling of messages and channels firstly. Then we continue by modeling the PSTM architecture. The formalization is proceeded based on the PSTM architecture which has been described in Section 2.

3.1. Modeling messages and channels

We assume the existence of the sets *Key, Version* and *Value* of all the passing keys, versions and values referred in our model respectively. We define the sets *Operator, Content* of the functions (i.e. functions that appear in six APIs) that the incoming transactions intend to perform, and the execution outcomes (i.e. a list of pairs or a list of **true**/**false**) intend to return. The set *Queue* is also defined of all the mutexes shared by functions. Besides, we define three other sets *dicKey, dicVersion* and *dicValue* of the keys, versions and values of the items stored in the system dictionary. For defining each item of the system dictionary, the set *DicItem* has the following general format:

$$dicItem =_{df} \{dickey, dicversion, dicvalue | dickey \in dicKey,$$
$$dicversion \in dicVersion, dicvalue \in dicValue\}$$

The messages include two types, which are the sent message MSG_{req} and the reply message MSG_{resp}. MSG_{req} represents multiple transactional request messages. From formula (3.1.1.2), we know that each transactional request message MSG_{req_i} consists of one or more small messages, where the number m of small messages is determined by the transaction requests. Each small message comprises five parts, namely an *operator* of the specific operation to perform, a *queue* of the mutex shared between six functions in each execution, a *key, version* and *value* of the key, version and value of the variable pending to be operated. Each part except the *version* and the *vaue* is not allowed to be empty.

Likewise, the reply message MSG_{resp} contains multiple transaction reply messages. From formula (3.1.1.1), we know that each transactional reply message MSG_{resp_i} also consists of one or more small messages. The number of small messages sent and received is the same. We define these small reply messages in five parts as well, including the outcome *content*, the mutex *queue* shared between six functions, the key of the corresponding variable *key*, and the updated version and value after execution *version, value*. The last two parts are allowed to be empty (only the parameter *content* must have a value, such as *addVars, getVars*, depending on the return structure of different API public functions).

$$MSG =_{df} MSG_{req} \cup MSG_{resp}, MSG_{req} =_{df} \bigcup_{i=1}^{m} MSG_{req_i},$$
$$MSG_{resp} =_{df} \bigcup_{i=1}^{m} MSG_{resp_i} \quad \ldots (3.1.1)$$
$$MSG_{req_i} =_{df} \{operator.queue.key.version.value | operator$$
$$\in Operator, queue \in Queue,$$
$$key \in Key, version \in Version, value \in Value\}, \quad \ldots (3.1.1.1)$$

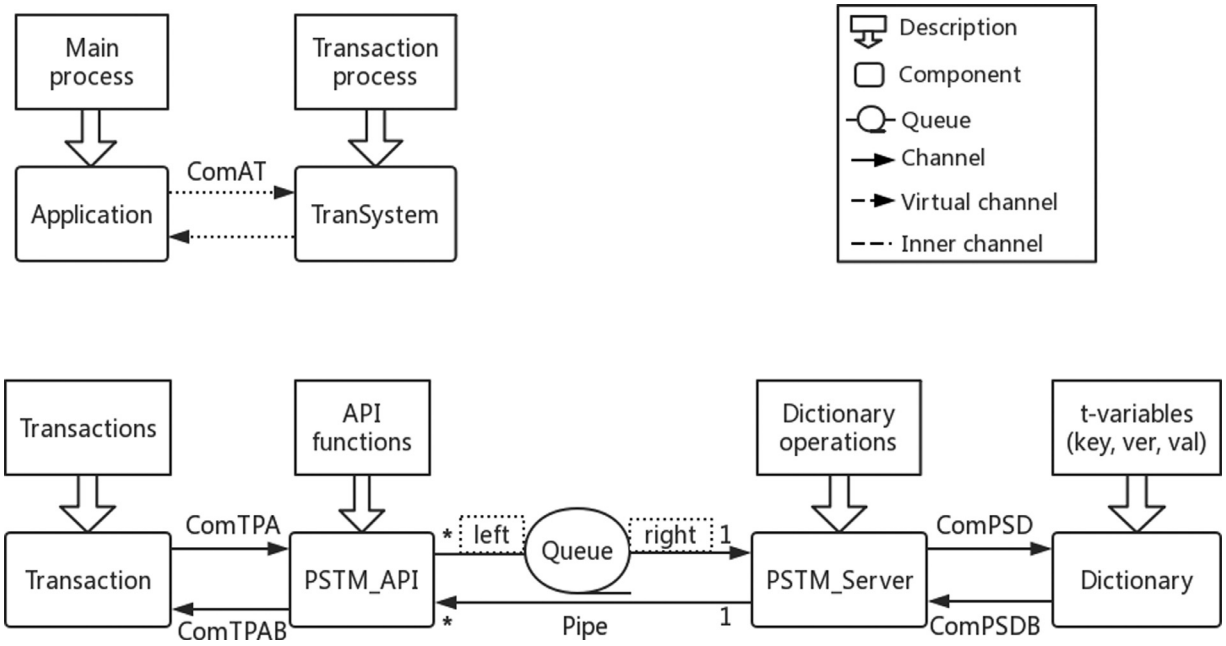

Fig. 3. The Communication Model of the PSTM Architecture.

$$MSG_{resp_i} =_{df} \{content.queue.key.version.value | content \in Content, queue \in Queue, key \in Key, version \in Version, value \in Value\}. \quad \ldots (3.1.1.2)$$

Next, we define four sets of channels to be used in the communication procedure of the PSTM architecture as shown in Fig. 3. Here, we use dotted lines to describe invisible channels, which means they are conceptual or inner channels. Solid lines are used to describe visible channels, and the capital 'Q' shape with two lines on both sides describes a queue. Blocks represent the components of the PSTM architecture, and boxes with arrows give the blocks explanation. Each set of channels helps different pair of components to communicate. We give their detailed descriptions as follows.

1. Channel *ComAT* is a virtual channel, indicated by dotted lines. *A* is short for Application component, and *T* is short for TranSystem component given in the Fig. 3. This channel is used to build a general communication process between the client and the server. The specific communication process will be revealed by the next three sets of channels.
2. Channel *ComTPA* and channel *ComTPAB* are used to build a procedure of transaction requests calling API public functions to obtain the execution result. *T* is short for TranSystem component, and *PA* for *PSTM_API* component. Through channel *ComTPA*, the request sent by the client (i.e., the transaction request containing the t-variable) is transmitted to the *PSTM_API* component, and the specific API function to be executed (i.e., six options such as add, get, or commit) is selected. Through the channel *ComTPAB*, the system receives the return outcomes after the execution. We define the channel *ComTPAB* as a collection of auxiliary channels which return reply messages received from the set of channel *Pipe*, any of which can be listed as $ComTPAB_i$. Specific implementations are given in the following section.
3. Channel *Pipe* is a collection of channels defined to build a procedure of a single server working as a loop to reply to multiple API requests, any of which can be listed as $Pipe_i$, connected to each corresponding $Transaction_i$. Here, *PA* is short for *PSTM_API* component, and *PS* for *PSTM_Server* component. This collection is implemented by the pipeline mechanism of Python 3.0. The task handled by each pipeline is to return a response and their overall performance is what channel *Pipe* achieves. The server processes each request in sequence according to their arrived order, and sends the returned execution result to the corresponding request through the channel *Pipe*.
4. Channel *ComPSD* and channel *ComPSDB* are used to build a procedure of the communication process between the server and the system dictionary. Similarly, *PS* is short for *PSTM_Server* component, and *D* for *Dictionary* component. During this communication, the server *PSTM_Server* reads, compares, or performs other operations on the corresponding items stored in the system dictionary according to the passing parameter *operator* through the channel *ComPSD*. The execution result is sent back through the channel *ComPSDB*.

As shown in Fig. 3, we abstract four components from the PSTM architecture, namely *Transaction* of transaction request, *PSTM_API* of the exported API public function part of the PSTM module, *PSTM_Server* of the server part of the PSTM module and *Dictionary* of the system dictionary. *Transaction* contains a number of pending transaction requests. *PSTM_API* provides six different calling services, and *PSTM_Server* manages the entry of the system dictionary. All communication procedures that may occur between components of the PSTM architecture are given in Fig. 3. Components are drawn in round rectangles and their descriptions or stored items are given inside the rectangles with arrows pointing to them. We build the model of channels as follows:

Channel *ComAT*,
Channel *ComTPA, ComTPAB, Pipe, ComPSD, ComPSDB*: MSG.

3.2. Modeling RPC interface

The RPC interface consists of two parts: queue and pipeline, which are abstracted from the concepts of queue and pipeline in Python 3.0. The queue is responsible for continuously receiving transaction requests and sorting them according to their arrival time. Requests arriving first are queued firstly, requests arriving

later are queued later, and if multiple requests arrive at exactly the same time, a sequential queue is randomly allocated. After entering the queue, the server uses the first-in-first-out rule to process the requests in turn. When the request is processed, the pipeline is responsible for sending the return result to the corresponding client. Each client connects the server side with a pipe through which the return result of the transaction request is received. When that client receives the response to the request from the server, a complete communication process ends. Next, we start modeling the RPC interface from the queue structure.

In order to model the queue concisely, we build two auxiliary channels. Channel *left* is defined to get messages from the outside to the queue, and channel *right* is defined to output the message in the queue to the outside.

Channel *left, right*: MSG.

Next, we build a queue with a storage buffer:

$$Queue() = que_{<>},$$

Here,

$$que_{<>} = left?MSG_{req}1 \rightarrow que_{<MSG_{req}1>},$$
$$que_{<MSG_{req}1>^\frown s} = left?MSG_{req}2 \rightarrow que_{<MSG_{req}1>^\frown s^\frown <MSG_{req}2>} \mid right!MSG_{req}1 \rightarrow que_s.$$

The process *Queue*() is always ready to receive messages from channel *left* and store them in a buffer. When the queue is no longer empty, two actions may occur: the first possibility is to continue to receive other messages and store them after the stored messages; and the other possibility is to output the first message that has been received but not yet output from channel *right* and delete it from the buffer. The environment chooses which action the queue would perform. Since the PSTM architecture does not mention the buffer's constraint, we would not limit the buffer size for the time being. So far we have modeled many-to-one functionality of the RPC interface. The auxiliary channels need to be integrated with the system when modeling the whole architecture. Please refer to the following section for detailed implementation.

Next, we adopt both the channel set *Pipe* and the channel set *ComTPAB* to realize the function of the pipeline. Each channel $Pipe_i$ and each channel $ComTPAB_i$ are distinguished by the identifier i of the transaction request and are associated with the $Transaction_i$ that issued the request. The communication process on channel *Pipe* involves two components, the *PSTM_API* and the *PSTM_Server* in the PSTM module. We define the message sets for all possible communications of the *PSTM_API* and *PSTM_Server* on channel $Pipe_i$, the *Transaction* and the *PSTM_API* on channel $ComTPAB_i$ as:

$$\alpha ComTPAB(Transaction, PSTM\_API) = \{MSG_{resp} \mid ComTPAB.MSG_{resp} \in \alpha(Transaction, PSTM\_API)\},$$
$$\alpha Pipe(PSTM\_API, PSTM\_Server) = \{MSG_{resp} \mid Pipe.MSG_{resp} \in \alpha(PSTM\_API, PSTM\_Server)\}.$$

Here, only the requesting transaction can receive its reply message, which means the set of all communication messages that may occur on each channel $Pipe_i$ and $ComTPAB_i$ respectively, are given as follows.

$$\alpha ComTPAB_i(Transaction, PSTM\_API) = \{MSG_{resp_i} \mid ComTPAB_i.MSG_{resp_i} \in \alpha(Transaction, PSTM\_API)\},$$
$$\alpha Pipe_i(PSTM\_API, PSTM\_Server) = \{MSG_{resp_i} \mid Pipe_i.MSG_{resp_i} \in \alpha(PSTM\_API, PSTM\_Server)\},$$

For each transaction request, the *PSTM_Server* sends a reply message to the exported API component in the module through the channel $Pipe_i$, and after receiving the message, the exported API component continues to return the reply message to the requesting transaction $Transaction_i$ through the channel $ComTPAB_i$. This part of the communication process can be characterized as:

$$Pipe_i!MSG_{resp_i}$$
$$Pipe_i?MSG_{resp_i} \rightarrow ComTPAB_i!MSG_{resp_i}$$

So far, we have implemented several one-to-one pipelines of the RPC interface. In the next section, we will give a complete formal analysis and modeling of the communication process between components in the PSTM architecture.

3.3. Modeling the PSTM architecture

As shown in Fig. 3, we build a two-layer communication model system for the PSTM architecture to model it more clearly. As a Client-Server structure, the first layer model depicts the communication between the transaction request sent from the client and the transaction processing system of the server. This layer is a conceptual model, which only describes the supply and demand relationship in the PSTM architecture system and does not contain specific communication detail. This layer includes the process *Application*(), which sends out application requests continuously, and the process *Transystem*(), which sends out responses as soon as it receives requests. We build this layer of model first.

The process *Application*() and the process *Transystem*() communicate with each other through the channel *ComAT*, where the *Application*() can start sending a new request message without waiting for a response of the previous request. The *Transystem*() loops itself to process incoming requests continuously and return executive outcomes.

$$Application() =_{df} ComAT!Msg_{req} \rightarrow ComAT?Msg_{resp} \rightarrow Application(),$$
$$TranSystem() =_{df} ComAT?Msg_{req} \rightarrow ComAT!Msg_{resp} \rightarrow TranSystem(),$$

Then, the first layer model is:

$$Application() \ [|\ ComAT\ |]\ TranSystem().$$

Then, we build the second layer of model to extend the first layer and describe the communication process of the PSTM architecture system in detail. This layer describes the communication procedure between four components in the PSTM architecture system, including the process *Transaction*(), the process *PSTM_API*(), the process *PSTM_Server*() and the process *Dictionary*(), respectively. Among these four processes, the process *Transaction*() enriches the process *Application*() and the remaining three processes enriches the process *TranSystem*(). We model them separately.

3.3.1. Modeling the process Transaction()

The process *Transaction*() characterizes the procedure of various transactions sending requests to the server in parallel. Each $Transaction(i)$ has a corresponding request message MSG_{req_i} and a reply message MSG_{resp_i}, it sends the request message through channel *ComTPA* and receives the reply message through channel $ComTPAB_i$. At this point, a process $Transaction(i)$ execution completes to terminate normally. When all parallel transaction requests are executed, the process $Transaction(i)$ ends. For the sake of brevity, we will simply refer to the collection of all transaction variables to be manipulated as vars. We model the behavior of the process *Transaction*() as follows:

$$Transaction() =_{df} ||_{i \in vars}\ Transaction(i);$$
$$Transaction(i) =_{df} ComTPA!MSG_{req_i} \rightarrow ComTPAB_i?MSG_{resp_i} \rightarrow Skip,$$

3.3.2. Modeling the process PSTM_API()

After receiving the first transaction request, the process *PSTM_API*() begins to send a message to the process *Queue*() via channel *left*. The process *Queue*() sorts these messages in the order of their arrival and always outputs the first entered message continuously through the internal channel *right*. The process

PSTM_API() receives the reply messages processed by the server through channel *Pipe* and continues to return the response to the corresponding transaction request through channel *ComTPAB*. Because the process *PSTM_API*() is always serving the process *Transaction*() for processing transaction requests, its behaviors need to be cycled repeatedly, waiting for the request message to be received from channel *ComTPA*. Also as its first action is a receiving action, so no following actions could be performed until it got messages in the queue, therefore no deadlock occurs.

We define the selection and calling procedure of API public functions as a subprocess *Assort*(). The subprocess *Assort*() is actually executed only when interaction with the system dictionary is considered. There is also no concurrency problem because the child process does not have the same actions with other processes. Therefore, we put this part into the modeling of the system dictionary process *Dictionary*(), which also facilitates the implementation of subsequent PAT coding and automatic simulation. We establish the behavior model of the *PSTM_API*() process as follows:

$$\begin{aligned} PSTM\_API() =_{df}\ & ComTPA?MSG_{req} \rightarrow left!MSG_{req} \\ & \rightarrow Queue(MSG_{req}) \rightarrow Pipe?MSG_{resp} \\ & \rightarrow ComTPAB!MSG_{resp} \rightarrow PSTM\_API(), \end{aligned}$$

3.3.3. Modeling the process PSTM_Server()

The process *PSTM_Server*() receives the transaction requests which need to be processed sequentially from auxiliary channel *right*. The entry management of the server to system dictionary is realized by sending requests to the system dictionary through channel *ComPSD*, and receiving the responses through the channel *ComPSDB*. After receiving the reply message MSG_{resp}, there is no need of the process *PSTM_Server*() to identify and allocate the messages autonomously, but directly to return it to the process *PSTM_API*() through the channel *Pipe*. The channel *Pipe* can filter them according to the transaction identifier *i* carried as a parameter in the messages, and select the accurate channel $Pipe_i$ to send the return results to the corresponding transaction request so as to play the role of point-to-point communication of the pipeline.

Similar to the definition of the process *PSTM_API*(), as the server listens for client requests, this process still needs to be defined as a circular process. As the first action is a receiving action, so there will be no deadlock either. We establish the behavior model of the process *PSTM_Server*() as follows:

$$\begin{aligned} PSTM\_Server() =_{df}\ & right?MSG_{req} \rightarrow ComPSD!MSG_{req} \\ & \rightarrow ComPSDB?MSG_{resp} \rightarrow Pipe!MSG_{resp} \\ & \rightarrow PSTM\_Server(), \end{aligned}$$

3.3.4. Modeling the process Dictionary*()*

The process *Dictionary*() receives a transaction request through the channel *ComPSD*. A corresponding dictionary item is added, read, or updated based on the operator carried in the message. We add a process *Assort*() here to describe the procedure of assigning the API public functions according to various given operators. After the called function is executed, the returned execution outcome is sent over the channel *ComPSDB*. Similarly, this process still needs to be defined as a circular process without deadlock. We model the behavior of the process *Dictionary*() as follows:

$$\begin{aligned} Dictionary() =_{df}\ & ComPSD?MSG_{req} \rightarrow Assort(MSG_{req}) \\ & \rightarrow ComPSDB!MSG_{resp} \rightarrow Dictionary(), \end{aligned}$$

The process *Assort*() sequentially calls the corresponding API public functions based on the operators (e.g. one or more) of the incoming message. When receiving the incoming message, it first checks if the message MSG_{req} is empty, and if it is empty, it will end the process. If not empty, the process *RunAssort*() will be called to start the selection. Each time the process *RunAssort*() reads the first small message in the MSG_{req}, we mark the remaining small messages as $MSG_{req}{}'$. If the operator in the message does not match any of the six keywords, an error occurs and *Stop* is executed, then the process terminates. If the corresponding keyword is found, the specified API public function is called to operate the system dictionary. The above operation is then repeated starting with the first small message of the remaining message. With all small messages executed, $MSG_{req}{}'$ would be empty and the process ends normally. When the incoming operator is *delete*, the transaction request tries to terminate all the communications and the process ends normally.

$$\begin{aligned} & Assort(MSG_{req}) =_{df} RunAssort(MSG_{req}) \lhd (MSG_{req}! = NULL) \rhd Skip, \\ & RunAssort(MSG_{req}) =_{df} \\ & if(operator == add)\ then\ \left(addVars(keys); Assort(MSG_{req}{}')\right) \\ & else\ if(operator == get)\ then\ \left(getVars(keys); Assort(MSG_{req}{}')\right) \\ & else\ if(operator == put)\ then\ \left(putVars(vars); Assort(MSG_{req}{}')\right) \\ & else\ if(operator == cmp)\ then\ \left(cmpVars(vars); Assort(MSG_{req}{}')\right) \\ & else\ if(operator == cmt)\ then\ \left(cmtVars(vars); Assort(MSG_{req}{}')\right) \\ & else\ if(operator == dlt)\ then\ (delete(); Skip) \\ & else\ (Stop), \end{aligned}$$

We have all the functions other than the *delete*() function to input the transaction identifier *i* and one or more elements of the triplets from each small message in the passing parameter MSG_{req}, and to return the corresponding execution result. The transaction identifier *i* is designed to specify the pipeline index number when sending execution result. As for the functional implementation of each function, a detailed description of these API public functions is given in the technical background of Section 2. In order to guarantee the atomic operation of function execution, we define all the actions of each function execution in events. In CSP method, all the actions within an event are transient or atomic by default. Each function processes the parameter list sequentially, we mark the remaining parameters as $keys'$, or $vars'$, or $readVars'$, $writeVars'$. Next, the formal models of six API public functions are given in turn.

1. The function *addVars*(*i, keys*) adds a new variable to the system dictionary. For each parameter *key*, the system dictionary adds a new item that holds a key *key*, a version 0, and the value is *NULL*. When an existing item in the dictionary is added repeatedly, it is not considered as an error and the operation is ignored directly. Therefore, this method is always successful and the boolean condition *content* is always set to the value *true*.

$$\begin{aligned} addVars(i, keys) =_{df}\ & Skip \lhd (keys == NULL) \\ & \rhd addKey\{set\_number(i); set\_key(key); \\ & set\_content(true); set\_version(0); \} \\ & \rightarrow addVars(i, keys'), \end{aligned}$$

2. The function *getVars*(*i, keys*) searches the set of keys *dicKey* in the system dictionary based on each passing *key*. If the keys can be found, the returned result *content* is set to the value *true*, then this function reads the corresponding version as *newVersion*, corresponding value as *newValue*. We introduce this situation in formula (3.3.4.1.1). Otherwise, *content* is set to the value *false*, shown in formula (3.3.4.1.2).

$$getVars(i, keys) =_{df} Skip \lhd (keys == NULL) \rhd \left(\begin{array}{ll} \left(\begin{array}{c} getVariable\{set\_number(i); set\_content(true); \\ get\_version(newVersion); get\_value(newValue); \} \rightarrow getVars(i, keys') \end{array} \right) & \ldots(3.3.4.1.1) \\ \lhd (key \in dicKey) \rhd & \\ getError\{set\_number(i); set\_content(false); \} \rightarrow getVars(i, keys') & \ldots(3.3.4.1.2) \end{array} \right) \ldots(3.3.4.1),$$

3. The function *cmpVars*(*i, vars*) searches the system dictionary based on the passing parameter *vars*. For each parameter *var*, it checks if there are any dictionary items holding *key* as the key in the triplet, and *version* as the version. As shown in formula (3.3.4.2.1), if the results match, the returned result *content* is set to the value *true*. Otherwise, *content* is set to the value *false* shown in formula (3.3.4.2.2).

$$cmpVars(i, vars) =_{df} Skip \lhd (vars == NULL) \rhd \left(\begin{array}{ll} cmpVariable\{set\_number(i); set\_content(true); \} \rightarrow cmpVars(i, vars') & \ldots(3.3.4.2.1) \\ \lhd (key \in dicKey \ \&\& \ version \in dicVersion \ \&\& \ var \in DicItem) \rhd & \\ cmpError\{set\_number(i); set\_content(false); \} \rightarrow cmpVars(i, vars') & \ldots(3.3.4.2.2) \end{array} \right) \ldots(3.3.4.2),$$

4. The function *putVars*(*i, vars*) attempts to update a system dictionary item. For each *var* in the parameter *vars*, it compares whether the key and the version of the dictionary entry matches the key and the version of the transaction variable. If so, as shown in formula (3.3.4.3.1), it uses the *value* as the variable value of *var* to update the stored value in the system dictionary, then increments the version, and sets the returned result *content* to the value *true*. Otherwise, *content* is set to the value *false*, shown in formula (3.3.4.3.2). This function is considered successful only if the keys and versions of all transaction variables in the parameter list match and it has executed all the operations successfully. Otherwise, as long as there is one transaction variable not matching or any partial operations fail to execute, the entire function fails.

$$putVars(i, vars) =_{df} Skip \lhd (vars == NULL) \rhd \left(\begin{array}{ll} \left(\begin{array}{c} putVariable\{set\_number(i); set\_content(true); \\ set\_version(dicVersion + 1); \\ set\_value(value); \} \rightarrow putVars(i, vars') \end{array} \right) & \ldots(3.3.4.3.1) \\ \lhd (key \in dicKey \ \&\& \ version \in dicVersion \ \&\& \ var \in DicItem) \rhd & \\ putError\{set\_number(i); setContent(false); \} \rightarrow Skip) & \ldots(3.3.4.3.2) \end{array} \right) \ldots(3.3.4.3),$$

5. The function *cmtVars*(*i, readVars, writeVars*) attempts to commit a transaction on a set of transaction variables that were read (which allowed to be empty) and a set of transaction variables that were updated (which allowed to be empty). The function first compares the keys and versions of the transaction variables in the parameter *readVars* and *writeVars* with the keys and versions of the corresponding items in the dictionary. If all of them match with each other, it uses the value of the transaction variable in *writeVars* to update the item, increments the version, and submits the execution result. Then, it sets the return result *content* to the value *true*, shown in formula (3.3.4.4.1). Otherwise, *content* is set to the value *false*, shown in formula (3.3.4.4.2). This function has no rollback operation, and if the first commit fails, it will just repeat to commit until its is successful. Here we use the prefix *read* or *write* to differentiate elements of different lists.

$$\begin{array}{l} cmtVars(i, readVars, writeVars) =_{df} Skip \\ \quad \lhd (readVars == NULL \ \&\& \ writeVars == NULL) \rhd \\ \left(\begin{array}{ll} \left(\begin{array}{c} cmtVariable\{set\_number(i); set\_content(true); get\_key(key); \\ set\_version(dicVersion + 1); set\_value(writeValue); \} \rightarrow \\ cmtVars(i, readVars', writeVars') \end{array} \right) & \ldots(3.3.4.4.1) \\ \lhd (readKey \in dicKey \ \&\& \ readVersion \in dicVersion \ \&\& & \\ \quad readVar \in DicItem \ \&\& \ writeKey \in dicKey \ \&\& & \\ \quad writeVersion \in dicVersion \ \&\& \ writeVar \in DicItem) \rhd & \\ \left(\begin{array}{c} cmtError\{set\_number(i); setContent(false); \} \rightarrow \\ cmtVars(i, readVars', writeVars')) \end{array} \right) & \ldots(3.3.4.4.2) \end{array} \right) \ldots(3.3.4.4), \end{array}$$

6. The function *delete*() attempts to delete the process to stop the communication. The operation object of this method is the process, not some items in the system dictionary. It always sets the return result *content* to true expressing that the process has been deleted, and communication ends.

$$delete() =_{df} join\{set\_content(true); \} \rightarrow Skip,$$

3.4. Modeling the PSTM architecture

After the above four groups of models are given respectively, the second layer communication, namely the PSTM architecture system model, can be modeled as a combination of these four groups of processes. Through parallel communication between three sets of channels, the system model is built as follows:

$$TranSystem() =_{df} Transaction() [|\ ComTPA, ComTPAB\ |] PSTM\_API() [|\ Pipe\ |] PSTM\_Server() [|\ ComPSD, ComPSDB\ |] Dictionary().$$

4. Verification

In this section, by encoding the CSP model into the model checker PAT, we analyze and verify five properties of the PSTM architecture and give the verification results for each.

4.1. The analysis of five properties

As the concept of transactional memory originates from transaction operations Gray and Reuter (1993) in the database, the reliable transaction operations in the database need four basic transaction properties called ACID (i.e. Atomicity, Consistency, Isolation, Durability) to guarantee the correctness of execution. This paper follows this idea to verify the PSTM architecture with five properties including deadlock freeness, atomicity, consistency, isolation and optimism. Among them, the property of durability mainly concerns with the persistence of changes made to the database after the transaction is completed. Even in the event of a sudden failure of the database system, such as a power failure, it is possible to restore to the previous consistent state. The guarantee of this property depends on the system log, which is not related to the communication process or the structure of the PSTM architecture, so in this paper we would not verify the property of durability. Alternatively, as the PSTM architecture has added a significant feature, that is, the version recording of the dictionary items, that coincides with the idea of optimistic/pessimistic concurrency property. We got enlightened from the pending commit property of (Guerraoui et al., 2005). As we all know, optimism property is also related to some versions ensuring in concurrent programming. Hence this paper adds the verification of the optimism to verify whether the version recording can play a role in guaranteeing the correctness of reading and writing the dictionary items.

Besides the PSTM architecture system process, we also implement three communication procedures to assist in verifying properties. The first procedure *Scenario*1() describes a group of requests (i.e. updating after adding) interact with the PSTM architecture. Atomicity means all or nothing. We verify the satisfaction of atomicity property with *Scenario*1(), that is, if conditions are matched, the dictionary would be updated. We also verify the satisfaction of consistency property with *Scenario*1() as well, which verifies if the dictionary items would always satisfy some rules before or after the executions. The second procedure *Scenario*2() describes two groups of requests interact with the PSTM architecture. We verify optimism property with *Scenario*2(), which verifies multiple transactions would work in parallel and make no influences on others until at least one of the transactions would succeed. Both of the groups request to add, update and commit sequentially, only the operation objects are different. The third procedure *Scenario*3() is similar to *Scenario*2(). The difference is that the first set of transactions in *Scenario*3() is to add, fetch, update, commit and fetch again while the second set of transactions remain unchanged. We verify isolation property with *Scenario*3(), which verifies if the dictionary item would change between multiple reads. Next, we verify the properties starting from the deadlock freeness.

4.2. Verification

Deadlock freeness. We verify whether the system process and the three communication processes mentioned above would have deadlock. Deadlock indicates such a situation in which there is no state having any further moves except waiting for resources occupied by other processes. Neither the system process nor the three communication processes should be stuck in a deadlock, and we verify this property by PAT. If the results are valid, all the processes can have a proper communication. Otherwise, there are problems existed in the communications.

$$\#assert\ PSTM()\ deadlockfree;\ \#assert\ Senario1()\ deadlockfree;$$
$$\#assert\ Senario2()\ deadlockfree;\ \#assert\ Senario3()\ deadlockfree;$$

Isolation. We verify this property by observing the third set of processes, *Senario*3(). As described in the first subsection of this section, the main reason leading to the inability to meet the isolation is the read inconsistency, in the sense that the same transaction has multiple accesses to the same variable, and the value is updated between multiple accesses. The process *Senario*3() just describes the situation like this, where the function *putVars*() and *commitVars*() are called to execute between two accesses. We add a conditional judgment in the function *getVars*() to give the flag variable *mark* the value *true* when the same dictionary entry is repeatedly read while its version remains unchanged. Otherwise, the flag variable *mark* remains the value *false*. We give a verification goal *enterTwice* here to describe the situation of $mark == true$. If *Senario*3() can reach the goal *enterTwice* after verification, then there will be a problem of duplicate reads, otherwise the PSTM architecture will not have this problem, which means the isolation property can be guaranteed.

$$\#define\ enterTwice\ (mark == true);$$
$$\#assert\ Senario3()\ reaches\ enterTwice;$$

Atomicity. We verify the atomicity property by observing the first communication procedure *Senario*1(). This process describes a scenario in which the data is updated once a dictionary entry has been added to the system dictionary. We add an array *test*[] to the function *putVars*(), and set *test*[*key*] to value 1 when key and version of both the inputting transaction variable and the dictionary item are matched. We define a proposition *start* to describe the scenario after comparison, and another proposition *end* to describe the scenario after execution where *putRunning*[*key*] equaling to 1. We define the verification goal *collusion* to describe the data updating is still in the process, with *putRunning*[*key*] remains at 0.

$$\#define\ start\ (xor\ a : \{0..N-1\}\ @(test[a] == 1));$$
$$\#define\ end\ (xor\ a : \{0..N-1\}\ @(putRunnning[a] == 1))$$
$$\#define\ collusion\ (\&\&\ a : \{0..N-1\}\ @(putRunning[a] == 0);$$
$$\#define\ Atomicity\ (start\ \&\&\ end)$$
$$\#define\ BreakAtomicity\ (start\ \&\&\ collusion);$$

We give the verification goals *Atomicity* and *BreakAtomicity*, in which the reachability of *Atomicity* requires that both the proposition *start* and *end* can be reached while the reachability of *BreakAtomicity* requires that both the proposition *start* and *collusion* can be reached. If the process *Senario*1() can only reach the verification goal *Atomicity*, and cannot reach the goal *BreakAtomicity*, then the PSTM architecture can guarantee the property of atomicity.

$$\#assert\ Senario1()\ reaches\ Atomicity;$$
$$\#assert\ Senario1()\ reaches\ BreakAtomicity;$$

Consistency. We verify the property of consistency also by observing the first communication procedure *Senario*1(), that is, whether the constraint can always be satisfied in the PSTM architecture. We give a verification goal *Snapshot*, indicating the scene where as long as the related variables involved in a constraint are added to the system dictionary, their versions can only appear to:

(1) Always be the same (now the execution is completed); (2) have one unit value as the difference (now the execution is being performed). Here, we define the last two items in the dictionary are involved in the constraint. Hence, as long as one item has updated its value, the value of the other item needs to be updated accordingly.

$$\#define\ Snapshot\ ((||\ e : \{-1, 0, 1\}@(dic[1][1]==dic[2][1]+e)) \\ ||\ (||\ k : \{1, 2\}@(dic[k][0] == 1000)));$$
$$\#assert\ Senario1() \mid =\ G\ Snapshot;$$

Alternately, we can verify the property from another point of view. We define two propositions *xchange* and *ychange*, any of which describes that there is one item of the two has been updated. Then we can use the next operator in logic (i.e. X) to complete the verification. if the first or the second item has been updated, then the second or the first item needs to be updated accordingly at the next moment. If the process *Senario*1() verifies that all the verification goals can be reached, then the PSTM architecture can guarantee the consistency property.

$$\#define\ xchange\ (||\ v : \{0..N-1\}\ @(dic[1][1] \\ == v\ \&\&\ dic[2][1]! = 1000));$$
$$\#define\ ychange\ (||\ v : \{0..N-1\}\ @(dic[2][1] \\ == v\ \&\&\ dic[1][1]! = 1000));$$
$$\#assert\ Senario1() \mid =\ [\ ]\ (xchange -> X\ ychange);$$
$$\#assert\ Senario1() \mid =\ [\ ]\ (ychange -> X\ xchange);$$

Optimism. We verify the optimism by observing the second communication procedure *Senario*2(). Optimism ensures that multiple transactions can execute in parallel without interfering with each other, and at least one of the transactions can succeed. The process *Senario*2() describes a scenario where two requests holding the same read lists and different write lists, trying to call the function *CommitVars*() to operate the same dictionary item. We give the verification goal *AtLeastOne*, which describes at least one of the two requests can successfully execute the function *CommitVars*(). Meanwhile, we define another verification goal *AllFalse*, which describes neither of the two requests can be successfully executed. The numbers given in the verification goal are matched with the input of the system, and all the numbers input into the system are given randomly. We are concerned about whether the system would meet the specified status under the current setting. We can adopt mathematical induction to verify that no matter what the current input is, the verification results always keep the same. If the process *Senario*2() verifies that *AtLeastOne* can be reached and *AllFalse* can not be reached, then the PSTM architecture can guarantee the optimism.

$$\#define\ AtLeastOne\ (dic[2][1] == 2\ \&\&\ (dic[2][2] \\ == 85\ ||\ dic[2][2] == 50));$$
$$\#define\ AllFalse\ ((dic[2][1] >= 3\ \&\&\ dic[2][1] < 1000) \\ ||\ (dic[2][1] == 2\ \&\& \\ (dic[2][2]! = 85\ \&\&\ dic[2][2]! = 50)));$$
$$\#assert\ Senario2()\ reaches\ AtLeastOne;$$
$$\#assert\ Senario2()\ reaches\ AllFalse;$$

Verification results. Fig. 4 shows the verification results and the traces found by PAT. Among them, the first four verification goals are valid proving the property of deadlock freeness of the PSTM architecture and the three communication processes. The fifth verification goal is not valid, proving that there is no read inconsistency of the PSTM architecture when processing a request with two times reading of the same dictionary item. While the sixth verification goal is valid and the seventh is not valid, which proves the atomicity of the PSTM architecture. The eighth to tenth properties are valid, proving the consistency of the PSTM architecture. And the validity of the eleventh to twelfth properties proves that the PSTM architecture can guarantee the optimism. The model checker runs on the machine with core i5 and 16G memory. It takes 5 min and 300 MB to verify the most complex property.

Fig. 4. The verification results of five properties.

5. Case study

We map the model built in Section 3 and the PAT process encoded in Section 4 to a case study with a specific scenario in this section. Based on the functionality realized by the case study and its Python source code, we formally analyze and model this case study, verifying that the PSTM architecture still guarantees the security for the same dictionary item.

5.1. The analysis of the case study

First, we introduce the concept of a shared counter. This shared counter is a variable that can be accessed and operated by multiple parallel processes. To the PSTM architecture system, the shared counter is an item stored in the system dictionary. Each process performs the same operation on this item, which means reading the current value, incrementing it, and committing it. As shown in Fig. 4, this case study consists of N processes, each of which performs the same transaction function $txnfn(q)$ on the shared counter by calling the API public functions. This case system mainly performs two functions, in addition to the transaction function $txnfn(q)$ mentioned above, there is also a main function called *main*().

The main process executes the function *main*(), and we give Python pseudocode shown in Algorithm 1. First, we define the number of processes as $NP = 6$. A queue q is then created to connect the request process to the server side. Next, we create a master process *ptm* to make some preparation that is to add the shared counter to the system dictionary and initialize it. We call the function *addVars*() to add a dictionary item with the *counter* as the key to system dictionary, after which the version of this item equals to 0. The shared variable can be updated by *NP* transaction processes, and the function *putVars*() is called to initialize it as shown in line 8 of the pseudocode. Now the version is 1 and the value is 0. The assertion of line 9 is to ensure the success of the initialization. Next, we create *NP* request processes which share the target function *txnFn*() and the connection channel q. Then, we run all processes and wait for them to finish. The last one to be called is the function *getVars*() to obtain the value of the shared variable. Line 22 asserts whether the value of the shared variable is equal to the number of *NP* at the moment. If they are equal, *NP* processes are proved to execute successfully. Otherwise, based on the current validation results, the PSTM architecture does not fully guarantee correct concurrency. Finally, after ensuring that multiple transaction processes have completed execution, the master process is ended.

All the processes except the master process run the same target function $txnFn(q)$, as shown in Algorithm 2. Within this function, *NP* transaction processes execute the code from line 3 to line 12 in parallel, and do not exit the while loop until the updated results are committed successfully. Within this while loop, each process: (1) calls the function *getVars*() to get triplet of the shared counter which holds *counter* as the key, and asserts the existence of this triplet. If so, the value of the triplet is updated. (2) is ready to commit, and put a new triplet which has the key *counter*, the version *version*, the updated value *new_counter_value* into the *writeList*. The *readList* is empty at this time because there is no read operation. (3) calls the function *commitVars*() to commit all the update operations until success.

In summary, each process increments the current value of the shared counter once. Therefore, if *NP* processes execute in parallel without any confusion or interruption, then the final value of the shared counter should be exactly *NP*. So we give a corollary here, namely Corollary 1:

Algorithm 1 The function main().

```
1: def main() :
2: nP ← 6
3: q ← Queue()
4: ptm ← new(q)
5: addVars(q, ['counter'])
6: counter_version ← 0
7: counter ← 0
8: ret ← putVars(q, [('counter', counter_version, counter)])
9: assert(ret == ['yes'])
10: for i = 1 to nP do
11:     transactions = [Process(target = txnFn, args = (q, ))]
12: end for
13: for all transaction in transactions do
14:     transaction.start()
15: end for
16: for all transaction in transactions do
17:     transaction.join()
18: end for
19: ret ← getVars(q, ['counter'])
20: (counter_existence, (counter_version, counter_value)) ← ret[0]
21: print('The final counter value =', counter_value)
22: assert(counter_value == nP)
23: ret ← delete(q)
24: assert(ret == ['yes'])
25: ptm.join()
```

Algorithm 2 The function txnFn(q).

```
1: def txnFn(q) :
2: while True do
3:     ret ← getVars(q, ['counter'])
4:     (counter_existence, (counter_version, counter_value)) ← ret[0]
5:     assert(counter_existence)
6:     new_counter_value ← counter_value + 1
7:     readList ← [ ]
8:     writeList ← [('counter', counter_version, new_counter_value)]
9:     ret ← commitVars(q, [readList, writeList])
10:    if ret==['yes'] then
11:        break
12:    end if
13: end while
```

Corollary 1. *The final value of the shared counter should be exactly the number of the parallel processes.*

In this case study, the main goal we want to verify is whether the final value of the shared counter is equal to the number of concurrent processes after all processes have been executed. (Fig. 5)

5.2. Verification

We use the model checker PAT to simulate and validate the CSP model established in the previous subsection. We define two assertion goals in this section to verify the achieved model from positive and negative points of view. The first goal is *corollary*, which describes the final value of the dictionary item (i.e., shared counter) is equal to the number of parallel processes *NP*. In order to observe a more obvious verification result, we add the second goal *overCorollary*, that is, the final value of the dictionary term is equal to $NP + 1$ and is smaller than the initial value 1000 to exclude the possibility of not being executed. We give the definitions as follows:

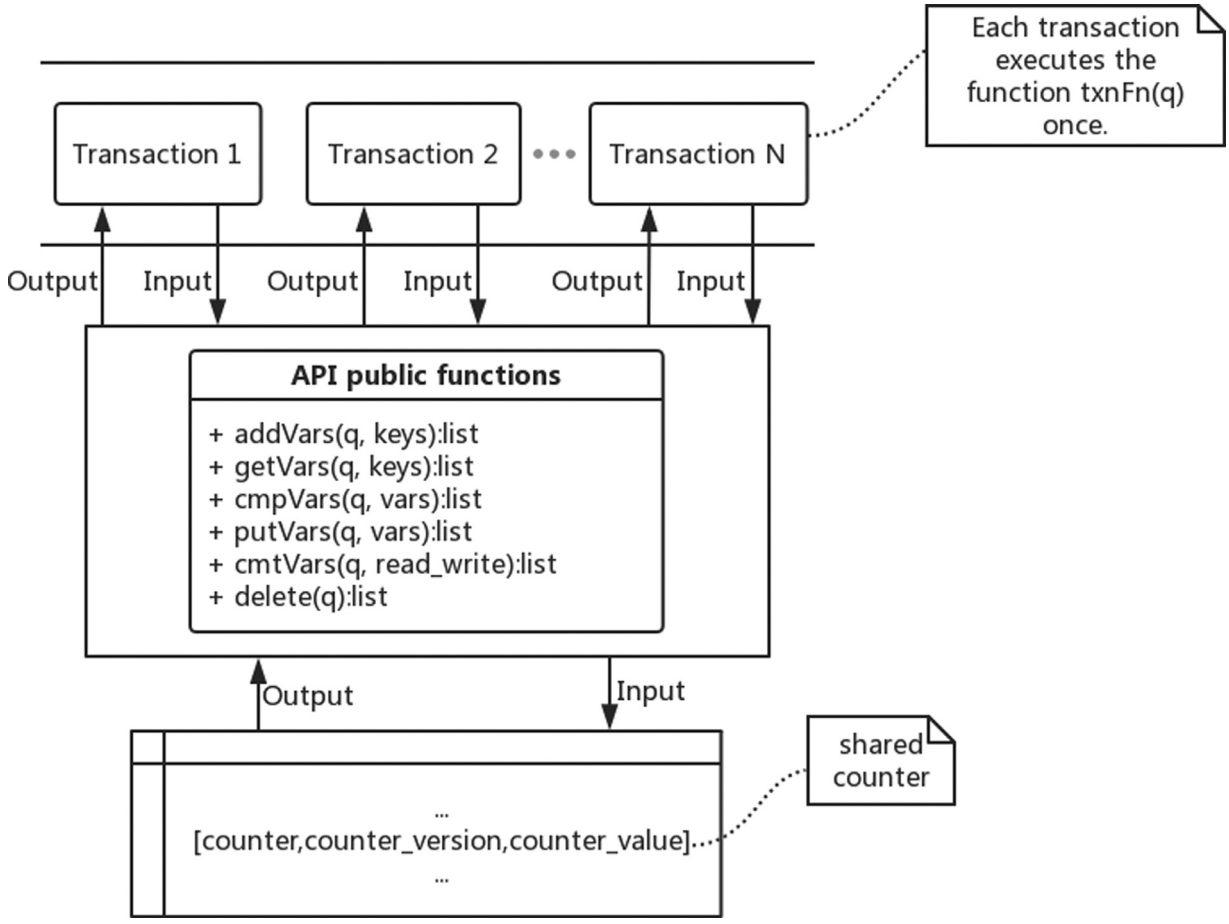

Fig. 5. The analysis of the case study.

$\#define\ corollary\ (dic[0][2] == nP)$;

$\#define\ overCorollary\ (dic[0][2] >= nP + 1\ \&\&\ dic[0][2] < 1000)$;

Similarly, we continue to use the primitive *assert* of the PAT to verify the case study system. At first, we need to verify that the *CaseStudy*() has a proper communication, in the sense that the system has no deadlocks.

$\#assert\ CaseStudy()\ deadlockfree$;

Then, in the case that the communication of the system process ensures no errors, we verify whether the system process can reach any verification goals mentioned above.

$\#assert\ CaseStudy()\ reaches\ corollary$;

$\#assert\ CaseStudy()\ reaches\ overCorollary$;

The verification results are shown in Fig. 6. Under the condition of no deadlocks happened in the case study system, the first goal *corollary* is valid (i.e. positive), and the second goal *overCorollary* is not valid (i.e. negative), which demonstrates that the final value of the dictionary item (i.e. shared counter) is indeed equal to the number of parallel processes *NP*. Fig. 6 also shows part of the traces PAT found.

6. Related work

The emergence of IBM blue Gene/Q computing cores (Haring et al., 2012) and the application of transaction synchronization techniques in Haswell TSH (2013), which marks the effort invested in the last two decades, began to pay off. Adir et al. focused on verifying transactional memory from the processor core layer in Adir et al. (2014). They verified the atomicity of transactions by detecting intermediate and final results, irregular traces, and test cases. During the functional verification of test cases, due to the limited number of different machine programs covered by the cases, they got partial certainty by simulating through randomly generated test cases and execution programs. El-kustaban et al. proposed a general specification model for transactional memory in (Cau et al., 2012), which allows interested researchers to make appropriate extensions based on the model. They built this model to develop a general, flexible formal framework to prove and verify the correctness of transactional memory systems. Abdulla et al. verified a composite transactional memory in (Abdulla et al, 2013). By adopting a small model approach, they have established some valuable properties, including strict serializability, abort consistency, and obstruction freedom for both an eager and a lazy conflict resolution strategies.

The study on verification of transactional memory has emerged in endlessly and each has its own strengths (Cohen et al., 2007; Manevich et al., 2010; Guerraoui et al., 2010; Doherty et al., 2013). From the work of HTM to STM, to the establishment of a composite systems, there has been a certain amount of researches in the world. If the studies mentioned above have one thing in common, that is the credible characterization of its subject. However, to the best of our knowledge, there is no formally verification work of PSTM architecture. We model the architecture vertically, which is also the challenge of this paper. That is to say, we pay more attention to model all the methods of the API including its internal calculation process rather than only focus on the communication between components.

Fig. 6. The verification results of case study.

The applications based on the PSTM architecture and subsequent in-depth studies have gradually increased. In (Kordic et al., 2015), Popovic et al. proposed a dual-end storage mechanism based on the PSTM architecture, which reduces the overhead of the queue and pipelines for message transmission mechanism introduced in the PSTM architecture, so that it can achieve better performance in the case of intensive transactions. While in (Basicevic et al., 2017), three transaction scheduling algorithms are proposed for the PSTM architecture, including Round Robin (RR) algorithm, the Execution Time based Load Balancing algorithm (ETLB) and the Avoid Conflicts algorithm (AC). Experimental results show that in the case of low contention and dual-core multicore, the RR algorithm performs worst, the ETLB algorithm is better than RR, and the AC algorithm has the best effect. Alternatively, in the

case of very short conflict-free transactions and high contention, the performances of three algorithms are comparable. Recently, we have (Xu et al., 2019) formalized the three transaction scheduling algorithms and verified some interesting properties (deadlock freeness and starvation freeness) of these models.

7. Conclusion and future work

This paper has presented how the PSTM architecture, as a novel framework based on Python language, guarantees some security of transactional memory.

- We have made a formal in-depth analysis and modeling of the PSTM architecture. The components of the PSTM architecture with their behaviors in the communication procedure are fully depicted, including the multi-level requests, the exported API component and the server component which provide circular services, and the system dictionary that stores items. We abstract and analyze the RPC interface to model the many-to-one and the one-to-many structures.
- We have simulated and verified five properties of the PSTM architecture with the model checker PAT. By improving some functions in the offered C# Library, we simplified coding of the model. After analyzing the characteristics of transactional properties embodied in the PSTM architecture, five properties including deadlock freeness, isolation, atomicity, consistency and optimism, are verified by judging the execution logic of communication procedure. The verification results and the specific traces found by PAT are given. All the properties are verified to be valid, indicating that the PSTM architecture can offer a proper and secured communication.
- We have applied the PSTM architecture to a specific application scenario with a shared counter, and also analyzed and modeled this case study formally. A corollary is put forward, indicating that the final value of the shared counter equals to the number of parallel processes. We verified whether the case study can satisfy this corollary from positive and negative two points of view. The results show that the corollary is true and the PSTM architecture is secured to use as well.

For the future, we are continuing to work on the formal modeling and verifying of novel architectures, systems or algorithms. We suppose this paper could also have some practical meanings, some "real world" examples could also be reached by these generalized models. We would pay more attention to this part in the future. With the deepening of researches on the PSTM architecture, its versions and scheduling algorithms are constantly optimized and updated (Cohen et al., 2007; Manevich et al., 2010; Guerraoui et al., 2010). The work is much more challenging and attractive. Further, we are also interested in whether the model achieved in this paper can be transplanted to the complex versions of the PSTM architecture or its algorithms.

CRediT authorship contribution statement

Ailun Liu: Methodology, Formal analysis. **Huibiao Zhu:** Formal analysis, Methodology. **Miroslav Popovic:** Formal analysis. **Shuangqing Xiang:** Methodology. **Lei Zhang:** Methodology.

Acknowledgements

This work was partly supported by National Key Research and Development Program of China (grant no. 2018YFB2101300), National Natural Science Foundation of China (grant no. 61872145), Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (grant no.ZF1213) and the Fundamental Research Funds for the Central Universities of China.

References

Adir, A., Hershkovitz, O., Hickerson, B., Holtz, K., Kadry, W., Koyfman, A., Ludden, J., Meissner, C., Nahir, A., Pratt, R.R., Schiffli, M., Onge, B.S., Thompto, B., Tsanko, E., Goodman, D., Hershcovich, D., Ziv, A., 2014. Verification of transactional memory in power8. In: 51st ACM/EDAC/IEEE Design Automation Conference (DAC).

Abdulla, P., Rezine, A., Shriraman, A., Zhu, Y., Dwarkadas, S., 2013. Verifying safety and liveness for the flextm hybrid transactional memory. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 785–790.

Basicevic, I., Popovic, M., Kordic, B., 2017. Transaction scheduling for software transactional memory. In: 2nd IEEE International Conference on Cloud Computing and Big Data Analysis, pp. 191–195.

Cohen, A., Leary, J.W.O, Zuck, L.D., 2007. Verifying correctness of transactional memories. In: Proceedings of the 7th International Conference on Formal Methods in Computer - Aided Design (FMCAD), pp. 37–44.

Cau, A., El-kustaban, A., Moszkowski, B., 2012. Formalising of transactional memory using interval temporal logic (itl). In: Spring Congress on Engineering and Technology, pp. 1–6.

Doherty, S., Groves, L., Luchangco, V., Moir, M., 2013. Towards formally specifying and verifying transactional memory. Formal Asp. Comput. 25 (5), 769–799.

Goldstein, M., Gerber, R., 2011. A new hybrid algorithm for finding the lowest minima of potential surfaces: approach and application to peptides. J. Comput. Chem. 32 (9), 1785–1800.

Goodman, J.R., 1983. Using cache memory to reduce processor-memory traffic. Proceedings of the 10th Annual Symposium on Computer Architecture, pp. 124–131.

Gray, J., Reuter, A., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann.

Guerraoui, R., Henzinger, T.A., Singh, V., 2010. Model checking transactional memories. Distrib. Comput. 22 (3), 129–145.

Guerraoui, R., Maurice, H., Pochon, B., 2005. Toward a theory of transactional contention managers. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC, pp. 258–264.

Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Christ, N., Kim, C., 2012. The IBM blue gene/q compute chip. IEEE Micro 32 (2), 48–60.

Herlihy, M., Eliot, J., Moss, B., 1993. Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300.

Hoare, C.A.R., 1985. Communicating Sequential Processes. Prentice-Hall.

Kordic, B., Popovic, M., Basicevic, I., 2015. DPM-PSTM: dual-port memory based python software transactional memory. In: 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, pp. 126–129.

Kordic, B., Popovic, M., Ghilezan, S., Basicevic, I., 2017. An approach to formal verification of python software transactional memory. In: Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, ECBS, pp. 13:1–13:10.

Liu, A., Popovic, M., Zhu, H., 2017. Formalization and verification of the PSTM architecture. In: 24th Asia-Pacific Software Engineering Conference, APSEC, pp. 427–435.

Lowe, G., Roscoe, B., 1997. Using CSP to detect errors in the TMN protocol. IEEE Trans. Softw. Eng. 23 (10), 659–669.

Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K., 2008. Stamp: stanford transactional applications for multi-processing, pp. 35–46.

Manevich, R., Emmi, M., Majumdar, R., 2010. Parameterized verification of transactional memories. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 134–145.

M. Goldstein, E.F., Herlihy, M., Shavit, N., 2013. The art of multiprocessor programming.

Popovic, M., Kordic, B., 2014. PSTM: python software transactional memory. In: 2014 22nd Telecommunications Forum Telfor (TELFOR), pp. 1106–1109.

Process analysis toolkit, http://pat.comp.nus.edu.sg/.

Roscoe, A.W., Huang, J., 2013. Checking noninterference in timed CSP. Formal Asp. Comput. 25 (1), 3–35.

Shavit, N., Touitou, D., 1997. Software transactional memory. Distrib. Comput. 10 (2), 99–116.

Si, Y., Sun, J., Liu, Y., Dong, J.S., Pang, J., Zhang, S., Yang, X., 2014. Model checking with fairness assumptions using PAT. Front. Comput. Sci. 8 (1), 1–16.

Sun, J., Liu, Y., Dong, J., 2008. Model checking CSP revisited: introducing a process analysis toolkit. In: Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, pp. 307–322.

Sun, J., Liu, Y., Dong, J. S., Pang, J., PAT: towards flexible verification under fairness. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26, - July 2, pp. 709–714.

Transactional synchronization in haswell 2013[online] available, http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

Xu, C., Wu, X., Zhu, H., Popovic, M., 2019. Modeling and verifying transaction scheduling for software transactional memory using CSP. In: 13th Theoretical Aspects of Software Engineering. IEEE Computer Society, pp. 1106–1109.

Yuan, T., Tang, Y., Wu, X., Zhang, Y., Zhu, H., Guo, J., Qin, W., 2014. Formalization and verification of REST on HTTP using CSP. Electr. Notes Theor. Comput. Sci. 309, 75–93.